Text Analysis *with* LingPipe 4

Text Analysis *with* LingPipe 4

Bob Carpenter
Breck Baldwin

LingPipe Publishing
New York
2011

Pending: *Library of Congress Cataloging-in-Publication Data*

Carpenter, Bob.
Text Analysis with LingPipe 4.0 / Bob Carpenter, Breck Baldwin
p. cm.
Includes bibliographical references and index.
ISBN X-XXX-XXXXX-X (pbk.)

1. Natural Language Processing 2. Java (computer program language) I. Carpenter, Bob II. Baldwin, Breck III. Title

QAXXX.XXXXXXXX 2011

XXX/.XX'X-xxXX XXXXXXXX

Contents

Preface

LingPipe is a software library for natural language processing implemented in Java. This book explains the tools that are available in LingPipe and provides examples of how they can be used to build natural language processing (NLP) applications for multiple languages and genres, and for many kinds of applications.

LingPipe's application programming interface (API) is tailored to abstract over low-level implementation details to enable components such as tokenizers, feature extractors, or classifiers to be swapped in a plug-and-play fashion. LingPipe contains a mixture of heuristic rule-based components and statistical components, often implementing the same interfaces, such as chunking or tokenization.

The presentation here will be hands on. You should be comfortable reading short and relatively simple Java programs. Java programming idioms like loop boundaries being inclusive/exclusive and higher-level design patterns like visitors will also be presupposed. More specific aspects of Java coding relating to text processing, such as streaming I/O, character decoding, string representations, and regular expression processing will be discussed in more depth. We will also go into some detail on collections, XML/HTML parsing with SAX, and serialization patterns.

We do not presuppose any knowledge of linguistics beyond a simple understanding of the terms used in dictionaries such as words, syllables, pronunciations, and parts of speech such as noun and preposition. We will spend considerable time introducing linguistic concepts, such as word senses or noun phrase chunks, as they relate to natural language processing modules in LingPipe.

We will do our best to introduce LingPipe's modules and their application from a hands-on practical API perspective rather than a theoretical one. In most cases, such as for logistic regression classifiers and conditional random field (CRF) taggers and chunkers, it's possible learn how to effectively fit complex and useful models without fully understanding the mathematical basis of LingPipe's estimation and optimization algorithms. In other cases, such as naive Bayes classifiers, hierarchical clusterers and hidden Markov models (HMM), the models are simpler, estimation is a matter of counting, and there is almost no hand-tuning required.

Deeper understanding of LingPipe's algorithms and statistical models requires familiarity with computational complexity analysis and basic probability theory including information theory. We provide suggested readings in algorithms, statistics, machine learning, and linguistics in Appendix E.

After introducing LingPipe's modules and their applications at a programming level, we will provide mathematically precise definitions. The goal is a complete description of the mathematical models underlying LingPipe. To understand these sections will require a stronger background in algorithms and statistical modeling than the other sections of the book.

We hope you have as much fun with LingPipe as we've had. We'd love to hear from you, so feel free to drop us a line at `info@lingpipe.com` with any kind of question or comment you might have.

If you'd like to join the LingPipe mailing list, it's at

    http://tech.groups.yahoo.com/group/LingPipe/

We also have a blog at

    http://lingpipe-blog.com/

Bob Carpenter
Breck Baldwin
New York
July 6, 2012

Text Analysis *with* LingPipe 4

Chapter 1

Getting Started

This book is intended to provide two things. First and foremost, it presents a hands-on introduction to the use of the LingPipe natural language processing library for building natural language processing components and applications. Our main focus is on programming with the LingPipe library. Second, the book contains precise mathematical definitions of all of the models and substantial algorithms underlying LingPipe. Understanding the finer mathematical points will not be assumed for the practical sections of the book. Of course, it's impossible to fully sidestep mathematics while explaining what is essentially a collection of algorithms implementing mathematical models of language.

Before beginning with a "hello world" example that will verify that everything's in place for compiling and executing the code in the book, we'll take some time to explain the tools we'll be using or that you might want to use. All of these tools are open source and freely available, though under a perplexing array of licensing terms and conditions. After describing and motivating our choice of each tool, we'll provide downloading and installation instructions.

1.1 Tools of the Trade

In this section, we go over the various tools we use for program development. Not all of these tools are needed to run every example, and in many cases, we discuss several alternatives among a large set of examples.

1.1.1 Unix Shell Tools

We present examples as run from a unix-like command-line shell. One reason to use the shell is that it's easy to script operations so that they may be repeated.

The main problem with scripts is that they are not portable across operating systems or even types of command-line shells. In this book, we use the so-called "Bourne Again Shell" (bash) for Unix-style commands. Bash is the default shell in Cygwin on Windows, for Mac OS X, and for Linux, and is also available for Solaris.

If you're working in Windows (XP, Vista or 7), we recommend the Cygwin suite of Unix command-line tools for Windows.

Cygwin is released under version 2 of the GNU Public License (GPLv2), with terms outlined here:

`http://cygwin.com/license.html`

and also available with a commercial license.

Cygwin can be downloaded and installed through Cygwin's home page,

`http://www.cygwin.com/`

The `setup.exe` program is small. When you run it, it goes out over the internet to find the packages from registered mirrors. It then lists all the packages available. You can install some or all of them. You can just run the setup program again to update or install new packages; it'll show you what version of what's already installed.

It might also be easier to list what Cygwin doesn't support. We use it mainly for running Unix-like commands in Windows, including pipes, find, grep, (bash) shell support, tar and gzip, wget, aspell, which, and so on. We also use its implementation of the Subversion and CVS version control systems. We do not install emacs or TeX through Cygwin. These packages are indexed under various categories in Cygwin, such as Web, Net, Devel, and Base.

Although it's possible to get Python, Perl, Emacs, $\TeX$ and $\LaTeX$, and so on, we typically install these packages directly rather than as part of Cygwin.

Archive and Compression Tools

In order to unpack data and library distributions, you need to be able to run the `tar` archiving tool, as well as the unpacking commands `unzip` and `gunizp`. These may be installed as part of Cygwin on Windows.

1.1.2 Version Control

If you don't live in some kind of version control environment, you should. Not only can you keep track of your own code across multiple sites and/or users, you can also keep up to date with projects with version control, such as this book, the LingPipe sandbox, and projects hosted by open source hosting services such as SourceForge or Google Code.

We are currently using Subversion (SVN) for LingPipe and this book. You can install a shell-based version of Subversion, the command-name for which is `svn`. Subversion itself requires a secure shell (SSH) client over which to run, at least for most installations. Both SVN and SSH can be installed through Cygwin for Windows users.

There are graphical interfaces for subversion. Web-SVN, which as its name implies, runs as a server and is accessible through a web browser,

WebSVN: `http://websvn.tigris.org/`

and Tortoise SVN, which integrates with the Windows Explorer,

Tortoise SVN: `http://tortoisesvn.net/`

Other popular version control systems include the older Concurrent Version System (CVS), as well as the increasingly popular Git system, which is used by the Linux developers.

The best reference for Subversion of which we are aware is the official guide by the authors, available onlie at

```
http://svnbook.red-bean.com/
```

1.1.3 Text Editors

In order to generate code, HTML, and reports, you will need to be able to edit text. We like to work in the emacs text editor, because of its configurability. It's as close as you'll get to an IDE in a simple text editor.

Spaces, Not Tabs

To make code portable, we highly recommend using spaces instead of tabs. Yes, it takes up a bit more space, but it's worth it. We follow Sun's coding standard for Java, so we use four spaces for a tab. This is a bit easier to read, but wastes more horizontal space.

(GNU) Emacs

We use the GNU Emacs distribution of emacs, which is available from its home page,

```
http://www.gnu.org/software/emacs/
```

It's standard on most Unix and Linux distributions; for Windows, there is a zipped binary distribution in a subdirectory of the main distribution that only needs to be unpacked in order to be run. We've had problems with the Cygwin-based installations in terms of their font integration. And we've also not been able to configure XEmacs for Unicode, which is the main reason for our preference for GNU Emacs.

We like to work with the Lucida Console font, which is distributed with Windows; it's the font used for code examples in this book. It also supports a wide range of non-Roman fonts. You can change the font by pulling down the `Options` menu and selecting the `Set Default Font...` item to pop up a dialog box. Then use the `Save Options` item in the same menu to save it. It'll show you where it saved a file called `.emacs` which you will need to edit for the next customizations.

In order to configure GNU Emacs to run UTF-8, you need to add the following text to your `.emacs` file:[1]

```
(prefer-coding-system 'utf-8)
(set-default-coding-systems 'utf-8)
(set-terminal-coding-system 'utf-8)
```

[1]The UTF-8 instructions are from the Sacrificial Rabbit blog entry `http://blog.jonnay.net/archives/820-Emacs-and-UTF-8-Encoding.html`, downloaded 4 August 2010.

```
(set-keyboard-coding-system 'utf-8)
(setq default-buffer-file-coding-system 'utf-8)
(setq x-select-request-type
    '(UTF8_STRING COMPOUND_TEXT TEXT STRING))
(set-clipboard-coding-system 'utf-16le-dos)
```

The requisite commands to force tabs to be replaced with spaces in Java files are:

```
(defun java-mode-untabify ()
  (save-excursion
    (goto-char (point-min))
    (if (search-forward "t" nil t)
        (untabify (1- (point)) (point-max))))
  nil)

(add-hook 'java-mode-hook
      '(lambda ()
         (make-local-variable 'write-contents-hooks)
         (add-hook 'write-contents-hooks 'java-mode-untabify)))

(setq indent-tabs-mode nil)
```

1.1.4 Java Standard Edition 6

We chose Java as the basis for LingPipe because we felt it provided the best tradeoff among efficiency, usability, portability, and library availability.

The presentation here assumes the reader has a basic working knowledge of the Java programming language. We will focus on a few aspects of Java that are particularly crucial for processing textual language data, such as character and string representations, input and output streams and character encodings, regular expressions, parsing HTML and XML markup, etc. In explaining LingPipe's design, we will also delve into greater detail on general features of Java such as concurrency, generics, floating point representations, and the collection package.

This book is based on the latest currently supported standard edition of the Java platform (Java SE), which is version 6. You will need the Java development kit (JDK) in order to compile Java programs. A java virtual machine (JVM) is required to execute compiled Java programs. A Java runtime environment (JRE) contains platform-specific support and integration for a JVM and often interfaces to web browsers for applet support.

Java is available in 32-bit and 64-bit versions. The 64-bit version is required to allocate JVMs with heaps larger than 1.5 or 2 gigabytes (the exact maximum for 32-bit Java depends on the platform).

Licensing terms for the JRE are at

    http://www.java.com/en/download/license.jsp

and for the JDK at

    http://java.sun.com/javase/6/jdk-6u20-license.txt

Java is available for the Windows, Linux, and Solaris operating systems in both 32-bit and 64-bit versions from

```
http://java.sun.com/javase/downloads/index.jsp
```

The Java JDK and JRE are included as part of Mac OS X. Updates are available through

```
http://developer.apple.com/java/download/
```

Java is updated regularly and it's worth having the latest version. Updates include bug fixes and often include performance enhancements, some of which can be quite substantial.

Java must be added to the operating system's executables path so that the shell commands and other packages like Ant can find it.

We have to manage multiple versions of Java, so typically we will define an environment variable `JAVA_HOME`, and add `${JAVA_HOME}/bin` (Unix) or `%JAVA_HOME%\bin` (Windows) to the `PATH` environment variable (or its equivalent for your operating system). We then set `JAVA_HOME` to either `JAVA_1_5`, `JAVA_1_6`, or `JAVA_1_7` depending on use case. Note that `JAVA_HOME` is one level above Java's `bin` directory containing the executable Java commands.

You can test whether you can run Java with the following command, which should produce similar results.

```
> java -version

java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) 64-Bit Server VM (build 16.0-b13, mixed mode)
```

Similarly, you can test for the Java compiler version, using

```
> javac -version

javac 1.6.0_20
```

Java Source

You can download the Java source files from the same location as the JDK. The source is subject to yet anoher license, available at

```
http://java.sun.com/javase/6/scsl_6-license.txt
```

The source provides a fantastic set of examples of how to design, code, and document Java programs. We especially recommend studying the source for strings, I/O and collections. Further, like all source, it provides a definitive answer to the questions of what a particular piece of code does and how it does it. This can be useful for understanding deeper issues like thread safety, equality and hashing, efficiency and memory usage.

1.1.5 Ant

We present examples for compilation and for running programs using the Apache Ant build tool. Ant has three key features that make it well suited for expository purposes. First, it's portable across all the platforms that support Java. Second, it provides clear XML representations for core Java concepts such as classpaths and command-line arguments. Third, invoking a target in Ant directly executes the dependent targets and then all of the commands in the target. Thus we find Ant builds easier to follow than those using the classic Unix build tool Make or its modern incarnation Apache Maven, both of which attempt to resolve dependencies among targets and determine if targets are up to date before executing them.

Although we're not attempting to teach Ant, we'll walk through a basic Ant build file later in this chapter to give you at least a reading knowledge of Ant. If you're using an IDE like Eclipse or NetBeans, you can import Ant build files directly to create a project.

Ant is is an Apache project, and as such, is subject to the Apache license,

```
http://ant.apache.org/license.html
```

Ant is available from

```
http://ant.apache.org/
```

You only need one of the binary distributions, which will look like `apache-ant-`*`version`*`-bin.tar.gz`.

First, you need to unpack the distribution. We like directory structures with release names, which is how ant unpacks, using top-level directory names like `apache-ant-1.8.1`. Then, you need to put the `bin` subdirectory of the top-level directory into the `PATH` environment variable so that Ant may be executed from the command line.

Ant requires the `JAVA_HOME` environment variable to be set to the path above the `bin` directory containing the Java executables. Ant's installation instructions suggest setting the `ANT_HOME` directory in the same way, and then adding but it's not necessary unless you will be scripting calls to Ant.

Ant build files may be imported directly into either the Eclipse or NetBeans IDEs (see below for a description of these).

You can test whether Ant is installed with

```
> ant -version
Apache Ant version 1.8.1 compiled on April 30 2010
```

1.1.6 Integrated Development Environment

Many pepeople prefer to write (and compile and debug) code in an integrated development environment (IDE). IDEs offer advantages such as automatic method, class and constant completion, easily configurable text formatting, stepwise debuggers, and automated tools for code generation and refactoring.

LingPipe development may be carried out through an IDE. The two most popular IDEs for Java are Eclipse and NetBeans.

Eclipse IDE

Eclipse provides a full range of code checking, auto-completion and code generation, and debugging facilities. Eclipse is an open-source project with a wide range of additional tools available as plugins. It also has modules for languages other than Java, such as C++ and PHP.

The full set of Eclipse downloads is listed on the following page,

```
http://download.eclipse.org/eclipse/downloads/
```

You'll want to make sure you choose the one compatible with the JDK you are using. Though originally a Windows-based system, Eclipse has been ported to Mac OS X (though Carbon) and Linux.

Eclipse is released under the Eclipse Public License (EPL), a slightly modified version of the Common Public License (CPL) from IBM, the full text of which is available from

```
http://www.eclipse.org/legal/epl-v10.html
```

NetBeans IDE

Unlike Eclipse, the NetBeans IDE is written entirely in Java. Thus it's possible to run it under Windows, Linux, Solaris Unix, and Mac OS X. There are also a wide range of plug-ins available for NetBeans.

Netbeans is free, and may be downloaded from its home page,

```
http://netbeans.org/
```

Its licensing is rather complex, being released under a dual license consisting of the Common Development and Distribution License (CDDL) and version 2 of the GNU Public License version 2 (GPLv2). Full details are at

```
http://netbeans.org/about/legal/license.html
```

1.1.7 Statistical Computing Environment

Although not strictly necessary, if you want to draw nice plots of your data or results, or compute p values for some of the statistics LingPipe reports, it helps to have a statistical computing language on hand. Such languages are typically interpreted, supported interactive data analysis and plotting.

In some cases, we will provide R code for operations not supported in LingPipe. All of the graphs drawn in the book were rendered as PDFs in R and included as vector graphics.

R Project for Statistical Computing

The most popular open-source statistical computing language is still the R Project for Statistical Computing (R). R runs under all the major operating systems, though the Windows installers and compatibility with libraries seem to be a bit more advanced than those for Linux or Mac OS X.

R implements a dialect of the S programming language for matrices and statistics. It has good built-in functionality for standard distributions and matrix operations. It also allows linking to C or Fortran back-end implementations. There are also plug ins (of varying quality) for just about everything else you can imagine. The officially sanctioned plug ins are available from the Comprehensive R Archive Network (CRAN).

R is available from its home page,

```
http://www.r-project.org/
```

The CRAN mirrors are also linked from the page and usually the first web search results for statistics-related queries in R.

R is distributed under the GNU Public License version 2 (GPLv2).

SciPy and NumPy

The Python programming/scripting language is becoming more and more popular for both preprocessing data and analyzing statistics. The library NumPy provides a matrix library on top of Python and SciPy a scientific library including statistics and graphing. The place to get started is the home page,

```
http://numpy.scipy.org/
```

NumPy and SciPy and Python itself are distributed under the much more flexible BSD License.

Like R, Python is interpreted and not very fast at operations like looping. NumPy helps by allowing some operations to be vectorized (as does R). Like R, Python also allows linking C and Fortran back-end packages for efficiency.

1.1.8 Full LingPipe Distribution

LingPipe is distributed under both commercial licenses and under a royalty-free license. A copy of the royalty free license is available at:

```
http://alias-i.com/lingpipe/licenses/lingpipe-license-1.txt
```

LingPipe may be downloaded with full source from its web site:

```
http://alias-i.com/lingpipe/
```

Other than unpacking the gzipped tarball, there is nothing required for installation. Downloading LingPipe is not technically necessary for running the examples in this book because the LingPipe library jar is included with this book's source download.

1.1.9 Book Source and Libraries

The code samples from this book are available via anonymous subversion checkout from the LingPipe sandbox. Specifically, the entire content of the book, including code and text, may be checked out anonymously using,

```
> svn co https://aliasi.devguard.com/svn/sandbox/lpbook
```

The distribution contains all of the code and Ant build files for running it. It also contains the LaTeX source for the book itself as well as the library we created to extract text from the source for automatic inclusion in the book. This last feature is actually critical to ensure the code in the distribution and in the book match.

1.2 Hello World Example

In this section, we provide a very simple hello world program. If you can run it, you'll be all set to jump into the next chapter and get started with the real examples. We'll also take this opportunity to walk through a simple Ant build file.

1.2.1 Running the Example

To the extent we are able, we'll start each discussion with an example of how to run examples. This is pretty easy for the hello-world example. As with all of our examples, we begin by changing directories into the directory containing the example. We suppose that `$BOOK/` is the directory in which the code for the book was unpacked. We then execute the change directory (`cd`) command to move into the subdirectory `/src/intro`,

```
> cd $BOOK/src/intro
```

Note that we have italicized the commands issued by the user. We then run the demo using Ant. In this case, the target is `hello`, invoked by

```
> ant hello
```

which produces the following output

```
Buildfile: c:\lpb\src\intro\build.xml

jar:
    [mkdir] Created dir: c:\lpb\src\intro\build\classes
    [javac] Compiling 1 source file to c:\lpb\src\intro\build\classes
      [jar] Building jar: c:\lpb\src\intro\build\lp-book-intro-4.0.jar

hello:
     [java] Hello World

BUILD SUCCESSFUL
Total time: 1 second
```

First, Ant echoes the name of the build file, here `c:\lpb\src\intro\build.xml`. When Ant is invoked without specifying a particular build file, it uses the `build.xml` in the directory from which it was called.

Reading down the left side of the output, you see the targets that are invoked. The first target invoked is the `jar` target, and the second target is the `hello` target. The targets are left aligned and followed by semicolons. A target consists of an ordered list of dependent targets to invoke before the current target, and an ordered list of tasks to execute after the dependent targets are invoked.

Under the targets, you see the particular tasks that the target executes. These are indented, square bracketed, and right aligned. The `jar` target invokes three tasks, `mkdir`, `javac`, and `jar`. The `hello` target invokes one task, `java`. All of the output for each task is shown after the task name. If there is more than one line of output, the name of the task is repeated.

In this example, the `mkdir` task makes a new directory, here the `build\classes` directory. The `javac` task runs the Java compiler, here compiling a single source file into the newly created directory. The `jar` task builds the Java library into the build subdirectory `build` in file `lp-book-intro-4.0.jar`. Moving onto the `hello` target, the `java` task runs a command-line Java program, in this case printing the output of the hello world program.

The reason the `jar` target was invoked was because the `hello` target was defined to depend on the `jar` target. Ant invokes all dependent targets (recursively) before the invoked target.

Finally, Ant reports that the build was successful and reports the amount of time taken. In the future, we will usually only show the output of the Java program executed and not all of the output from the compilation steps. To save more space, we also remove the `[java]` tag on the task. Under this scheme, the `hello` target invocation would be displayed in this book as

```
> ant hello

Hello World
```

1.2.2 Hello with Arguments

Often we need to supply arguments to programs from the command line. The easiest way to do this with Ant is by setting the properties on the command line. Our second demo uses this facility to provide a customized greeting.

```
> ant -Dfirst=Bobby -Dlast=C hello-name

Hello, Bobby C.
```

Each argument to a program corresponds to a named property, here `first` and `last`. In general, properties are specified `-D`*`key`*`=`*`val`*.[2]

1.2.3 Code Walkthrough

The code for the hello world program can be found in `$BOOK/src/intro/src/com/lingpipe/book/intro/HelloWorld.java`. Throughout the book, the files are organized this way, under the top-level `src` directory, then the name of the chapter (here `intro`), followed by `src`, followed by the path to the actual program. We follow the Java convention of

[2]Depending on your shell, this may also require varying amounts of quoting and/or escaping special characters. For example, keys or values with spaces typically require double quotes on Windows.

placing files in a directory structure that mirrors their package structure, so the remaining part of the path is `com/lingpipe/book/intro/HelloWorld.java`. The contents of the `HelloWorld.java` program file is as follows.

```
package com.lingpipe.book.intro;

public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World");
    }

}
```

As usual for Java programs, the first thing in the file is the package declaration, here `com.lingpipe.book.intro`; note how it matches the end of the path to the file, `com/lingpipe/book/intro`. Next is the class definition, `HelloWorld`, which matches the name of the file, `HelloWorld.java`.

When a Java class is invoked from the command line (or equivalently through Ant), it must implement a static method with a signature like the one shown. Specifically, the method must be named `main`, must take a single string array argument of type `String[]`, must not have a return value (i.e., return type `void`), and must be declared to be both public and static using the `public` and `static` modifiers. The method may optionally throw any kind of exception.

The body of the program here just prints `Hello World` to the system output. As usual, Java provides a handle to the system output through the public static variable `out` of type `java.io.PrintStream` in the `java.lang.System` class.

The more complex program `HelloName` that takes two command-line arguments varies in the body of its `main()` method,

```
public class HelloName {

    public static void main(String[] args) {
        String first = args[0];
        String last = args[1];
        System.out.printf("Hello, %s %s.\n",first,last);
```

As a convention, we assign the command-line arguments read from the array `args` to a sequence of named variables. We then use these same names in the Ant properties (see Section 1.3.3).

1.3 Introduction to Ant

Although we will typically skip discussing the Ant build files, as they're almost all the same, we will go over the one for the hello world program in some detail.

The build file for hello world is located at `$BOOK/src/intro/build.xml`, where `$BOOK/` is the root of the directory in which this book's files have been unpacked. In general, the build files will be broken out by chapter and/or section. The goal is to make each of them relatively self contained so that they may be used as a minimal basis for further development.

1.3.1 XML Declaration

The first thing in any Ant build file is the XML declaration, here

```
<?xml version="1.0" encoding="ASCII"?>
...
```

This just tells Ant's XML parser that what it's looking at is an XML file and that the ASCII character encoding is being used for subsequent data. We chose ASCII because we didn't anticipate using any non-ASCII characters; we could have chosen Latin1 or UTF-8 or even Big5 and written the build file in Chinese. Most XML parsers are robust enough to infer the character encoding, but it's always good practice to make it explicit.

The ellipses (`...`) indicate ellided material that will be filled in (or not) in continuing discussion.

1.3.2 Top-Level Project Element

The top-level element in the XML file is the project declaration, which is just

```
<project>
...
</project>
```

Given our convention for ellipses, the file actually looks as follows,

```
<?xml version="1.0" encoding="ASCII"?>
<project>
...
</project>
```

with the remaining ellipses to be filled in below.

The project element's tag is `project`, and there are no required attributes. The project declaration may optionally be given a name as the value of attribute `name`, a default target to invoke as the value of attribute `default`, and a base directory from which to run as the value of `basedir`. The base directory defaults to the directory containing the build file, which is where we set everything up to run from. We will not need a default target as we will specify all targets explicitly for clarity. The name doesn't really have a function.

1.3.3 Ant Properties

We organize Ant build files in the conventional way starting with properties, which are declared first in the project element's content. The properties define constant values for reuse. Here, we have

```
  <property name="version"
            value="4.0"/>

  <property name="jar"
            value="build/lp-book-intro-${version}.jar"/>
```

The first property definition defines a property `version` with value `4.0` (all values are strings). The second property is named `jar`, with a value, defined in terms of the first property, of `build/lpb-intro-4.0.jar`. note that the value of the property `version` has been substituted for the substring `${version}`. In general, properties are accessed by `${...}` with the property filled in for the ellipses.

Properties may be overridden from the command line by declaring an environment variable for the command. For example,

```
> ant -Djar=foo.jar jar
```

calls the build file, setting the value of the `jar` property to be the string `foo.jar` (which would create a library archive called `foo.jar`). The value of a property in Ant is always the first value to which it is set; further attempts to set it (as in the body of the Ant file) will be ignored.

1.3.4 Ant Targets

Next, we have a sequence of targets, each of which groups together a sequence of tasks. The order of targets is not important.

Clean Target

The first target we have performs cleanup.

```
<target name="clean">
  <delete dir="build"/>
</target>
...
```

This is a conventional clean-up target, given the obvious name of `clean`. Each target has a sequence of tasks that will be executed whenever the target is invoked. Here, the task is a delete task, which deletes the directory named `build`.

It is conventional to have a clean task that cleans up all of the automatically generated content in a project. Here, the `.class` files generated by the compiler (typically in a deletable subdirectory `build`), and the `.jar` file produced by the Java archiver (typically in the top-level directory).

Compiliation/Archive Target

The next target is the compilation target, named `jar`,

```
<target name="jar">
  <mkdir dir="build/classes"/>
  <javac debug="yes"
         debuglevel="source,lines,vars"
         destdir="build/classes"
         includeantruntime="false">
    <compilerarg value="-Xlint:all"/>
```

```
      <src path="src/"/>
    </javac>
    <jar destfile="${jar}">
      <fileset dir="build/classes"
               includes="**/*.class"/>
    </jar>
  </target>
  ...
```

Invoking the `jar` target executes three tasks, a make-directory (`mkdir`), java compilation (`javac`), and Java archive creation (`jar`). Note that, as we exploit here, it is allowable to have a target with the same name as a task, because Ant keeps the namespaces separate.

The first task is the make-directory task, `mkdir`, takes the path for a directory and creates that directory and all of its necessary parent directories. Here, it builds the directory `build/classes`. All files are relative to the base directory, which is by default the directory containing the `build.xml` file, here `$BOOK/src/intro`.

The second task is the Java compilation task, `javac`, does the actual compilation. Here we have supplied the task element with four attributes. The first two, `debug` and `debuglevel` are required to insert debugging information into the compiled code so that we can get stack traces with line numbers when processes crash. These can be removed to save space, but they can be very helpful for deployed systems, so we don't recommend it. The `debug` element says to turn debugging on, and the `debuglevel` says to debug the source down to lines and variable names, which is the maximal amount of debugging information available. The `javac` task may specify the character encoding used for the Java program using the attribute `encoding`.

The destination directory attribute, `destdir` indicates where the compiled classes will be stored. Here, the path is `build/classes`. Note that this is in the directory we first created with the make-directory task. Further recall that the `build` directory will be removed by the clean target.

Finally, we have a flag with attribute `includeantruntime` that says the Ant runtime libraries should not be included in the classpath for the compilation. In our opinion, this should have defaulted to `false` rather than `true` and saved us a line in all the build files. If you don't specify this attribute, Ant gripes and tells you what it's defaulting to.

The java compilation task here has content, startign with a compiler argument. The element `compilerarg` passes the value of the attribute `value` to the underlying call to the `javac` executable in the JDK. Here, we specified a value of `-Xlint:all`. The `-X` options to `javac` are documented by running the command `javac -X`. This `-X` option specifies `lint:all`, which says to turn all lint detection on. This provides extended warnings for form, including deprecation, unchecked casts, lack of serial version IDs, lack or extra override specifications, and so on. These can be turned on and off one by one, but we prefer to leave them all on and produce lint-free programs. This often involves suppressing warnings that would otherwise be unavoidable, such as casts after object I/O or internal uses of deprecated public methods.

When compilation requires external libraries, we may add the classpath specifications either as attributes or elements in the `javac` task. For instance, we can have a classpath specification, which is much like a property definition,

```
<path id="classpath">
  <pathelement location="${jar}"/>
  <pathelement location="../../lib/icu4j-4_4_1.jar"/>
</path>
```

And then we'd add the element

```
<classpath refid="classpath"/>
```

as content in the `javac` target.

The java compilation task's content continues a source element with tag `src`. This says where to find the source code to compile. Here we specify the value of attribute for the path to the source, `path`, as `src/`. As usual, this path is interpreted relative to the base directory, which by default is the one holding the `build.xml` file, even if it's executed from elsewhere.

The third task is the Java archive task, `jar`, which packages up the compiled code into a single compressed file, conventionally suffixed with the string `.jar`. The file created is specified as the value of the `destfile` attribute, here given as `${jar}`, meaning the value of the `jar` property will be substituted, here `build/lpb-intro-4.0.jar`. As ever, this is interpreted relative to the base project directory. Note that the jar is being created in the `build` directory, which will be cleaned by the clean target.

Java Execution Task

The final target is the one that'll run Java,

```
<target name="hello"
        depends="jar">
  <java classname="com.lingpipe.book.intro.HelloWorld"
        classpath="${jar}"
        fork="true">
  </java>
</target>
```

Unlike the previous targets, this target has a dependency, as indicated by the `depends` attribute on the target element. Here, the value is `jar`, meaning that the `jar` target is invoked before the tasks in the `hello` target are executed. This means that the compilation and archive tasks are always executed before the `hello` target's task.

It may seem at this point that Ant is using some notion of targets being up to date. In fact, it's Java's compiler, `javac`, and Java's `jar` command which are doing the checking. In particular, if there is a compiled class in the compilation location that has a later date than the source file, it is not recompiled.[3] Similarly, the `jar` command will not rebuild the archive if all the files from which it were built are older than the archive itself.

In general, there can be multiple dependencies specified as target names separated by commas, which will be invoked in order before the target declaring the dependencies. Furthermore, if the targets invoked through `depends` themselves have dependencies, these will be invoked recursively.

The `hello` target specifies a single task as content. The task is a run-Java task, with element tag `java`. The attribute `classname` has a value indicating which class is executed. This must be a fully specified class with all of its package qualifiers separated by periods (`.`).

To invoke a Java program, we must also have a classpath indicating where to find the compiled code to run. In Ant, this is specified with the `classpath` attribute on the `java` task. The value here is `${jar}`, for which the value of the Java archive for the project will be substituted. In general, there can be multiple archives or directories containing compiled classes on the classpath, and the classpath may be specifeid with a nested element as well as an attribute. Ant contains an entire syntax for specifying path-like structures.

Finally, there is a flag indicated by attribute `fork` being set to value `true`, which tells Ant to fork a new process with a new JVM in which to run the indicated class.

The target `hello-name` that we used for the hello program with two arguments consists of the following Java task.

```
<java classname="com.lingpipe.book.intro.HelloName"
      classpath="${jar}"
      fork="true">
  <arg value="${first}"/>
  <arg value="${last}"/>
</java>
```

This time, the element tagged `java` for the Java task has content, specifically two argument elements. Each argument element is tagged `arg` and has a single attribute `value`, the value of which is passed to the named Java programs `main()` program as arguments. As a convention, our programs will all create string variables of the same name as their first action, just as in our hello program with arguments, which we repeat here,

```
public class HelloName {

    public static void main(String[] args) {
        String first = args[0];
        String last = args[1];
        System.out.printf("Hello, %s %s.\n",first,last);
```

[3]This leads to a confusing situation for statics. Static constants are compiled by value, rather than by reference if the value can be computed at compile time. These values are only recomputed when a file containing the static constant is recompiled. If you're changing the definition of classes on which static constants depend, you need to recompile the file with the constants. Just clean first.

In more elaborate uses of the `java` task, we can also specify arguments to the Java virtual machine such as the maximum amount of memory to use or to use ordinary object pointer compression, set Java system properties, and so on.

1.3.5 Property Files

Properties accessed in an Ant build file may also be specified in an external Java properties file. This is particular useful in a setting where many users access the same build file in different environments. Typically, the build file itself is checked into version control. If the build file specifies a properties file, each user may have their own version of the properties file.

Properties may be loaded directly from the build file, or they may be specified on the command line. From within an Ant build file, the `file` attribute in the `property` element may be used to load properties. For example,

```
<property file="build.properties"/>
```

From the command line, a properties file may be specified with

```
> ant -propertyfile build.properties ...
```

The properties file is interpreted as a Java properties file. For instance, we have supplied a demo properties file `demo.properties` in this chapter's directory we can use with the named greeting program. The contents of this file is

```
first: Jacky
last: R
```

We invoke the Ant target `hello-name` to run the demo, specifying that the properties from the file `demo.properties` be read in.

```
> ant -propertyfile demo.properties hello-name
Hello, Jacky R.
```

Any command-line specifications using `-D` override properties. For example, we can override the value of `first`,

```
> ant -Dfirst=Brooks -propertyfile demo.properties hello-name
Hello, Brooks R.
```

Parsing Properties Files

The parser for properties file is line oriented, allowing the Unix (`\n`), Macintosh (`\r\n` Windows (`\r`) line-termination sequences. Lines may be continued like on the shell with a backslash character.

Lines beginning with the hash sign # or exclamation point (`!`) are taken to be comments and ignored. Blank lines are also ignored.

Each line is interpreted as a key followed by a value. The characters equal sign (=), colon (`:`) or a whitespace character (there are no line terminators within lines; line parsing has already been done).

Character escapes, including Unicode escapes, are parsed pretty much as in Java string literals with a bit more liberal syntax, a few additional escapes (notably for the colon and equals sign) and some excluded characters (notably backspace).

1.3.6 Precedence of Properties

In Ant, whichever property is read first survives. The command line properties precede any other properties, followed by properties read in from a command-line properties file, followed by the properties read in from Ant build files in the order they appear.

Chapter 2

Handlers, Parsers, and Corpora

LingPipe uses a parser-handler pattern for parsing objects out of files and processing them. The pattern follows the design of XML's SAX parser and content handlers, which are built into Java in package `org.xml.sax`.

A parser is created for the format in which the data objects are represented. Then a handler for the data objects is created and attached to the parser. At this point, the parser may be used to parse objects from a file or other input specification, and all objects found in the file will be passed off to the handler for processing.

Many of LingPipe's modules provide online training through handlers. For instance, the dynamic language models implement character sequence handlers.

LingPipe's batch training modules require an entire corpus for training. A corpus represents an entire data set with a built-in training and test division. A corpus has methods that take handlers and deliver to them the training data, the test data, or both.

2.1 Handlers and Object Handlers

2.1.1 Handlers

The marker interface `Handler` is included in LingPipe 4 only for backward compatibility. The original parser/handler design allowed for handlers that implemented arbitrary methods, following the design of the SAX content handler interface.

2.1.2 Object Handlers

As of Lingpipe 4, all built-in parser/handler usage is through the `ObjectHandler<E>` interface, in package `com.aliasi.corpus`, which extends `Handler`. The `ObjectHandler<E>` interface specifies a single method, `handle(E)`, which processes a single data object of type `E`. It is like a simplified form of the built-in interface for processing a streaming XML document, `ContentHandler`, in package `org.xml.sax`.

2.1.3 Demo: Character Counting Handler

We define a simple text handler in the demo class `CountingTextHandler`. It's so simple that it doesn't even need a constructor.

```
public class CountingTextHandler
     implements ObjectHandler<CharSequence> {

     long mNumChars = 0L;
     long mNumSeqs = 0L;

     public void handle(CharSequence cs) {
         mNumChars += cs.length();
         ++mNumSeqs;
     }
```

We declare the class to implement `ObjectHandler<CharSequence>`. This contract is satisfied by our implementation of the `handle(CharSequence)` method. The method just increments the counters for number of characters and sequences it's seen.

The `main()` method in the same class implements a simple command-line demo. It creates a conting text handler, then calls is handle method with two strings (recall that `String` implements `CharSequence`).

```
CountingTextHandler handler = new CountingTextHandler();
handler.handle("hello world");
handler.handle("goodbye");
```

The Ant target `counting-text-handler` calls the main method, which assumes no command-line arguments.

```
> ant counting-text-handler

# seqs=2 # chars=18
```

The output comes from calling the handler's `toString()` method, which is defined to print the number of sequences and characters. In most cases, handlers will either hold references to collections they update or will make summary statistics or other data accumulated from the calls to the `handle()` method. We'll see more examples when we discuss parsers and corpora in the following sections.

2.2 Parsers

A parser reads character data from an input source or character sequence to produce objects of a specified type. By attaching a handler to a parser, the handler will receive a stream of objects produced by the parser.

2.2.1 The Parser Abstract Base Class

The type hierarchy for a parser is more general than usage in LingPipe 4 demands. The abstract base class is specified as `Parser<H extends Handler>`, in package `com.aliasi.corpus`. The generic specification requires the instantiation of the type parameter `H` to extend the `Handler` interface.

For use in LingPipe 4, all parsers will use object handler insances, so all parsers will actually extend `Parser<ObjectHandler<E>>` for some arbitrary object type `E`.

Getting and Setting Handlers

The base parser class implements getters and setters for handlers of the appropriate type; `getHandler()` returns the handler, and `setHandler(H)` sets the value of the handler.

Parsing Methods

There are a range of parsing methods specified for `Parser`. Two of them are abstract, `parseString(char[],int,int)` and `parse(InputSource)`, which read data from a character array slice or an input source. The input source abstraction, borrowed from XML's SAX parsers, generalizes the specification of a character sequence through a reader or an input stream or URL plus a character encoding specification; input sources also conveniently allow a relative directory to be specified for finding resources such as DTDs for XML parsing.

There are several helper methods specified on top of these. The method `parse(CharSequence)` reads directly from a character sequence. The `parse(File,String)` method reads the characters to parse from a file using the specified character encoding.[1] The method `parse(String,String)` reads from a resource specified as a system identifier, such as a URL, using the specified character encoding. There are also two parallel methods, `parse(File)` and `parse(String)`, which do not specify their character encodings.

2.2.2 Abstract Base Helper Parsers

There are two helper subclasses of `Parser`, both abstract and both in package `com.aliasi.corpus`. The class `StringParser` requires a concrete subclass to implement the abstract method `parse(char[],int,int)`. The `StringParser` class implements the `parse(InputSource)` method by reading a character array from the input source using LingPipe's static utility method `Streams.toCharArray(InputSource)`

The second helper subclass is `InputSourceParser`, which requires subclasses to implement `parse(InputSource)`. It implements `parse(char[],int,int)` by converting the character array to an input source specified through a reader.

[1]The encoding is set in an `InputSource`, which a parser implementation may choose to ignore. For instance, an XML parser may choose to read the encoding from the header rather than trusting the specification. The helper implementations in LingPipe all respect the character encoding declaration.

2.2.3 Line-Based Tagging Parser

There are two more specialized classes for parsing. The `LineTaggingParser`, also in `com.aliasi.corpus`, parses output for a tagger from a line-oriented format. This format is common due to the simplicity of its parsing and its use in the Conference on Natural Language Learning (CoNLL).

2.2.4 XML Parser

The `XMLParser`, in package `com.aliasi.corpus`, extends `InputSourceParser` to support parsing of XML-formatted data. The XML parser requires the method `getXMLHandler()` to return an instance of `DefaultHandler` (built into Java in package `org.xml.sax.helpers`).

The parser works by creating an XML reader using the static method `createXMLReader()` in `XMLReaderFactory`, which is in `org.xml.sax.helpers`. The XML handler returned by the parser method `getXMLHandler()` is then set on the reader as the content handler, DTD handler, entity resolver, and error handler. Thus the XML handler must actually hold a reference to the LingPipe handler or retain a reference to the parser (for instance, by being an inner class) and use the `getHandler()` method in `Parser` to pass it data.

2.2.5 Demo: MedPost Part-of-Speech Tagging Parser

As an example, we provide a demo class `MedPostPosParser`, which provides a parser for taggings represented in the MedPost format. The MedPost corpus is a collection of sentences drawn from biomedical research titles and abstracts in MEDLINE (see Section D.3 for more information and downloading instructions). This parser will extract instances of type `Tagging<CharSequence>` and send them to an arbitrary parser specified at run time.

Corpus Structure

The MedPost corpus is arranged by line. For each sentence fragment, [2] we have a line indicating its source in MEDLINE followed by a line with the tokens and part of speech tags. Here are the first three entries from file `tag_cl.ioc`.

```
P01282450A01
Pain_NN management_NN ,_, nutritional_JJ support_NN ,_, ...
P01350716A05
Seven_MC trials_NNS of_II beta-blockers_NNS and_CC 15_MC ...
P01421653A15
CONCLUSIONS_NNS :_: Although_CS more_RR well-designed_VVNJ ...
...
```

We have truncated the part-of-speech tagged lines as indicated by the ellipses (`...`). The source of the first entry is a MEDLINE citation with PubMed ID

[2]The sentence fragments were extracted automatically by a system that was not 100% accurate and were left as is for annotation.

01282450, with the rest of the annotation, A01, indicating it was the first fragment from the abstract. The first token in the first fragment is *Pain*, which is assigned a singular noun tag, NN. The second token is *management*, assigned the same tag. The third token is a comma, (,), assigned a comma (,) as a tag. Note that the string *beta-blockers* is analyzed as a single token, and assigned the plural noun tag NNS, as is *well-designed*, which is assigned a deverbal adjective category VVNJ.

Code Walkthrough

We define the MedPost parser to extend StringParser.

```
public class MedPostPosParser
     extends StringParser<ObjectHandler<Tagging<String>>> {
```

The generic parameter for the parser's handler is specified to be of type ObjectHandler<Tagging<String>>. LingPipe's Tagging class, which represents tags applied to a sequence of items, is in package com.aliasi.tag; we gloss over its details here as we are only using it as an example for parsing.

Because we extended StringParser, we must implement the parseString() method, which is done as follows.

```
@Override
public void parseString(char[] cs, int start, int end) {
     String in = new String(cs,start,end-start);
     for (String sentence : in.split("\n"))
         if (sentence.indexOf('_') >= 0)
             process(sentence);
}
```

We construct a string out of the character array slice passed in, switching from the start/end notation of LingPipe and later Java to the start/length notation of early Java. We then split it on Unix newlines, looking at each line in the for-each loop. If the line doesn't have an underscore character, there are no part-of-speech tags, so we ignore it (we also know there are no underscores in the identifier lines and that there are no comments).

We send each line in the training data with an underscore in it to the process() method.

```
private void process(String sentence) {
     List<String> tokenList = new ArrayList<String>();
     List<String> tagList = new ArrayList<String>();

     for (String pair : sentence.split(" ")) {
         String[] tokTag = pair.split("_");
         tokenList.add(tokTag[0]);
         tagList.add(tokTag[1]);
     }
     Tagging<String> tagging
         = new Tagging<String>(tokenList,tagList);
```

```
    ObjectHandler<Tagging<String>> handler = getHandler();
    handler.handle(tagging);
}
```

In order to construct a tagging, we will collect a list of tokens and a list of tags. We split the line on whitespace, and consider the strings representing token/tag pairs in turn. For instance, the first value for `pair` we see in processing the fragment above is `Pain_NN`.

We break the token/tag pair down into its token and tag components by splitting on the underscore character (again, keeping in mind that there are no underscores in the tokens). We then just add the first and second elements derived from the split, which are our token and tag pair.

After we add the tokens and tags from all the sentences, we use them to construct a tagging. Then, we use the `getHandler()` method, which is implemented by the superclass `Parser`, to get the handler itself. The handler then gets called with the tagging that's just been constructed by calling its `handle(Tagging)` method. In this way, the handler gets all the taggings produced by parsing the input data.

Even though we've implemented our parser over character array slices, through its superclass, we can use it to parse files or input sources. The `main()` method takes a single command-line argument for the directory containing the tagged data, comprising a set of files ending in the suffix `ioc`.

The first real work the method does is assign a new tree set of strings to the final variable `tagSet`.

```
final Set<String> tagSet = new TreeSet<String>();
ObjectHandler<Tagging<String>> handler

    = new ObjectHandler<Tagging<String>>() {

        public void handle(Tagging<String> tagging) {
            tagSet.addAll(tagging.tags());
        }

};
```

It is final because it's used in the anonymous inner class defiend to create an object to assign to variable `handler`. The anonymous inner class implements `ObjectHandler<Tagging<String>>`, so it must define a method `handle(Tagging<String>)`. All the handle method does is add the list of tags found in the tagging to the tag set.

Next, we create a parser and set its handler to be the handler we just defined.

```
Parser<ObjectHandler<Tagging<String>>> parser
    = new MedPostPosParser();
parser.setHandler(handler);
```

The last thing the main method does before printing out the tag set is walk over all the files ending in `ioc` and parse each of them, declaring the character encoding to be ASCII (see the section on I/O in the companion volume *Text Processing in Java* for more information on LingPipe's file extension filter class).

Running the Demo

There is an Ant target `medpost-parse` that passes the value of property `medpost.dir` to the command in `MedPostPosParser`. We have downloaded the data and unpacked it into the directory given in the command,

```
> ant -Dmedpost.dir=c:/lpb/data/medtag/medpost medpost-parse
```

```
#Tags=63    Tags=    ''    (    )    ,    .    :    CC    CC+
...
VVGJ    VVGN    VVI    VVN    VVNJ    VVZ    ``
```

We've skipped the output for most of the tags, where denoted by ellipses (`...`).

2.3 Corpora

A corpus, in the linguistic sense, is a body of data. In the context of LingPipe, a corpus of data may be labeled with some kind of conceptual or linguistic annotation, or it may consist solely of text data.

2.3.1 The Corpus Class

To represent a corpus, LingPipe uses the abstract base class `Corpus`, in package `com.aliasi.corpus`. It is parameterized generically the same way as a parser, with `Corpus<H extends Handler>`. As of LingPipe 4, we only really have need of instances of `Corpus<ObjectHandler<E>>`, where `E` is the type of object being represented by the corpus.

The basic work of a corpus class is done by its `visitTrain(H)` and `visitTest(H)` methods, where `H` is the handler type. Calling these methods sends all the training or testing events to the specified handler.

The method `visitCorpus(H)` sends both training and testing events to the specified handler and `visitCorpus(H,H)` sends the training events to the first handler and the test events to the second.

In all the LingPipe built ins, these methods will be `visitTrain(ObjectHandler<E>)` and `visitTest(ObjectHandler<E>)` for some object `E`.

2.3.2 Demo: 20 Newsgroups Corpus

We will use the 20 newsgroups corpus as an example (see Section D.2). The corpus contains around 20,000 newsgroup posts to 20 different newsgroups.

The demo class `TwentyNewsgroupsCorpus` implements a corpus based on this data. The corpus is implemented so that it doesn't need to keep all the data in memory; it reads it in from a compressed tar file on each pass. This approach is highly scalable because only one item at a time need reside in memory.

Code Walkthrough

The 20 newsgroups corpus class is defined to extend `Corpus`.

```
public class TwentyNewsgroupsCorpus
     extends Corpus<ObjectHandler<Classified<CharSequence>>> {

     private final File mCorpusFileTgz;

     public TwentyNewsgroupsCorpus(File corpusFileTgz) {
         mCorpusFileTgz = corpusFileTgz;
     }
```

The handler is an object handler handling objects of type `Classified<CharSequence>`, which represent character sequences with first-best category assignments from a classification. In the 20 newsgrops corpus, the categories are the newsgroups from which the posts originated and the objects being classified, of type `CharSequence`, are the texts of the posts themselves.

The constructor takes the corpus file, which is presumed to be directory structures that have been tarred and then compressed with GZIP (see the sections in the I/O chapter of the companion volume *Text Processing in Java* for more information on tar and GZIP). The corpus file is saved for later use.

The `visitTrain()` method sends all the classified character sequences in the training data to the specified hander.

```
@Override
public void visitTrain(ObjectHandler<Classified<CharSequence>>
                        handler)
     throws IOException {

     visitFile("20news-bydate-train",handler);
}
```

The `visitTest()` method is defined similarly.

The `visitFile()` method which is doing the work, is defined as follows.

```
private void visitFile(String trainOrTest,
           ObjectHandler<Classified<CharSequence>> handler)
     throws IOException {

     InputStream in = new FileInputStream(mCorpusFileTgz);
     GZIPInputStream gzipIn = new GZIPInputStream(in);
     TarInputStream tarIn = new TarInputStream(gzipIn);
```

It takes the string to match against the directories in the tarred directory data, and the handler. It starts by creating an input stream for gzipped tar data.

Next, it walks through the directories and files in he tarred data.

```
while (true) {
     TarEntry entry = tarIn.getNextEntry();
     if (entry == null) break;
     if (entry.isDirectory()) continue;
```

```
        String name = entry.getName();
        int n = name.lastIndexOf('/');
        int m = name.lastIndexOf('/',n-1);
        String trainTest = name.substring(0,m);
        if (!trainOrTest.equals(trainTest)) continue;
        String newsgroup = name.substring(m+1,n);
        byte[] bs = Streams.toByteArray(tarIn);
        CharSequence text = new String(bs,"ASCII");
        Classification c = new Classification(newsgroup);
        Classified<CharSequence> classified
            = new Classified<CharSequence>(text,c);
        handler.handle(classified);
}
tarIn.close();
```

We loop, getting the next tar entry, breaking out of the loop if the entry is null, indicating we have processed every entry. If the entry is a directory, we skip it. Otherwise, we get its name, then parse out the directory structure using the indexes of the directory structure slash (/) separators, pulling out the name of the directory indicating whether we have training or test data. For example, a file tar entry with name 20news-bydate-test/alt.atheism/53265 uses the top-level directory to indicate the training-testing split, here test, and the subdirectory to indicate the category, here alt.atheism. We are not keeping track of the article number, though it would make sense to set things up to keep track of it for a real application.

If the top-level train-test directory doesn't match, we continue. If it does match, we pull out the newsgroup name, then read the actual bytes of the entry using LingPipe's Streams.toByteArray() utility method. We create a string using the ASCII-encoding because the 20 newsgroups corpus is in ASCII. We then create a classification with category corresponding to the name of the newsgroup, and use it to create the classified cahracter sequence object. Finally, we give the classified character sequence to the object handler, which is defined to handle objects of type Classified<CharSequence>.

When we finish the loop, we close the tar input stream (which closes the other streams) and return.

There is a main() method which is called by the demo, converting a single command-line argument into a file tngFileTgz. The code starts by defining a final set of categories, then an anonymous inner class to define a handler.

```
final Set<String> catSet = new TreeSet<String>();
ObjectHandler<Classified<CharSequence>> handler
     = new ObjectHandler<Classified<CharSequence>>() {
     public void handle(Classified<CharSequence> c) {
         catSet.add(c.getClassification().bestCategory());
     }
};
```

The handle method extracts the best category from the classification and adds it to the category set. The category set needed to be final so that it could be referenced from within the anonymous inner class's handle method.

Once the handler is defined, we create a corpus from the gzipped tar file, then send the train and testing events to the handler.

```
Corpus<ObjectHandler<Classified<CharSequence>>> corpus
     = new TwentyNewsgroupsCorpus(tngFileTgz);
corpus.visitTrain(handler);
corpus.visitTest(handler);
```

The rest of the code just prints out the categories.

Running the Demo

The Ant target `twenty-newsgroups` invokes the `TwentyNewsgroupsCorpus` command, sending the value of property `tng.tgz` as the command-line argument.

```
>                    ant -Dtng.tgz=c:/lpb/data/20news-bydate.tar.gz
twenty-newsgroups
```

```
Cats= alt.atheism    comp.graphics    comp.os.ms-windows.misc
...
talk.politics.mideast    talk.politics.misc    talk.religion.misc
```

We have left out some in the middle, but the result is indeed the twenty expected categories, which correspond to the newsgroups.

2.3.3 The `ListCorpus` Class

LingPipe provides a dynamic, in-memory corpus implementation in the class `ListCorpus`, in package `com.aliasi.corpus`. The class has a generic parameter, `ListCorpus<E>`, where `E` is the type of objects handled. It is defined to extend `Corpus<ObjectHandler<E>>`, so the handler will be an object handler for type `E` objects.

There is a zero-argument constructor `ListCorpus()` that constructs an initially empty corpus. The two methods `addTrain(E)` and `addTest(E)` allow training or test items to be added to the corpus.

The methods `trainCases()` and `testCases()` return lists of the training and test items in the corpus. These lists are unmodifiable views of the underlying lists. Although they cannot be changed, they will track changes to the lists underlying the corpus.

There is an additional method `permuteCorpus(Random)`, that uses the specified random number generator to permute the order of the test and training items.

Serialization

A list-based corpus may be serialized if the underlying objects it stores may be serialized. Upon being reconstituted, the result will be a `ListCorpus` that may be cast to the appropriate generic type.

Thread Safety

A list corpus must be read-write synchronized, where the add and permute methods are writers, and the visit methods are readers.

2.3.4 The `DiskCorpus` Class

The `DiskCorpus` class provides an implementation of the `Corpus` interface based on one directory of files containing training cases and one containing test cases. A parser is used to extract data from the files in the training and test directories.

Like our example in the previous section, a disk corpus reads its data in dynamically rather than loading it into memory. Thus, if the data changes on disk, the next run through the corpus may produce different data.

The directories are expored recursively, and files with GZIP or Zip compression indicated by suffix `.gz` or `.zip` are automatically unpacked. For zip files, each entry is visited.

At the lowest level, every ordinary file, either in the directory structure or in a Zip archive is visited and processed.

Constructing a Disk Corpus

An instance of `DiskCorpus<H>`, where `H` extends or implements the marker interface `Handler`, is constructed using `DiskCoropus(Parser<H>,File,File)`, with the two files specifying the training and test directories respectively.

Once a disk corpus is constructed, the methods `setSystemId(String)` and `setCharEncoding(String)` may be used to configure it. These settings will be passed to the input sources constructed and sent to the parser.

Thread Safety

As long as there are no sets and the file system doesn't change, it'll be safe to use a single insance of a disk corpus from multiple threads.

2.4 Cross Validation

Cross validation is a process for evaluating a learning system in such a way as to extract as many evaluation points as possible from a fixed collection of data.

Cross validation works by dividing a corpus up into a set of non-overlapping slices, called folds. Then, each fold is used for evaluation in turn, using the other folds as training data. For instance, if we have 1000 data items, we may divide them into 3 folds of 333, 333, and 334 items respectively. We then evaluate fold 1 using fold 2 and 3 for training, evaluate on fold 2 using folds 1 and 3 for training, and evaluate on fold 3 using folds 1 and 2 for training. This way, every item in the corpus is used as a test case.

Leave-One-Out Evaluation

At the limit where the number of folds equals the number of data items, we have what is called leave-one-out (LOO) evaluation. For instance, with 1000 data items, we have 1000 folds. This uses as much data as possible in the training set. This is usually too slow to implement directly by retraining learning systems for each fold of leave-one-out evaluation.

Bias in Cross Validation

In some sitautions, cross-validation may be very biased. For instance, suppose we have 100 items, 50 of which are “positive” and 50 of which are “negative” and we consider leave-one-out evaluation. Even though our full training data is balanced, with equal numbers of positive and negative items, each fold of leave-one-out evaluation has 50 examples of the other category, and 49 examples of the category being evaluated. For learners that use this ratio in the training data to help with predictions, such as naive Bayes classifiers, there will be a noticeable bias against the category being evaluated in a leave-one-out evaluation.

Non-Stationarity and Cross Validation

Another problem plaguing cross-validating (and other evaluations) is non-stationary distributions of text. If text were stationary at the sentence level, then the content of a sentence (conditioned on the category, say, in a classifier) would be independent of the content of the other sentences in a single text item. In reality, natural language text is highly non-stationary. This is easiest to see with names, where if a name is mentioned in an article it is much more likely to be mentioned again in the same article than be mentioned in a different article.

The evidence for this kind of non-stationarity is results that vary among folds more than you would expect them to by chance alone. A system that is 80% accurate and classifies 100 documents, will find 95% of its runs in the range from 72 to 88, with lots of variation. This can be computed with the binomial confidence interval. If more than this amount of variance is seen, there are problems with non-stationarity.

In some situations, we can avoid intra-document word and phrase correlations in our evaluations by choosing only small bits out of a single document.

Another issue with stationarity is the temporal nature of language. If we have a year of newswire, it gives a very different perspective to break the documents up at random or to put them in temporal buckets. Not only does what gets talked about differ by month (different holidays, weather, etc.), once something gets introduced into a document collection, it can stick around as a “hot” topic for a while, then disappear. This can lead to overly optimistic performance estimates if part of such a topic is used to train and the other part for evaluation.

If each fold is a month, the fairest test is to train on the last month given the earlier months. Training by cross-validating by month, or even worse, cross-validating by sentence, leads to a situation where you use text in the future to train a system to operate on

In real applications, if the system is not trained continuously, it will gradually fall out of step with the kind of text it's about.

2.4.1 Permuting the Corpus

Cross-validation only makes sense theoretically when the items are all drawn at random from some large population (or potential population) of items. Very rarely does natural language data satisfy this requirement. Instead, we gather data by document and by date. If you have the choice, by all means take a truly random sample of data.

It is sometimes reasonable to permute the items in a cross-validating corpus. Often, this is because examples of the same item may be stored together and should be broken up for evaluation.

Permuting may make some problems too easy. For instance, if the items are chunkings of sentences for named entities, and more than one sentence is drawn from a given document, it makes sense to keep these together (or at least not intentionally split them apart). Otherwise, we are much more likely to have been trained no the rare words appearing in sentences.

2.4.2 The `CrossValidatingObjectCorpus` Class

LingPipe provides a corpus implementation tailored to implementing cross validation. The class `CrossValidatingObjectCorpus<E>` has a generic parameter `E` defining the objects that are handled in the corpus. The class extends `Corpus<ObjectHandler<E>>`, so the handlers for a cross-validating object corpus will have a `handle(E)` method.

Construction

The cross-validating corpus has a constructor `CrossValidatingObjectCorpus(int)`, where the number of folds must be specified. These may be changed later using the `setNumFolds(int)` method.

Populating the Corpus

Items are added to the corpus using the method `handle(E)`. These are all stored in lists in memory.

For convenience, the cross-validating corpus class is defined to implement `ObjectHandler<E>` itself, which only requires the `handle(E)` method. Thus an entire cross-validating corpus may be sent to a parser for population.

The `size()` method indicates the total number of objects in the corpus.

Using the Corpus

The particular fold to use to divide up the data is set using the method `setFold(int)`. The current fold is available through the method `fold()`. The folds are numbered starting at 0 and ending at the number of folds minus one. The number of folds may also be reset at any time using the method `setFold(int)`. The corpus may be permuted using a specified pseudorandom number generator with the method `permuteCorpus(Random)`.

Serialization

A cross-validating corpus will be serializable if the objects it contains are serializable.

Thread Safety and Concurrent Cross-Validation

Cross-validating corpus instances must be read-write synchronized (see the section in the companion volume *Text Processing in Java* for information on read-write synchronization). The read operations are in the handle, permute and set methods.

The obstacle to using a cross-validating corpus across threads is that the fold is set on the class itself. To get around this limitation, the method `itemView()` returns a "view" of the corpus that allows the fold number or number of folds to be changed, but not the items. Thus to cross-validate across folds, just create and populate and permute a corpus. Then use `itemView()` to produce as many virtual copies as you need, setting the folds on each one.

Note that this does not help with the problem of evaluators being used to aggregate scores across folds not being thread safe. Given that it is usually training that is the bottleneck in cross validation, explicit synchronization of the evaluators should not cause too much of a synchronization bottleneck.

Examples of Cross Validation

Because cross-validation is so critical, there are examples of its use for most of the systems that can be evaluated. The following is a list of examples.

Evaluator	*Model*	*Reference*
`JointClassifier`	`TradNaiveBayes`	Section 10.11

Chapter 3

Tokenization

Many natural-language processing algorithms operate at the word level, such as most part-of-speech taggers and named-entity chunkers, some classifiers, some parameterizations of spelling correction, etc.

A token is a generalized kind of word that is derived from segmenting an input character sequence and potentially transforming the segments. For instance, a search engine query like `[London restaurants]` might be converted into a boolean search for the (case normalized) token *london* and the (plurality normalized) token *restaurant*.

3.1 Tokenizers and Tokenizer Factories

LingPipe provides a package `com.aliasi.tokenizer` for handling tokenization.

3.1.1 The `TokenizerFactory` Interface

The `TokenizerFactory` factory interface defines a single method, `tokenizer(char[],int,int)`, which takes a slice of a character array as an argument and returns an instance of `Tokenizer`.

LingPipe's tokenizer factory implementations come in two flavors Basic tokenizer factories are constructed from simple parameters. For the basic tokenizers with no parameters, a singleton instance is supplied as a static constant in the class. For consistency, these singleton constants all have the variable name `INSTANCE`.

Tokenizer filters are constructed from other tokenizer factories and modify their outputs in some way, such as by case normalization, stemming, or stop-word filtering.

In order to bundle a tokenizer factory with a model, it must be serializable. All of LingPipe's tokenizer factories are serializable, including ones made up by composing a sequence of filters.

3.1.2 The `Tokenizer` Base Class

All tokenizers extend the abstract base class `Tokenizer`. Tokenizers provide a stream of tokens. An instance of `Tokenizer` represents the state of tokenizing a particular string.

Constructing a Tokenizer

There is no state represented in the `Tokenizer` abstract class, so there is a single no-argument constructor `Tokenizer()`.

Because tokenizers are usually created through the `TokenizerFactory` interface, most classes extending `Tokenizer` are not delcared to be public. Instead, only the factory is visible, and the documentation for a tokenizer's behavior will be in the factory's class documentation.

Streaming Tokens

The only method that is required to be implemented is `nextToken()`, which returns the next token in the token stream as a string, or `null` if there are no more tokens. There is no reference in a tokenizer itself to the underlying sequence of characters.

Token Positions

We often wish to maintain the position of a token in the underlying text. Given that tokens may be modified or even dropped altogether, the position of a token is not necessarily going to be recoverable from the sequence of tokens and whitespaces. So the `Tokenizer` class supplies methods `lastTokenStartPosition()` and `lastTokenEndPosition()`, which return the index of the first character and of one past the last character. If no tokens have yet been returned, these methods both return -1. These positions are relative to the slice being tokenized, not to the underlying character array.

The token position methods are implemented in the `Tokenizer` base class to throw an `UnsupportedOperationException`. All of LingPipe's built-in tokenizer factories produce tokenizers that properly handle token positions.

User-defined ubclasses that need token positions should override these methods. Tokenizer filters should almost always just pass the positions of the tokens being modified, so that the derived token may be linked back to its source in the original text.

Iteration

The method `iterator()` returns an iterator over strings representing tokens. In the `Tokenzer` base class, the iterator is defined by delegation to the `nextToken()` method. Thus subclasses do not usually need to redefine this method.

This `iterator()` method allows the `Tokenizer` class to implement the `Iterable<String>` interface. Thus the tokens can be read from a tokenizer with a for loop. Given a tokenizer factory `tokFac` and the character slice for input, the usual idiom is

```
Tokenizer tokenizer = tokFac.tokenizer(cs,start,length);
for (String token : tokenizer) {
     // do something
}
```

Serializability and Thread Safety

Because they involve dynamic state, tokenizers are almost never serializable and almost never thread safe.

Streaming Whitespaces

Over time, LingPipe has moved from the use of whitespace returned from tokenizers to token start and end positions. Unless otherwise noted, tokenizers need not concern themselves with whitespace. LingPipe's built-in tokenizers almost all define the whitespace to be the string between the last token and the next token, or the empty string if that is not well defined.

The method `nextWhitespace()` returns the next whitespace from the tokenizer. "White space" is the general term for the material between tokens, because in most cases, all non-whitespace is part of some token. LingPipe's `Tokenizer` class generalizes the notion of whitespace to arbitrary strings.

Each token is preceded by a whitespace and the sequence ends with a whitespace. That is, the sequence goes whitespace, token, whitespace, token, ..., whitespace, token, whitespace. So the number of whitespaces is one greater than the number of tokens, and the minimum output of a tokenizer is a single whitespace.

If the `nextWhitespace()` method is not implemented by a subclass, the implementation inherited from the `Tokenizer` base class simply returns a string consisting of a single space character, U+0020, SPACE.

If the `nextWhitespace()` method is implemented to return the text between tokens, tokens do not overlap, and the string for a token is not modified in any way, then concatenating the sequence of whitespaces and tokens will produce the underlying characters that were tokenized.

3.1.3 Token Display Demo

We provide a demo program `DisplayTokens`, which runs a tokenizer over the command-line argument. The `main()` method of the command calls a utility method on a string variable `text` supplied on the command line, using a built-in tokenizer

```
TokenizerFactory tokFact
     = IndoEuropeanTokenizerFactory.INSTANCE;
displayTextPositions(text);
displayTokens(text,tokFact);
```

The IndoEuropeanTokenizerFactory class is in com.aliasi.tokenizer, and provides a reusable instance through the static constant INSTANCE.

The displayTextPositions() method just prints out the string on a single line followed by lines providing indexes into the string. This method won't display properly if there are newlines in the string itself.

The work is actually done in the subroutine, the body of which is

```
char[] cs = Strings.toCharArray(in);
Tokenizer tokenizer = tokFact.tokenizer(cs,0,cs.length);

for (String token : tokenizer) {
     int start = tokenizer.lastTokenStartPosition();
     int end = tokenizer.lastTokenEndPosition();
```

We first convert the character sequence in to a character array using the utility method toCharArray(CharSequence) from the class Strings in the package com.aliasi.util. then, we create the tokenizer from the tokenizer factory. Next, we just iterate over the tokens, extract their start and end positions, and print the results.

We provide a corresponding Ant target display-tokens, which is given a single command-line argument consisting of the value of the property text.

```
>     ant -Dtext="The note says, 'Mr. Sutton-Smith owes $15.20.'"
display-tokens

The note says, 'Mr. Sutton-Smith owes $15.20.'
0123456789012345678901234567890123456789012345
0         1         2         3         4

START   END  TOKEN          START   END TOKEN
    0     3  |The|             26    27  |-|
    4     8  |note|            27    32  |Smith|
    9    13  |says|            33    37  |owes|
   13    14  |,|               38    39  |$|
   15    16  |'|               39    44  |15.20|
   16    18  |Mr|              44    45  |.|
   18    19  |.|               45    46  |'|
   20    26  |Sutton|
```

In writing the string, we put the ones-place indexes below it, then the tens-place indexes, and so on. Because we count from zero, the very first *T* has index 0, whereas the *M* in *Mr* has index 16 and the final apostrophe index 45. The string is 46 characters long; the last index is always one less than the length.

We then show the tokens along with their start and end positions. As always, the start is the index of the first character and the end is one past the index of the last character. Thus the name *Smith* starts at character 27 and ends on character 31. This also means that the end position of one token may be the start position of the next when there is no space between them. For example, *says* has an end position of 13 (which is exclusive) and the following comma a start position of 13 (which is inclusive).

3.2 LingPipe's Base Tokenizer Factories

LingPipe provides several base tokenizer factories. These may be combined with filters, which we describe in the next section, to create compound tokenizers.

3.2.1 The `IndoEuropeanTokenizerFactory` Class

LingPipe's `IndoEuropeanTokenizer` is a fairly fine-grained tokenizer in the sense that it splits most things apart. The notable exception in this instance is the number *15.20*, which is kept as a whole token. Basically, it consumes as large a token as possible according to the following classes.

Kinds of Tokens

An alphanumeric token is a sequence of one or more digits or letters as defined by the utility methods `isDigit()` and `isCharacter()` methods in Java's `Character` class. Thus *aaa*, *A1*, and *1234* are all considered single tokens.

A token may consist of a combination of digits with any number of token-internal periods or commas. Thus *1.4.2* is considered a token, as is *12,493.27*, but *a2.1* is three tokens, *a2*, *.* (a period), and *1*.

A token may be an arbitrarily long sequence of hyphens (-) or an arbitrarily long sequence of equal signs (=). The former are often used as en-dashes and em-dashes in ASCII-formatted text, and longer sequences are often used for document structure.

Double grave accents (“) and double apostrophes (”) are treated as single tokens, as they are often used to indicate quotations.

All other non-space characters, such as question marks or ampersands, are considered tokens themselves.

Construction

The static constant `INSTANCE` refers to an instance of `IndoEuropeanTokenizerFactory` that may be reused. The no-argument constructor may also be used.

Thread Safety and Serializability

The `IndoEuropeanTokenizerFactory` class is both thread safe and serializable. The deserialized object will be reference identical to the factory picked out by the `INSTANCE` constant.

3.2.2 The `CharacterTokenizerFactory` Class

The `CharacterTokenizerFactory` treats each non-whitespace `char` value in the input as a token. The definition of whitespace is from Java's `Character.isWhitespace()` method. The whitespaces returned will consist of all of the whitespace found between the non-whitespace characters.

For instance, for the string *a dog*, there are four tokens, *a*, *d*, *o*, and *g*. The whitespaces are all length zero other than for the single space between the *a* and *d* characters.

This tokenizer factory is particularly useful for Chinese, where there is no whitespace separating words, but words are typically only one or two characters (with a long tail of longer words).

This class produces a token for each `char` value. This means that a surrogate pair consisting of a high surrogate UTF-16 value and low surrogate UTF-16 value, which represent a single Unicode code point, will produce two tokens. Specifically, unicode code points outside of the basic multilingual plane (BMP), that is with values at or above U+10000, will produce two tokens (see the section on UTF-16 in the companion volume, *Text Processing in Java*, for more information).

The class is thread safe and serializable. There is no constructor, only a static constant `INSTANCE`. The instance is also the result of deserialization.

3.2.3 The `RegExTokenizerFactory` Class

For parsing programming languages, the lexical analysis stage typically breaks a program down into a sequence of tokens. Traditional C packages like Lex and Java packages like JavaCC specify regular expressions for tokens (and context-free grammars with side effects for parse trees). Regular expression-based tokenization is also popular for natural language processing.

LingPipe's `RegExTokenizerFactory` implements tokenizers where the tokens are defined by running regular expression matchers in find mode (see the section in the companion volume *Text Processing in Java* for more on the find operation). An instance may be constructed from a `Pattern`, or from a `String` representing a regular expression with optional integer flags (see the section on modes in patterns in the regular expression chapter of the companion volume, *Text Processing in Java*, for information on how to use the flags and their numerical values).

The demo class `RegexTokens` implements a simple regular expression tokenizer factory based on three command line arguments, one for the regular expression, one for the flags, and one for the text being tokenized.

```
int flagInt = Integer.valueOf(flags);
TokenizerFactory tokFact
     = new RegExTokenizerFactory(regex,flagInt);

DisplayTokens.displayTokens(text,tokFact);
```

The Ant target `regex-tokens` runs the command, setting three command line arguments for three properties, `regex` for the string representation of the regular expression, `flags` for the integer flags, and `text` for the text to be tokenized.

```
> ant -Dregex="\p{L}+" -Dflags=32 -Dtext="John likes 123-Jello."
regex-tokens

regex=|\p{L}+|
flags=32 (binary 100000)
```

```
John likes 123-Jello.
012345678901234567890
0         1         2

START   END TOKEN
    0     4  |John|
    5    10  |likes|
   15    20  |Jello|
```

First we print the regex and the flags (in both decimal and binary forms), then the string with indexes, and finally the tokens. Because we used the pattern `\p{L}+`, our tokens consist of maximally long sequences of letter characters. Other characters such as spaces, numbers, punctuation, etc., are simply skipped so that they become part of the whitespace.

Instances of `RegExTokenizerFactory` are serializable. They are also thread safe.

3.2.4 The `NGramTokenizerFactory` Class

An `NGramTokenizerFactory` produces tokens from an input string consisting of all of the substrings of the input within specified size bounds. Substrings of an input string of length n are typically called n-grams (and sometimes q-grams).

Thus such a factory is constructed using a minimum and maximum sequence size, with `NGramTokenizerFactory(int,int)`. The sequences produced include the whitespaces between what we typically think of as tokens. Tokenizers such as these are especially useful for extracting features for classifiers.

We provide an example implementation in the demo class `NGramTokens`. The only thing novel is the construction of the tokenizer factory itself,

```
NGramTokenizerFactory tokFact
    = new NGramTokenizerFactory(minNGramInt,maxNGramInt);
DisplayTokens.displayTokens(text,tokFact);
```

It takes three command-line arguments, the minimum and maximum sequence lengths and the text to analyze. These are supplied to Ant target `ngram-tokens` as properties `minNGram`, `maxNGram`, and `text`.

```
> ant -DminNGram=1 -DmaxNGram=3 -Dtext="I ran." ngram-tokens
minNGram=1
maxNGram=3

I ran.
012345

START   END TOKEN              START   END TOKEN
    0     1  |I|                   2     4  |ra|
    1     2  | |                   3     5  |an|
    2      3   |r|                      4      6   |n.|
    3      4   |a|                      0      3   |I r|
    4      5   |n|                      1      4   | ra|
    5      6   |.|                      2      5   |ran|
    0      2   |I |                     3      6   |an.|
    1      3   | r|
```

You can see from the output that the tokenizer first outputs the 1-grams , then the 2-grams, then the 3-grams. These are typically referred to as “unigrams,” “bigrams,” and “trigrams.”[1] Note that the spaces and punctuation are treated like any other characters. For instance, the unigram token spanning positions 1 to 2 is the string consisting of a single space character and the one spanning from 5 to 6 is the string consisting of a single period character. The first bigram token, spanning from positions 0 to 2, is the two-character string consisting of an uppercase I followed by a space. By the time we get to trigrams, we begin to get cross-word tokens, like the trigram token spanning from position 0 to 3, consistinf of an uppercase I, space, and lowercase r.

In terms of the information conveyed, the spaces before and after words help define word boundaries. Without the bigram token spanning 1 to 3 or the trigram spanning 1 to 4, we wouldn’t know that the lowercase r started a word. The cross-word ngram tokens help define the transitions between words. With long enough ngrams, we can even get three-word effects,

3.2.5 The `LineTokenizerFactory` Class

A `LineTokenizerFactory` treats each line of input as a single token. It is just a convenience class extending `RegExTokenizerFactory` supplying the regex `.+`.

Line termination is thus defined as for regular expression patterns (see the section on line-ending in the regular expression chapter of the companion volume, *Text Processing in Java*, for the full set of line-end sequences recognized). Final empty lines are not included in the sequence of tokens.

There is a static constant `INSTANCE` which may be used. This is also the value of deserialization. Line tokenizer factories are thread safe.

3.3 LingPipe’s Filtered Tokenizers

Following the same filter-based pattern as we saw for Java’s I/O, LingPipe provides a number of classes whose job is to filter the tokens from an embedded tokenizer. In all cases, there will be a base tokenizer producing a tokenizer which is then modified by a filter to produce a different tokenizer.

The reason the tokenizer is modified rather than, say, the characters being tokenized, is that we usually want to be able to link the tokens back to the original

[1]Curiously, we get a transition in common usage from Latin to Greek at 4-grams (tetragrams), and 5-grams (pentagrams), 6-grams (hexagrams) etc. Some authors stick to all Greek prefixes, preferring the terms “monogram” and “digram” to “unigram” and “bigram.”

text. If we allow an arbitrary transformation of the text, that's no longer possible. Thus if the text itself needs to be modified, that should be done externally to tokenization.

3.3.1 The `ModifiedTokenizerFactory` Abstract Class

The abstract class `ModifiedTokenizerFactory` in the package `com.aliasi.tokenizer` provides a basis for filtered tokenizers implemented in the usual way. Like many LingPipe filter classes, a modified tokenizer factory is immutable. Thus the base tokenizer factory it contains must be provided at construction time. This is done through the single constructor `ModifiedTokenizerFactory(TokenizerFactory)`. The specified tokenizer factory will be stored, and is made available through the method `baseTokenizerFactory()` (it is common to see protected member variables here, but we prefer methods).

It then implements the `tokenizer(char[],int,int)` method in the tokenizer factory interface using the base tokenizer factory to produce a tokenizer which is then passed through the method `modify(Tokenizer)`, which returns a `Tokenizer` that may then be returned. This is perhaps easier to see in code,

```
protected abstract Tokenizer modify(Tokenizer tokenizer);

public Tokenizer tokenizer(char[] cs, int start, int length) {
     TokenizerFactory tokFact = baseTokenizerFactory();
     Tokenizer tok = tokFact.tokenizer(cs,start,length);
     return modify(tok);
}
```

Serialization

A modified tokenizer factory will be serializable if its base tokenizer factory is serializable. But unless the subclass defines a public no-argument constructor, trying to deserialize it will throw an exception. If a subclass fixes a base tokenizer factory, there may be a public no-argument constructor. A subclass intended to be used as a general filter will not have a public no-argument constructor because it would lead to instances without base tokenizers defined.

As elsewhere, subclasses should take control of their serialization. We recommend using the serialization proxy, and show an example in the next section.

Thread Safety

A modified tokenizer factory is thread safe if its `modify()` method is thread safe.

3.3.2 The `ModifyTokenTokenizerFactory` Abstract Class

In the previous section, we saw how the `ModifyTokenizerFactory` base class could be used to define tokenizer filters. That class had a modify method that consumed a tokenizer and returned a tokenizer. The class

ModifyTokenTokenizerFactory is a subclass of ModifiedTokenizerFactory that works at the token level. Almost all of LingPipe's tokenizer filters are instances of this subclass.

The method modifyToken(String) operates a token at a time, returning either a modified token or null to remove the token from the stream. The implementation in the base class just returns its input.

There is a similar method modifyWhitespace(String), though whitespaces may not be removed. If tokens are removed, whitespaces around them are accumulated. For instance, if we had w_1, t_1, w_2, t_2, w_3 as the stream of whitespaces and tokens, then removed token t_2, the stream would be $w_1, t_1, w_2 \cdot w_3$, with the last two whitespaces being concatenated and the token being removed.

The start and end points for modified tokens will be the same as for the underlying tokenizer. This allows classes that operate on modified tokens to link their results back to the string from which they arose.

This subclass behaves exactly the same way as its superclass with respect to serialization and thread safety. For thread safety, both the modify methods must both be thread safe.

Example: Reversing Tokens

As an example, we provide a demo class ReverseTokenTokenizerFactory. This filter simply reverses the text of each token. Basically, the implementation involves a declaration, constructor and definition of the modify operation,

```
public class ReverseTokenTokenizerFactory
     extends ModifyTokenTokenizerFactory {

     public ReverseTokenTokenizerFactory(TokenizerFactory f) {
         super(f);
     }

     public String modifyToken(String tok) {
         return new StringBuilder(tok).reverse().toString();
     }
```

In addition, the class implements a serialization proxy (see the section on serialization proxies in the I/O chapter of the companion volume, *Text Processing in Java*).

The main() method for the demo reads a command line argument into the variable text, then constructs a token-reversing tokenizer factory based on an Indo-European tokenizer factory, then displays the output,

```
TokenizerFactory baseTokFact
     = IndoEuropeanTokenizerFactory.INSTANCE;
ReverseTokenTokenizerFactory tokFact
     = new ReverseTokenTokenizerFactory(baseTokFact);
DisplayTokens.displayTextPositions(text);
DisplayTokens.displayTokens(text,tokFact);
```

It also serializes and deserializes the factory and displays the output for that, but we've not shown the code here.

The Ant target `reverse-tokens` calls the class's `main()` method, supplying the value of property `text` as the command-line argument.

```
> ant reverse-tokens
The note says, 'Mr. Sutton-Smith owes $15.20.'
0123456789012345678901234567890123456789012345
0         1         2         3         4

START   END TOKEN
    0     3  |ehT|
    4     8  |eton|
    9    13  |syas|
...
   39    44  |02.51|
   44    45  |.|
   45    46  |'|
```

The results are the same positions as for the Indo-European tokenizer, only with the content of each token reversed. We have not shown the serialized and then deserialized output; it's the same.

3.3.3 The `LowerCaseTokenizerFactory` Class

The `LowerCaseTokenizerFactory` class modifies tokens by converting them to lower case. This could've just as easily been upper case – it just needs to be normalized.

Case normalization is often applied in contexts where the case of words doesn't matter, or provides more noise than information. For instance, search engines typically store case-normalized tokens in their reverse index. And classifiers often normalize words for case (and other properties).

Instances are constructed by wrapping a base tokenizer factory. Optionally, an instance of `Locale` to define the specific lowercasing operations. If no locale is specified, `Locale.ENGLISH` is used. (This ensures consistent behavior across platforms, unlike using the platform's default locale.)

Whitespace and start and end positions of tokens are defined as by the base classifier.

The class is used like other filters. For instance, to create a tokenizer factory that returns lowercase tokens produced the Indo-European tokenizer using German lowercasing, just use

```
TokenizerFactory f = IndoEuropeanTokenizerFactory.INSTANCE;
Locale loc = Locale.GERMAN;
TokenizerFactory fact = new LowerCaseTokenizerFactory(f,loc);
```

Instances will be serializable if the base tokenizer factory is serializable. Serialization stores the locale using its built-in serialization, as well as the base tokenizer factory. Instances will be thread safe if the base tokenizer factory is thread safe.

3.3.4 The `StopTokenizerFactory` Class

The `StopTokenizerFactory` provides a tokenizer factory filter that removes tokens that appear in a specified set of tokens. The set of tokens removed is typically called a *stop list*. The stop list is case sensitive for this class, but stoplisting is often applied after case normalization.

It is common in token-based classifiers and search engines to further normalize input by removing very common words. For instance, the words *the* or *be* in English documents typically convey very little information themselves (they can convey more information in context). Sometimes this is done for efficiency reasons, in order to store fewer tokens in an index or in statistical model parameter vectors. It may also help with accuracy, especially in situations with limited amounts of training data. In small data sets, the danger is that these stop words will become associated with some categories more than others at random and thus randomly bias the resulting system.

Stop lists themselves are represented as sets of tokens. Stoplisting tokenizers are constructed by wrapping a base tokenizer and providing a stop list. For instance, to remove the words *the* and *be*, we could use

```
Set<String> stopSet = CollectionUtils.asSet("the","be");
TokenizerFactory f1 = IndoEuropeanTokenizerFactory.INSTANCE;
TokenizerFactory f2 = new LowerCaseTokenizerFactory(f1);
TokenizerFactory f3 = new StopTokenizerFactory(f2,stopSet);
```

The first line uses the static utility method `asSet()` in the LingPipe utility class `CollectionUtils`. This method is defined with signature

```
static <E> HashSet<E> asSet(E... es) {
```

so that it takes a variable-length number of `E` objects and returns a hash set of that type. Because Java performs type inference on method types (like the <E> generic in `asSet()`), we don't need to specify a type on the method call.

We then wrap an Indo-European tokenizer factory in a lowercasing and then a stoplisting filter. Operationally, what will happen is that the Indo-European tokenizer will tokenize, the resulting tokens will be converted to lower case, and then stop words will be removed. The calls start out in reverse, though, with `f3` called to tokenize, which then calls `f2` and filters the resulting tokens, where `f2` itself calls `f1` and then filters its results.

What's a Good Stop List?

The utility of any given stop list will depend heavily on the application. It often makes sense in an application to run several different stop lists to see which one works best.

Usually words on the stop list will be of relatively high frequency. One strategy is to print out the top few hundred or thousand words in a corpus sorted by frequency and inspect them by hand to see which ones are "uninformative" or potentially misleading.

Built-in For English

LingPipe provides a built-in stop list for English. The `EnglishStopTokenizerFactory` class extends the `StopTokenizerFactory` class, plugging in the following stoplist for English:

> *a, be, had, it, only, she, was, about, because, has, its, of, some, we, after, been, have, last, on*

Note that the entries are all lowercase. It thus makes sense to run this filter to a tokenizer factory that produces only lowercase tokens.

This is a fairly large stop list. It's mostly function words, but also includes some content words like *mr*, *corp*, and *last*. Also note that some words are ambiguous, which is problematic for stoplisting. For instance, the word *can* acts as both a modal auxiliary verb (presently possible) and a noun (meaning among other things, a kind of container).

3.3.5 The `RegExFilteredTokenizerFactory` Class

The `RegExFilteredTokenizerFactory` class provides a general filter that removes all tokens that do not match a specified regular expression. For a token to be retained, it must match the specified regular expression (in the sense of `match()` from `Matcher`).

An instance of `RegExFilteredTokenizerFactory` may be constructed from a base tokenizer factory and a `Pattern` (from `java.util.regex`). For instance, we could retain only tokens that started with capital letters (as defined by Unicode) using the following code for a variable `f` of type `TokenizerFactory`,

```
Pattern p = Pattern.compile("\\p{Lu}.*");
TokenizerFactory f2 = new RegExFilteredTokenizerFactory(f,p);
```

We could just as easily retain only tokens that didn't start with an uppercase letter by using the regex `[^\p{Lu}].*` instead of the one supplied.

Although stop lists could be implemented this way, it would not be efficient. For one thing, regex matching is not very efficient for large disjunctions; set lookup is much faster. Also, at the present time, the class creates a new `Matcher` for each token, generating a large number of short-lived objects for garbage collection.[2]

Like the other filter tokenizers, a regex filtered tokenizer is thread safe and serializable if its base tokenizer factory is.

3.3.6 The `TokenLengthTokenizerFactory` Interface

The class `TokenLengthTokenizerFactory` implements a tokenizer filter that removes tokens that do not fall in a prespecified length range. In real text, tokens can get very long. For instance, mistakes in document formatting can produce

[2]A more efficient implementation would reuse a matcher for each token in a given tokenization. The only problem is that it can't be done naturally and thread-safely by extending `ModifyTokenTokenizerFactory`. It would be possible to do this by extending `ModifiedTokenizerFactory`.

a document where all the words are concatenated without space. If you look at large amounts of text (in the gigabytes range), especially if you're looking at unedited sources like the web, you will find longer and longer tokens used on purpose. It's not unusual to see *no*, `noo`, and `nooooooo!`. Look at enough text, and someone will add 250 lowercase O characters after the *n*.

The length range is supplied in the constructor, which takes a tokenizer factory, minimum token length, and maximum token length. Like the other filters, a length filtering tokenizer will be serializable and thread safe if the base tokenizer factory is. And like some of the other filters, this could be implemented very easily, but not as efficiently, with regular expressions (in this case the regex `.{`*m*`,`*n*`}` would do the trick).

3.3.7 The `WhitespaceNormTokenizerFactory` Class

When interpreting HTML documents on the web, whitespace (outside of `<pre>` environments) comes in two flavors, no space and some space. Any number of spaces beyond will be treated the same way for rendering documents as a single space.

This is such a reasonable normalization scheme that we often normalize our input texts in exactly the same way. This is easy to do with a regex, replacing a string `text` with `text.replaceAll("\\s+"," ")`, which says to replace any non-empty sequence of spaces with a single space.

It's not always possible to normalize inputs. You may need to link results back to some original text that's not under your control. In these cases, it's possible to modify the whitespaces produced by the tokenizer. This will not have any effect on the underlying text or the spans of the tokens themselves. Unfortunately, very few of LingPipe's classes pay attention to the whitespace, and for some that do, it's the token bounds that matter, not the values returned by `nextWhitespace()`.

A `WhitespaceNormTokenizerFactory` is constructed from a base tokenizer factory. Like the other filter tokenizer factories, it will be serializable if the base tokenizer factory is serializable and will be thread safe if the base tokenizer factory is thread safe.

3.4 Morphology, Stemming, and Lemmatization

In linguistics, morphology is the study of how words are formed. Morphology crosses the study of several other phenomena, including syntax (parts of speech and how words combine), phonology (how words are pronounced), orthography (how a language is written, including how words are spelled), and semantics (what a word means).

LingPipe does not contain a general morphological analyzer, but it implements many parts of word syntax, semantics and orthography.

A morpheme is the minimal meaningful unit of language. A morpheme doesn't need to be a word itself. As we will see shortly, it can be a word, a suffix, a vowel-change rule, or a template instantiation. Often the term is extended to include syntactic markers, though these almost always have some semantic or semantic-like effect, such as marking gender.

We consider these different aspects of word forms in different sections. LingPipe has part-of-speech taggers to determine the syntactic form a word, syllabifiers to break a word down into phonemic or orthographic syllables, word-sense disambiguators which figure out which meaning of an ambiguous word is intended, and many forms of word clustering and comparison that can be based on frequency, context similarity and orthographic similarity. LingPipe also has spelling correctors to fix misspelled words, which is often handled in the same part of a processing pipeline as morphological analysis.

In this section, we provide a brief overview of natural language morphology, then consider the processes of stemming and lemmatization and how they relate to tokenization.

3.4.1 Inflectional Morphology

Inflectional morphology is concerned with how the basic paradigmatic variants of a word are realized. For instance, in English, the two nouns `computer` and `computers` differ in number, the first being singular, the second plural. These show up in different syntactic contexts — we say *one computer* and *two computers*, but not *one computers* and *two computer*.[3] This is an instance of agreement — the grammatical number of the noun must agree with the number of the determiner. In English, there are only two numbers, singular and plural, but other languages, like Icelandic and Sanskrit, have duals (for two things), and some languages go even further, defining numbers for three objects, or distinguishing many from a few.

In English, the verbs `code`, `codes`, `coded`, and `coding` are all inflectional variants of the same underlying word, differing in person, number, tense, and aspect. We get agreement between verbs and subject nouns in English for these features. For instance, a person might say *I code* in referring to themselves, but *she codes* when referring to their friend writing code. Other languages, such as French or Russian, also have agreement for gender, which can be natural (male person versus female person versus versus non-person) or grammatical (as in French). Natural gender is based on what is being talked about. That's the distinction we get in English between the relative pronouns *who*, which is used for people and other animate objects, and *what*, which is used for inanimate objects. For instance, we say *who did you see?* if we expect the answer to be a person, and *what did you see?* if we want a thing.

Most languages have fairly straightforward inflectional morphology, defining a few variants of the basic nouns and verbs. The actual morphological operations may be as simple as adding an affix, but even this is complicated by boundary effects. For instance, in English present participle verbs, we have *race/racing*, *eat/eating*, and *run/running*. In the first case, we delete the final *e* before adding the suffix *-ing*, in the second case, we just append the suffix, and in the third case, we insert an extra *n* before adding the suffix. The same thing happens with

[3]These ungrammatical forms like *one computers* will appear in text. You'll find just about everything in free text and even in fairly tightly edited text like newswire if you look at enough of it.

number for nouns, as with *boy/boys* versus *box/boxes*, in which the first case appends the suffix *s* whereas the second adds an *e* before the suffix (the result of which, *boxes*, is two syllables).

It is also common to see internal vowel changes, as in the English alternation *run/ran*. Languages like Arabic and Hebrew take this to the extreme with their templatic morphology systems in which a base form consists of a sequence of consonants and an inflected form mixes in vowels.

3.4.2 Derivational Morphology

Derivational morphology, in contrast to inflectional morphology, involves modifying a word to create a new word, typically with a different function. For instance, from the verb *run* we can derive the noun *runner* and from the adjective *quick* we can derive the adverb *quickly*. After we've derived a new form, we can inflect it, with singular form *runner* and plural form *runners*.

Inflectional morphology is almost always bounded to a finite set of possible inflected forms for each base word and category. Derivational morphology, on the other hand, is unbounded. We can take the noun *fortune*, derive the adjective *fortunate*, the adjective *unfortunate*, and then the adverb *unfortunately*.

Sometimes there's zero derivational morphology, meaning not that there's no derivational morphology, but that it has no surface effect. For instance, we can turn most nouns in English into verbs, especially in the business world, and most verbs into nouns (where they are called gerunds). For instance, I can say *the running of the race* where the verb *running* is used as a noun, or *Bill will leverage that deal*, where the noun *leverage* (itself an inflected form of *lever*) may be used as a verb. In American English, we're also losing the adjective/adverb distinction, with many speakers not even bothering to use an adverbial form of an adjective, saying things like *John runs fast*.

3.4.3 Compounding

Some languages, like German, allow two nouns to be combined into a single word, written with no spaces, as in combining *schnell* (fast) and *Zug* (train) into the compound *schnellzug*. This is a relatively short example involving two nouns.

Comnpounds may themselves be inflected. For instance, in English, we may combine *foot* and *ball* to produce the compound noun *football*, which may then be inflected in the plural to get *footballs*. They may also be modified derivationally, to get *footballer*, and then inflected, to get *footballers*.

3.4.4 Languages Vary in Amount of Morphology

Languages are not all the same. Some code up lots of information in their morphological systems (e.g., Russian), and some not much at all (e.g., Chinese). Some languages allow very long words consisting of lots of independent morphemes (e.g., Turkish, Finnish), whereas most only allow inflection and derivation.

3.4.5 Stemming and Lemmatization

In some applications, the differences between different morphological variants of the same base word are not particularly informative. For instance, should the query `[new car]` provide a different search result than `[new cars]`? The difference is that one query uses the singular form *car* and the other the plural *cars*.

Lemmatization is the process of mapping a word to its root form(s). It may also involve determining which affixes, vowel changes, wrappers or infixes or templates are involved, but typically not for most applications.

Stemming is the process of mapping a word to a canonical representation. The main difference from a linguistic point of view is that the output of a stemmer isn't necessarily a word (or morpheme) itself.

Example: Lemma *author*

It is very hard to know where to draw the line in stemming. Some systems take a strictly inflectional point of view, only stripping off inflectional affixes or templatic fillers. Others allow full derivational morphology. Even so, the line is unclear.

Consider the stem *author*. In the English Gigaword corpus,[4] which consists of a bit over a billion words of English newswire text, which is relatively well behaved, the following variants are observed

> antiauthoritarian, antiauthoritarianism, antiauthority, author, authoratative, authoratatively, authordom, authored, authoress, authoresses, authorhood, authorial, authoring, authorisation, authorised, authorises, authoritarian, authoritarianism, authoritarians, authoritative, authoritatively, authoritativeness, authorities, authoritory, authority, authorization, authorizations, authorize, authorized, authorizer, authorizers, authorizes, authorizing, authorless, authorly, authors, authorship, authorships, coauthor, coauthored, coauthoring, coauthors, cyberauthor, deauthorized, multiauthored, nonauthor, nonauthoritarian, nonauthorized, preauthorization, preauthorizations, preauthorized, quasiauthoritarian, reauthorization, reauthorizations, reauthorize, reauthorized, reauthorizes, reauthorizing, semiauthoritarian, semiauthorized, superauthoritarian, unauthorised, unauthoritative, unauthorized

We're not just looking for the substring *author*. Because of the complexity of morphological affixing in English, this is neither necessary or sufficient. To see that it's not sufficient, note that *urn* and *turn* are not related, nor are *knot* and *not*. To see that it's not necessary, consider *pace* and *pacing* or *take* and *took*.

Each of the words in the list above is etymologically derived from the word *author*. The problem is that over time, meanings drift. This makes it very hard to draw a line for which words to consider equivalent for a given task. The shared

[4]Graff, David and Christopher Cieri. 2003. *English Gigaword*. Linguistic Data Consortium (LDC). University of Pennsylvania. Catalog number LDC2003T05.

root of *author* and *authorize* is not only lost to most native speakers, the words are only weakly semeantically related, despite the latter form being derived from the former using the regular suffix *-ize*. The meaning has drifted.

In contrast, the relation between *notary* and *notarize* is regular and the meaning of the latter is fairly predictable from the meaning of the former. Moving down a derivational level, the relation between *note* and *notary* feels more opaque.

Suppose we search for `[authoritarian]`, perhaps researching organizational behavior. We probably don't want to see documents about coauthoring papers. We probably don't want to see documents containing *unauthoritative*, because that's usually used in a different context, but we might want to see documents containing the word *antiauthority*, and would probably want to see documents containing *antiauthoritarianism*. As you can see, it's rather difficult to draw a line here.

David A. Hull provides a range of interesting examples of the performance of a range of stemmers from the very simple (see Section 3.4.6) through the mainstream medium complex systems (see Section 3.4.7) to the very complex (Xerox's finite-state-transducer based stemmer).[5] For example, one of Hull's queries contained the word *superconductivity*, but matching documents contained only the word *superconductor*. This clearly requiring derivational morphology to find the right documents. A similar problem occurs for the terms *surrogate mother* versus *surrogate motherhood* and *genetic engineering* versus *genetically engineered* in another. Another example where stemming helped was *failure* versus *fail* in the context of bank failures.

An example where stemming hurt was in the compound term *client server* matching documents containing the words *serve* and *client*, leading to numerous false positive matches. This is really a frequency argument, as servers in the computer sense do serve data, but so many other people and organizations serve things to clients that it provides more noise than utility. Another example that caused problems was was reducing *fishing* to *fish*; even though this looks like a simple inflectional change, the nominal use of *fish* is so prevalent that documents about cooking and eating show up rather than documents about catching fish. Similar problems arise in lemmatizing *privatization* to *private*; the latter term just shows up in too many contexts not related to privatization.

These uncertainties in derivational stem equivalence are why many people want to restrict stemming to inflectional morphology. The problem with restricting to inflectional morphology is that it's not the right cut at the problem. Sometimes derivational stemming helps and sometimes inflectional stemming hurts.

Context Sensitivity of Morphology

Words are ambiguous in many different ways, including morphologically. Depending on the context, the same token may be interpreted differently. For instance, the plural noun *runs* is the same word as the present tense verb *runs*. For nouns, only *run* and *runs* are appropriate, both of which are also verbal forms.

[5]Hull, David A. 1996. Stemming algorithms: a case study for detailed evaluation. *Journal of the American Society for Information Science* **47**(1).

Additional verbal forms include *running* and *ran*. A word like *hyper* might be used as the short-form adjective meaning the same thing as *hyperactive*, or it could be a noun derived from the the verb *hype*, meaning someone who hypes something (in the publicicity sense).

3.4.6 A Very Simple Stemmer

One approach to stemming that is surprisingly effective for English (and other mostly suffixing languages) given its simplicity is to just map every word down to its first *k* characters. Typical values for *k* would be 5 or 6. This clearly misses prefixes like *un-* and *anti-*, it'll lump together all the other forms of *author* discussed in the previous section. It is also too agressive at lumping things together, merging words like *regent* and *regeneration*.

We provide an implementation of this simple stemmer in the class `PrefixStemmer`. The class is defined to extend `ModifyTokenTokenizerFactory`, the constructor passes the base tokenizer factory to the superclass, and the work is all done in the `modifyToken()` method,

```
public String modifyToken(String token) {
     return token.length() <= mPrefixLength
          ? token
          : token.substring(0,mPrefixLength);
```

There is also a `main()` method that wraps an Indo-European tokenizer factory in a prefix stemmer and displays the tokenizer's output.

The Ant target `prefix-stem-tokens` runs the program, with first argument given by property `prefixLen` and the second by property `text`.

```
> ant -DprefixLen=4 -Dtext="Joe smiled" prefix-stem-tokens

START   END TOKEN           START   END TOKEN
    0     3  |Joe|              4    10  |smil|
```

As you can see, the token *smiled* is reduced to *smil*, but *Joe* is unaltered.

In an application, the non-text tokens would probably also be normalized in some way, for instance by mapping all non-decimal numbers to *0* and all decimal numbers to *0.0*, or simply by replacing all digits with *0*. Punctuation might also get normalized into bins such as end-of-sentence, quotes, etc. The Unicode classes are helpful for this.

3.4.7 The `PorterStemmerTokenizerFactory` Class

The most widely used approach to stemming in English is that developed by Martin Porter in what has come to be known generically as the Porter stemmer.[6] One of the reasons for its popularity is that Porter and others have made it freely available and ported it to a number of programming languages such as Java.

[6]Porter, Martin. 1980. An algorithm for suffix stripping. *Program*. **14**:3.

The stemmer has a very simple input/output behavior, taking a string argument and returning a string. LingPipe incorporates Porter's implementation of his algorithm in the `PorterStemmerTokenizerFactory`. Porter's stemming operation is encapsulated in the static method `stem(String)`, which returns the stemmed form of a string.

The tokenizer factory implementation simply delegates the `modifyToken(String)` method in the superclass `ModifyTokenTokenizerFactory` to the `stem(String)` method.

Demo Code

We provide an example use of the stemmer in the demo class `PorterStemTokens`. The `main()` method does nothing more than create a Porter stemmer tokenizer,

```
TokenizerFactory f1 = IndoEuropeanTokenizerFactory.INSTANCE;
TokenizerFactory f2 = new PorterStemmerTokenizerFactory(f1);
```

and then display the results.

The Ant target `porter-stemmer-tokens` runs the program with a command line argument given by the value of the property `text`,

```
>         ant -Dtext="Smith was bravely charging the front lines"
porter-stemmer-tokens
```

```
Smith was bravely charging the front lines.
0123456789012345678901234567890123456789012
0         1         2         3         4

START   END TOKEN           START   END TOKEN
    0     5  |Smith|           27    30  |the|
    6     9  |wa|              31    36  |front|
   10    17  |brave|           37    42  |line|
   18    26  |charg|           42    43  |.|
```

Note that the token spans are for the token before stemming, so that the token *brave* starts at position 10 (inclusive) and ends at 17 (exclusive), spanning the substring of the input text *bravley*.

You can see that the resulting tokens are not necessarily words, with *was* stemmed to *wa* and *charging* stemmed to *charge*. The token *charge* also stems to *charg*. In many cases, the Porter stemmer strips off final *e* and *s* characters. It performs other word-final normalizations, such as stemming *cleanly* and *cleanliness* to *cleanli*.

You can also see from the reduction of *Bravely* to *Brave* that the stemmer is not case sensitive, but also does not case normalization. Typically the input or output would be case normalized. You can also see that punctuation, like the final period, is passed through unchanged.

The Porter stemmer does not build in knowledge of common words of English. It thus misses many non-regular relations. For instance, consider

```
> ant -Dtext="eat eats ate eaten eating" porter-stemmer-tokens

START   END TOKEN          START   END TOKEN
    0     3  |eat|            13    18  |eaten|
    4     8  |eat|            19    25  |eat|
    9    12  |at|
```

It fails to equate *ate* and *eaten* to the other forms. And it unfortunately reduces *at* to the same stem as would be found for the preposition of the same spelling.

The Porter stemmer continues to apply until no more reductions are possible. This provides the nice property that for any string `s`, the expression `stem(stem(s))` produces a string that's equal to `stem(s)`.

```
> ant -Dtext="fortune fortunate fortunately"
porter-stemmer-tokens

START   END TOKEN          START   END TOKEN
    0     7  |fortun|         18    29  |fortun|
    8    17  |fortun|
```

The Porter stemmer does not attempt to remove prefixes.

```
> ant -Dtext="antiwar unhappy noncompliant inflammable
tripartite" porter-stemmer-tokens

START   END TOKEN          START   END TOKEN
    0     7  |antiwar|        29    40  |inflamm|
    8    15  |unhappi|        41    51  |tripartit|
   16    28  |noncompli|
```

Thread Safety and Serialization

A Porter stemmer tokenizer factory is thread safe if its base tokenizer factory is thread safe. It is serializable if the base factory is serializable.

3.5 Soundex: Pronunciation-Based Tokens

For some applications involving unedited or transliterated text, it's useful to reduce tokens to punctuation. Although LingPipe does not contain a general pronunciation module, it does provide a combination of lightweight pronunciation and stemming through an implementation of the Soundex System. Soundex was defined in the early 20th century and patented by Robert C. Russell. [7] The main application was for indexing card catalogs of names. The LingPipe version is based on the a modified version of Soundex known as "American Soundex," which was used by the United States Census Bureau in the 1930s. You can still access census records by Soundex today.

[7] United States Patents 1,261,167 and 1,435,663.

3.5.1 The American Soundex Algorithm

Soundex produces a four-character long representation of arbitrary words that attempts to model something like a pronunciation class.

The Algorithm

The basic procedure, as implemented in LingPipe, is described in the following pseudocode, which references the character code table in the next section. The algorithm itself, as well as the examples, are based on pseudocode by Donald Knuth.[8]

1. Normalize input by removing all characters that are not Latin1 letters, and converting all other characters to uppercase ASCII after first removing any diacritics.
2. If the input is empty, return *0000*
3. Set the first letter of the output to the first letter of the input.
4. While there are less than four letters of output do:
   (a) If the next letter is a vowel, unset the last letter's code.
   (b) If the next letter is *A*, *E*, *I*, *O*, *U*, *H*, *W*, *Y*, continue.
   (c) If the next letter's code is equal to the previous letter's code, continue.
   (d) Set the next letter of output to the current letter's code.
5. If there are fewer than four characters of output, pad the output with zeros (*0*)
6. Return the output string.

Character Codes

Soundex's character coding attempts to group together letters with similar sounds. While this is a reasonable goal, it can't be perfect (or even reasonably accurate) because pronunciation in English is so context dependent.

The table of individual character codes is as follows.

Characters	*Code*
B, *F*, *P*, *V*	1
C, *G*, *J*, *K*, *Q*, *S*, *X*, *Z*	2
D, *T*	3
L	4
M, *N*	5
R	6

[8]Knuth, Donald E. 1973. *The Art of Computer Programming, Volume 3: Sorting and Searching.* 2nd Edition. Addison-Wesley. Pages 394-395.

Examples

Here are some examples of Soundex codes, drawn from Knuth's presentation.

Tokens	*Encoding*
Gutierrez	G362
Pfister	P236
Jackson	J250
Tymczak	T522
Ashcraft	A261
Robert, *Rupert*	R163
Euler, *Ellery*	E460

Tokens	*Encoding*
Gauss, *Ghosh*	G200
Hilbert, *Heilbronn*	H416
Knuth, *Kant*	K530
Lloyd, *Liddy*	L300
Lukasiewicz, *Lissajous*	L222
Wachs, *Waugh*	W200

3.5.2 The SoundexTokenizerFactory Class

LingPipe provides a tokenizer factory filter based on the American Soundex defined above in the class SoundexTokenizerFactory in the package com.aliasi.tokenizer.

Like the Porter stemmer tokenizer filter, there is a static utility method, here named soundexEncoding(String), for carrying out the basic encoding. Provide a token, get its American Soundex encoding back. Also like the Porter stemmer, the rest of the implementation just uses this static method to transform tokens as they stream by.

Demo

We provide a demo of the Soundex encodings in the class SoundexTokens. Like the Porter stemmer demo,[9] it just wraps an Indo-European tokenizer factory in a Soundex tokenizer factory.

```
TokenizerFactory f1 = IndoEuropeanTokenizerFactory.INSTANCE;
TokenizerFactory f2 = new SoundexTokenizerFactory(f1);
```

Also like the Porter stemmer demo, the main() method consumes a single command-line argument and displays the text with indexes and then the resulting tokens.

We provide an Ant target soundex-tokens, which feeds the value of property text to the program.

```
> ant -Dtext="Mr. Roberts knows Don Knuth." soundex-tokens
```

```
Mr. Roberts knows Don Knuth.
0123456789012345678901234567
0         1         2

START   END TOKEN         START   END TOKEN
    0     2  |M600|          18    21  |D500|
    2     3  |0000|          22    27  |K530|
```

[9]While we could've encapsulated the text and display functionality, we're trying to keep our demos relatively independent from one another so that they are easier to follow.

```
   4    11  |R163|           27    28  |0000|
  12    17  |K520|
```

For instance, the first token spans from 0 to 2, covering the text *Mr* and producing the Soundex encoding `M600`. The following period, spanning from position 2 to 3, gets code `0000`, because it is not all Latin1 characters. The token spanning 12 to 17, covering text *Roberts*, gets code `R163`, the same as *Robert* in the table of examples. Similarly, we see that *Knuth* gets its proper code from positions 22 to 27.

From this example, we see that it might make sense to follow the Soundex tokenizer with a stop list tokenizer that removes the token *0000*, which is unlikely to be very informative in most contexts.

3.5.3 Variants of Soundex

Over the years, Soundex-like algorithms have been developed for many languages. Other pronunciation-based stemmers have also been developed, perhaps the most popular being Metaphone and its variant, Double Metaphone.

3.6 Character Normalizing Tokenizer Filters

We demonstrated how to use the International Components for Unicide (ICU) package for normalizing sequences of Unicode characters in the companion volume, *Text Processing in Java*. In this section, we'll show how to plug that together with tokenization to provide tokenizers that produce output with normalized text.

The alternative to normalizing with a token filter would be to normalize the text before input. As a general rule, we prefer not to modify the original underlying text if it can be avoided. A case where it makes sense to parse or modify text before processing would be an XML and HTML document.

3.6.1 Demo: ICU Transliteration

We wrote a demo class, `UnicodeNormTokenizerFactory`, which uses the the `Transliterate` class from ICU to do the work (see the section on transliteration in the companion volume, *Text Processing in Java*, for an introduction to the class and more examples). A transliterator is configured with a transliteration scheme, so we configure our factory the same way,

```
public class UnicodeNormTokenizerFactory
     extends ModifyTokenTokenizerFactory {

     private final String mScheme;
     private final Transliterator mTransliterator;

     public UnicodeNormTokenizerFactory(String scheme,
                                        TokenizerFactory f) {
         super(f);
```

```
        mScheme = scheme;
        mTransliterator = Transliterator.getInstance(scheme);
    }

    public String modifyToken(String in) {
        return mTransliterator.transliterate(in);
    }
```

As with our other filters, we extend `ModifyTokenTokenizerFactory`, and then do the actualy work in the `modifyToken(String)` method. Here, the constructor stores the scheme and creates the transliterator using the static factory method `getInstance(String)` from `Transliterator`. Transliteration operations in ICU are thread safe as long as the transliterator is not modified during the transliteration.

The class also includes a definition of a serialization proxy, which only needs to store the scheme and base tokenizer in order to reconstruct the transliterating tokenizer.

The Ant target `unicode-norm-tokens` runs the program, using the property `translit.scheme` to define the transliteration scheme (see the section on Unicode transliteration with ICU in the companion volume, *Text Processing in Java*, for a description of the available schemes and the language for composing them) and property `text` for the text to modify.

```
> ant "-Dtext=Déjà vu" "-DtranslitScheme=NFD; [:Nonspacing Mark:]
Remove; NFC" unicode-norm-tokens

translitScheme=NFD; [:Nonspacing Mark:] Remove; NFC

Déjà vu
0123456

START   END TOKEN
    0     4  |Deja|
    5     7  |vu|
```

To get the output encoded correctly, we had to reset `System.out` to use UTF-8 (see the section on standard input and output in the the companion volume, *Text Processing in Java*).

3.7 Penn Treebank Tokenization

One of the more widely used resources for natural language processing is the Penn Treebank.[10]

[10]Marcus, Mitchell P., Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. 1999. *Treebank-3*. Linguistic Data Consortium. University of Pennsylvania. Catalog number LDC99T42.

3.7.1 Pre-Tokenized Datasets

Like many older data sets, the Penn Treebank is presented only in terms of its tokens, each separated by a space. So there's no way to get back to the underlying text. This is convenient because it allows very simple scripts to be used to parse data. Thus it is easy to train and evaluate models. You don't even have to write a tokenizer, and there can't be any tokenization errors. More modern datasets tend to use either XML infix annotation or offset annotation, allowing the boundaries of tokens in the original text to be recovered.

The major drawback is that there is no access to the underlying text. As a consequence, there is no way to train whitespace-sensitive models using the Penn Treebank as is (it might be possible to reconstitute the token alignments given the original texts and the parser).

Although it is easy to train and evaluate in a pre-tokenized data set, using it on new data from the wild requires an implementation of the tokenization scheme used to create the corpus.

3.7.2 The Treebank Tokenization Scheme

In this section, we provide an overview of the Penn Treebank's tokenization scheme.[11] The major distinguishing feature of the Treebank's tokenizer is that they analyze punctuation based on context. For instance, a period at the end of the sentence is treated as a standalone token, whereas an abbreviation followed by a period, such as *Mr.*, is treated as a single token. Thus the input needs to already be segmented into sentences, a process we consider in Chapter 16.

The Treebank tokenizer was designed for the ASCII chaacters (U+0000–U+007F), which means it is not defined for characters such as Unicode punctuation. Following the convention of LaTeX rather than Unicode, the Treebank tokenizer analyzes a quote character (") as two grave accents (``) if it's an opening quote and two apostrophes ('') if it's a closing quote. Like periods, determining the status of the quotes requires context.

The Treebank tokenizer is also English specific. In its context-sensitive treatment of apostrophes ('), it attempts to distinguish the underlying words making up contractions together, so that a word like *I'm* is split into two tokens, *I* and *'m*, the second of which is the contracted form of *am*. To handle negative contractions, it keeps the negation together, splitting *don't* into *do* and *n't*. The reason for this is that the Treebank labels syntactic categories, and they wanted to assign *I* to a pronoun category and *'m* to an auxiliary verb category just like *am*; similarly, *do* is an auxiliary verb and *n't* a negative particle.

Ellipses, which appear in ASCII text as a sequence of periods (...), are handled as single tokens. Pairs of hyphens (--), which are found as an ASCII replacement for an en-dash (–).

Other than the treatment of periods, quotation marks and apostrophes in the aforementioned contexts, the Treebank tokenizer splits out other punctuation

[11] We draw from the online description `http://www.cis.upenn.edu/~treebank/tokenization.html` (downloaded 30 July 2010) and Robert MacIntyre's implementation as a SED script, released under the Gnu Public License, in `http://www.cis.upenn.edu/~treebank/tokenizer.sed`.

characters into their own tokens.

The ASCII bracket characters, left and right, round and square and curly, are replaced with acronyms and hyphens. For instance, U+005B, LEFT SQUARE BRACKET (*[*) produces the token *-LSB-*. The acronym *LSB* stands for left square bracket; other brackets are coded similarly alternating *R* for right in the first position, and *R* for round and *C* for curly in the second position.

3.7.3 Demo: Penn Treebank Tokenizer

We provide a port and generalization of the Penn Treebank tokenizer based on Robert MacIntyre's SED script in the class `PennTreebankTokenizerFactory`. We have also generalized it to Unicode character codes where possible.

The original SED script worked by transforming the input string by adding or deleting spaces and transforming characters. If we transformed the input, the positions of the tokens would all be off unless we somehow kept track of them. Luckily, an easier solution presents itself with Java regular expressions and LingPipe's built-in regex tokenizer factories and token filter.

Our implementation works in three parts. First, the original string is transformed, but in a position-preserving way. This mainly takes care of issues in detecting opening versus closing quotes. Second, a regular expression tokenizer is used to break the input into tokens. Third, the tokens are transformed on a token-by-token basis into the Penn Treebank's representation.

Construction

Before any of these steps, we declare the tokenizer factory to extend the token-modifying token factory and to be serailizable,

```
public class PennTreebankTokenizerFactory
     extends ModifyTokenTokenizerFactory
     implements Serializable {

     public static final TokenizerFactory INSTANCE
         = new PennTreebankTokenizerFactory();

     private PennTreebankTokenizerFactory() {
         super(new RegExTokenizerFactory(BASE_REGEX,
                                         Pattern
                                         .CASE_INSENSITIVE));
     }
```

We had to supply the base tokenizer factory as an argument to `super()` in order to supply the superclass with its base tokenizer. This is a regex-based tokenizer factory which we describe below. There's also a serialization proxy implementation we don't show.

We chose to implement the Penn Treebank tokenizer as a singleton, with a private constructor and static constant instance. The constructor could've just as easily been made public, but we only need a single instance.

Input String Munging

In the first processing step, we override the tokenization method itself to trasnform the input text using a sequence of regex replace operations (see the section on replace in the regular expression chapter of the companion volume, *Text Processing in Java*),

```
@Override
public Tokenizer tokenizer(char[] cs, int start, int len) {
     String s = new String(cs,start,len)
         .replaceAll("(^\")","\u201C")
         .replaceAll("(?<=[ \\p{Z}\\p{Ps}])\"","\u201C")
         .replaceAll("\"","\u201D")
         .replaceAll("\\{Pi}","\u201C")
         .replaceAll("\\{Pf}","\u201D");
     return super.tokenizer(s.toCharArray(),0,len);
}
```

The `replaceAll()` methods chain, starting with a string produced from the character slice input. The first replacement replaces an instance of U+0022, QUOTATION MARK, ("), the ordinary typewriter non-directional quotation mark symbol, with U+201C, LEFT DOUBLE QUOTATION MARK (“). The second replacement replaces a quotation mark after characters of class `Z` (separators such as spaces, line or paragraph separators) or class `Ps`, the start-punctuation class, which includes all the open brackets specified in the original SED script. The context characters are specified as a non-capturing lookbehind (see the relevant section of the companion volume). The third replacement converts remaining quote characters U+201D, RIGHT DOUBLE QUOTATION MARK ((”)). The fourth replacement converts any character typed as initial quote punctuation (`Pi`) and replaces it with a left double quotation mark, and similarly for final quote punctuation (`Pf`).

We take the final string and convert it to a character array to send the superclass's implementation of `tokenizer()`. We have been careful to only use replacements that do not modify the length of the input, so that all of our offsets for tokens will remain correct.

Base Tokenizer Regex

The implementation of `tokenize()` in the superclass `ModifyTokenTokenizerFactory` invokes the base tokenizer, which was supplied to the constructor, then feeds the results through the `modifyToken()` method, which we describe below.

The base tokenizer factory is constructed from a constant representing the regular expression and the specification that matching is to be case insensitive. The regex used is

```
static final String BASE_REGEX
     = "("
     + "\\.\\.\\." + "|" + "--"
```

```
    + "|" + "can(?=not\\b)" + "|" + "d'(?=ye\\b)"
    + "|" + "gim(?=me\\b)" + "|" + "lem(?=me\\b)"
    + "|" + "gon(?=na\\b)" + "|" + "wan(?=na\\b)"
    + "|" + "more(?='n\\b)" + "|" + "'t(?=is\\b)"
    + "|" + "'t(?=was\\b)" + "|" + "ha(?=ddya\\b)"
    + "|" + "dd(?=ya\\b)" + "|" + "ha(?=tcha\\b)"
    + "|" + "t(?=cha\\b)"
    + "|" + "'(ll|re|ve|s|d|m|n)" + "|" + "n't"
    + "|" + "[\\p{L}\\p{N}]+(?=(\\.$|\\.([\\{Pf}\"'])+|n't))"
    + "|" + "[\\p{L}\\p{N}\\.]+"
    + "|" + "[^\\p{Z}]"
    + ")";
```

We have broken it out into pieces and used string concatenation to put it back together so it's more readable.

The first disjunct allows three period characters to be a token, the second a pair of dashes.

The next thirteen disjuncts are spread over several lines. Each of these simply breaks a compound apart. For instance, *cannot* will be split into two tokens, *can* and *not*. This is achieved in each case by matching the first half of a compound when followed by the second half. The following context is a positive lookahead, requiring the rest of the compound and then a word boundary (regex `\b` is the word boundary matcher).

The next two disjuncts pull out the contracted part of contractions, such as *'s*, *'ve* and *n't*.

Next, we match tokens consisting of sequences of letters and/or numbers (Unicode types `L` and `N`). The first disjunct pulls off tokens that directly precede the final period or the sequence *n't*. The disjunct uses positive lookahead to match the following context. Because regexes are greedy, the disjunct just described will match if possible. If not, periods are included alongw ith letters and numbers (but no other punctuation).

The final disjunct treats any other non-separator character as a token.

Modified Token Output

To conform to the Penn Treebank's output format for characters, we do a final replace on the tokens, using

```
@Override
public String modifyToken(String token) {
    return token
        .replaceAll("\\(","-LRB-").replaceAll("\\)","-RRB-")
        .replaceAll("\\[","-LSB-").replaceAll("\\]","-RSB-")
        .replaceAll("\\{","-LCB-").replaceAll("\\}","-RCB-")
        .replaceAll("\u201C","“").replaceAll("\u201D","”");
}
```

Each of the ASCII brackets is replaced with a five-letter sequence. Any initial quote characters are replaced with two grave accents (“) and then final quote characters are replaced with two apostrophes (’’).

Running It

The Ant target `treebank-tokens` runs the Penn Treebank tokenizer, supplying the value of the Ant property `text` as the first argument, which is the string to tokenize.

```
>       ant -Dtext="Ms.  Smith’s papers’ - weren’t (gonna) miss."
treebank-tokens
Ms. Smith’s papers’ -- weren’t (gonna) miss.
01234567890123456789012345678901234567890123
0         1         2         3         4

START   END TOKEN          START   END TOKEN
    0     3  |Ms.|            27    30  |n’t|
    4     9  |Smith|          31    32  |-LRB-|
    9    11  |’s|             32    35  |gon|
   12    18  |papers|         35    37  |na|
   18    19  |’|              37    38  |-RRB-|
   20    22  |--|             39    43  |miss|
   23    27  |were|           43    44  |.|
```

This short example illustrates the different handling of periods sentence-internally and finally, the treatment of contractions, distinguished tokens like the double-dash, and compounds like *gonna*. It also shows how the brackets get substituted, for instance the left with token *-LRB-*; note that these have their original spans, so that the token *-RRB-* spans from 37 (inclusive) to 38 (exclusive), the position of the right round bracket in the original text.

The first disjunct after all the special cases allows a sequence of letters and numbers to form a token subject to the negative lookahead constraint. The negative lookahead makes sure that periods are not picked up as part of tokens if they are followed by the end of input or a non-empty sequence of punctuation characters including final punctuation like right brackets (Unicode class `Pf`), the double quote character or the apostrophe.

```
> ant -Dtext="(John ran.)" treebank-tokens
(john ran.)
01234567890

START   END TOKEN          START   END TOKEN
    0     1  |-LRB-|           9    10  |.|
    1     5  |John|           10    11  |-RRB-|
    6     9  |ran|
```

The period is not tokenized along with *ran*, because it is followed by a right round bracket character ()), an instance of final punctuation matching `Pf`. Without the negative lookahead disjunct appearing before the general disjunct, the period would’ve been included.

The final disjunct in the negative lookahead constraint ensures the suffix *n't* is kept together as we saw in the first example; without lookahead, the *n* would be part of the previous token.

Given how difficult it is to get a standard quote character into a property definition,[12] we've used a properties file in `config/text2.properties`. The content of the properties file is

```
text=I say, \\"John runs\\".
```

Recall that in Java properties files, strings are parsed in the same way as string literals in Java programs. In particular, we can use Unicode and Java backslash escapes,[13] We run the Ant target by specifying the properties file on the command line,

```
> ant -propertyfile=config/text2.properties treebank-tokens
I say, "John runs".
0123456789012345678
0         1

START   END TOKEN          START   END TOKEN
    0     1  |I|               8    12  |John|
    2     5  |say|            13    17  |runs|
    5     6  |,|              17    18  |''|
    7     8  |``|             18    19  |.|
```

The output demonstrates how open double quotes are treated differently than close double quotes. Like the bracket token substitutions, the two characters substituted for quotes still span only a single token of the input.

Discussion

There are several problems with the Penn Treebank tokenizer as presented. The problems are typical of systems developed on limited sets of data with the goal of parsing the data at hand rather than generalizing. We fixed one such problem, only recognizing ASCII quotes and only recognizing ASCII brackets, by generalizing to Unicode classes in our regular expressions.

Another problem, which is fairly obvious from looking at the code, is that we've only handled a small set of contractions. We haven't handled *y'all* or *c'mon*, which are in everyday use in English. I happen to be a fan of nautical fiction, but the tokenizer will stumble on *fo'c's'le*, the conventional contraction of *forecastle*. Things get even worse when we move into technical domains like chemistry or mathematics, where apostrophes are used as primes or have other meanings.

We've also only handled a small set of word-like punctuation. What about the e-mail smiley punctuation, *:-)*? It's a unit, but will be broken into three tokens by the current tokenizer.

[12]Although the double quote character itself can be escaped in any halfway decent shell, they all do it differently. The real problem is that Ant's scripts can't handle `-D` property specifications with embedded quotes.

[13]Unicode escapes for the quote character will be interpreted rather than literal, so you need to use the backslash. And the backslash needs to be escaped, just as in Java source.

3.8 Adapting to and From Lucene Analyzers

In the section Lucene analysis in the Lucene chapter of the companion volume, *Text Processing in Java*, we discuss the role of Lucene's analyzers in the indexing process. A Lucene document maps field names to text values. A Lucene analyzer maps a field name and text to a stream of tokens. In this section, we show how to convert a Lucene analyzer to a LingPipe tokenizer factory and vice-versa.

3.8.1 Adapting Analyzers for Tokenizer Factories

Because of the wide range of basic analyzers and filters employed by Lucene, it's useful to be able to wrap a Lucene analyzer for use in LingPipe. For instance, we could analyze Arabic with stemming this way, or even Chinese.

The demo class `AnalyzerTokenizerFactory` implements LingPipe's `TokenizerFactory` interface based on a Lucene `Analyzer` and field name. In general design terms, the class adapts an analyzer to do the work of a tokenizer factory.

The class is declared to implement the tokenizer factory interface to be serializable,

```
public class AnalyzerTokenizerFactory
     implements TokenizerFactory, Serializable {

     private final Analyzer mAnalyzer;
     private final String mFieldName;

     public AnalyzerTokenizerFactory(Analyzer analyzer,
                                     String fieldName) {
         mAnalyzer = analyzer;
         mFieldName = fieldName;
     }
```

The constructor simply stores an analyzer and field name in private final member variables.

The only method we need to implement is `tokenizer()`,

```
public Tokenizer tokenizer(char[] cs, int start, int len) {
     Reader reader = new CharArrayReader(cs,start,len);
     TokenStream tokenStream
         = mAnalyzer.tokenStream(mFieldName,reader);
     return new TokenStreamTokenizer(tokenStream);
}
```

It creates a reader from the character array slice and then uses the stored analyzer to convert it to a Lucene `TokenStream`. A new instance of `TokenStreamTokenizer` is constructed based on the token stream and returned.

The `TokenStreamTokenizer` class is a nested static class. It is defined to extend LingPipe's `Tokenizer` base class and implement Java's `Closeable` interface.

```
static class TokenStreamTokenizer extends Tokenizer {

    private final TokenStream mTokenStream;
    private final TermAttribute mTermAttribute;
    private final OffsetAttribute mOffsetAttribute;

    private int mLastTokenStartPosition = -1;
    private int mLastTokenEndPosition = -1;

    public TokenStreamTokenizer(TokenStream tokenStream) {
        mTokenStream = tokenStream;
        mTermAttribute
            = mTokenStream.addAttribute(TermAttribute.class);
        mOffsetAttribute
            = mTokenStream.addAttribute(OffsetAttribute.class);
    }
```

The constructor stores the token stream in a member variable, along with the term and offset attributes it adds to the token stream. The token start positions are initialized to -1, which is the proper return value before any tokens have been found.

The `nextToken()` method is implemented by delegating to the contained token stream's `incrementToken()` method, which advances the underlying token stream.

```
@Override
public String nextToken() {
    try {
        if (mTokenStream.incrementToken()) {
            mLastTokenStartPosition
                = mOffsetAttribute.startOffset();
            mLastTokenEndPosition
                = mOffsetAttribute.endOffset();
            return mTermAttribute.term();
        } else {
            closeQuietly();
            return null;
        }
    } catch (IOException e) {
        closeQuietly();
        return null;
    }
}
```

After the increment, the start and end positions are stored befre returning the term as the next token. If the token stream is finished or the increment method throws an I/O exception, the stream is ended and closed quietly (see below).

```
public void closeQuietly() {
    try {
        mTokenStream.end();
    } catch (IOException e) {
        /* ignore */
    } finally {
        Streams.closeQuietly(mTokenStream);
    }
}
```

The methods for start and end just return the stored values. For example,

```
@Override
public int lastTokenStartPosition() {
    return mLastTokenStartPosition;
}
```

The main() method for the demo takes a command-line argument for the text. It sets up a standard Lucene analyzer, then constructs a tokenizer factory from it using the field text (the field doesn't matter to the standard analyzer, but may matter for other analyzers).

```
String text = args[0];

StandardAnalyzer analyzer
    = new StandardAnalyzer(Version.LUCENE_30);
AnalyzerTokenizerFactory tokFact
    = new AnalyzerTokenizerFactory(analyzer,"foo");

DisplayTokens.displayTextPositions(text);
DisplayTokens.displayTokens(text,tokFact);
```

We then just display the text and tokens as we have in previous demos.

The Ant target lucene-lp-tokens runs the example, with property text passed in as the command-line argument.

```
> ant -Dtext="Mr. Smith is happy!" lucene-lp-tokens
Mr. Smith is happy!
0123456789012345678
0         1

START   END TOKEN           START   END TOKEN
    0     2  |mr|              13    18  |happy|
    4     9  |smith|
```

As before, we see the stoplisting and case normalization of the standard Lucene analyzer.

An Arabic Analyzer with Stemming and Stoplisting

The contributed Lucene class ArabicAnalyzer implements Larkey, Ballesteros and Connell's tokenizer and stemmer.[14] This class performs basic word segmentation, character normalization, "light" stemming, and stop listing, with a

[14]Larkey, Leah, Lisa Ballesteros and Margaret Connell. 2007. Light Stemming for Arabic Information Retrieval. In A. Soudi, A. van den Bosch, and G. Neumann, (eds.) *Arabic Computational Morphology*. 221–243. Springer.

configuarable stop list. Because it's a contributed class, it's found in a separate jar, `lucene-analyzers-3.0.2.jar`, which we've included with the distribution and inserted into the classpath for this chapter's Ant build file.

We provide a demo class `ArabicTokenizerFactory`, which provides a static singleton implementation of a tokenizer factory wrapping Lucene's Arabic analyzer. We don't need anything other than the Lucene `Analyzer` and LingPipe `TokenizerFactory`,

```
public static final Analyzer ANALYZER
    = new ArabicAnalyzer(Version.LUCENE_30);

public static final TokenizerFactory INSTANCE
    = new AnalyzerTokenizerFactory(ANALYZER,"foo");
```

In the file `src/tok/config/arabic-sample.utf8.txt`, we have placed some Arabic text[15] encoded with UTF-8; the first few lines of the file are as follows.

أولويوكي (Oulujoki) نهر في مقاطعة أولو بفنلندا. منشؤه هو بحيرة أولويارفي، يغطي جزءاً كبيرا من إقليم كاينو ويصب في بحر البلطيق. النهر مجهز ب ١٢ محطة لتوليد الطاقة الكهرمائية بقدرة إنتاجية تبلغ ٥٤١ ميغاواط. ...

There is a `main()` class that reads a file name from the command line, reads the file in and tokenizes it for display. The Ant target `arabic-tokens` runs the tokenizer over a file specified by name as the first command specified file,

```
> ant "-Dfile=config\arabic-sample.utf8.txt" arabic-tokens
file=config\arabic-sample.utf8.txt
 -----text-----
أولويوكي (Oulujoki) ...
...
from: http://ar.wikipedia.org/wiki/نهر_أولويوكي
-----end text-----
START   END TOKEN
    0     8  |اولويوك|
   10    18  |oulujoki|
   20    23  |نهر|
   27    33  |مقاطع|
...
  292   295  |org|
  296   300  |wiki|
  301   309  |اولويوك|
  310   313  |نهر|
```

Note that the text mixes Roman characters and Arabic characters. The source URL is part of the text. The Roman characters are separated out as their own tokens and retained, but all punctuation is ignored. The very first token, which is also part of the URL, is stemmed.

[15] Downloaded from `http://ar.wikipedia.org/wiki/%D9%86%D9%87%D8%B1_%D8%A3%D9%88%D9%84%D9%88%D9%8A%D9%88%D9%83%D9%8A` on 9 July 2010.

3.8.2 Adapting Tokenizer Factories for Analyzers

Recall that a Lucene analyzer maps fields and readers to token streams. We can implement an analyzer given a mapping from fields to tokenizer factories. We will also use a default tokenizer factory to apply to fields that are not defined in the map.

The demo class TokenizerFactoryAnalyzer provides such an implementation. The class definition and constructor are

```
public class TokenizerFactoryAnalyzer extends Analyzer {

     private final Map<String,TokenizerFactory> mTfMap;

     private final TokenizerFactory mDefaultTf;

     public
     TokenizerFactoryAnalyzer(Map<String,
                                   TokenizerFactory> tfMap,
                                 TokenizerFactory defaultTf) {
         mTfMap = new HashMap<String,TokenizerFactory>(tfMap);
         mDefaultTf = defaultTf;
   }
```

The mapping and default factory are stored as final member variables.

Version 3.0 of Lucene has been defined to allow greater efficiency through object pooling, such as reuse of token streams. This makes the implementation of tokenStream() more complex than may at first seem necessary.

```
@Override
public TokenStream tokenStream(String fieldName,
                                Reader reader) {
     TokenizerTokenStream tokenizer
         = new TokenizerTokenStream();
     tokenizer.setField(fieldName);
     try {
         tokenizer.reset(reader);
         return tokenizer;
     } catch (IOException e) {
         return new EmptyTokenStream();
     }
}
```

The method creates a new instance of TokenizerTokenStream, the definition of which we show below. It then sets the field name using the method setField() and the reader using the method reset(). If the setting a field throws an exception, the tokenStream() method returns an instance EmptyTokenStream.[16]

We also define the reusableTokenStream() method for efficiency, which reuses the previous token stream if there is one or uses tokenStream() to create one.

[16]From the package org.apache.lucene.analysis.miscellaneous, in the contributed analyzers library, lucene-analyzers-3.0.2.jar.

The class TokenizerTokenStream is defined as a (non-static) inner class.

```
class TokenizerTokenStream
     extends org.apache.lucene.analysis.Tokenizer {

     private TermAttribute mTermAttribute;
     private OffsetAttribute mOffsetAttribute;
     private PositionIncrementAttribute mPositionAttribute;

     private String mFieldName;
     private TokenizerFactory mTokenizerFactory;
     private char[] mCs;
     private Tokenizer mTokenizer;

     TokenizerTokenStream() {
         mOffsetAttribute = addAttribute(OffsetAttribute.class);
         mTermAttribute = addAttribute(TermAttribute.class);
         mPositionAttribute
             = addAttribute(PositionIncrementAttribute.class);
     }
```

It is defined to extend Lucene's Tokenizer interface, whose full package, org.apache.lucene.analysis, is included to avoid conflict with LingPipe's Tokenizer class. Lucene's Tokenizer class is a subclass of TokenStream defined to support basic tokenizers (the other abstract base class supports token stream filters). As with our use of token streams in the previous section to adapt analyzers to tokenizer factories, we again add and store the term, offset and position increment attributes.

The field and reader are set and reset as follows.

```
public void setField(String fieldName) {
     mFieldName = fieldName;
}

@Override
public void reset(Reader reader) throws IOException {
     mCs = Streams.toCharArray(reader);
     mTokenizerFactory
         = mTfMap.containsKey(mFieldName)
         ? mTfMap.get(mFieldName)
         : mDefaultTf;
     reset();
}

@Override
public void reset() throws IOException {
     if (mCs == null) {
         String msg = "Cannot reset after close()"
             + " or before a reader has been set.";
         throw new IOException(msg);
     }
```

```
    mTokenizer = mTokenizerFactory.tokenizer(mCs,0,mCs.length);
}
```

The setField() method simply stores the field name. The reset(Reader) method extracts and stores the characters from the reader using LingPipe's utility method toCharArray(). The tokenizer factory is retrieved from the map using the field name as a key, returning the default tokenizer factory if the field name is not in the map. After setting the characters and tokenizer, it calls the reset() method. The reset() method sets the tokenizer based on the current tokenizer factory and stored strings, throwing an exception if reset(Reader) has not yet been called or the close() method has been called.

Once a token stream is set up, its incrementToken() method sets up the properties for the next token if there is one.

```
@Override
public boolean incrementToken() {
    String token = mTokenizer.nextToken();
    if (token == null)
        return false;
    char[] cs = mTermAttribute.termBuffer();
    token.getChars(0,token.length(),cs,0);
    mTermAttribute.setTermLength(token.length());
    mOffsetAttribute
        .setOffset(mTokenizer.lastTokenStartPosition(),
                   mTokenizer.lastTokenEndPosition());
    return true;

}
```

It gathers the token itself from the LingPipe tokenizer stored in member variable mTokenizer. If the token's null, we've reached the end of the token stream, and return false from the incrementToken() method to signal the end of stream. Otherwise, we grab the character buffer from the term attribute and fill it with the characters from the token using the String method getChars(). We also set the Lucene tokenizr's offset attribute based on the LingPipe tokenizer's start and end position.

Lucene's Tokenizer class specifies a method end that advances the token position past the final token; we set the offset to indicate there is no more of teh input string left. The close() method, which is intended to be called after a client is finished with the token stream, frees up the character array used to store the characters being tokenized.

Tokenizer Factory Analzyer Demo

There's a main() method in the TokenizerFactoryAnalzyer class that we will use as a demo. It mirrors the Lucene analysis demo.

```
String text = args[0];

Map<String,TokenizerFactory> fieldToTokenizerFactory
```

```
     = new HashMap<String,TokenizerFactory>();
fieldToTokenizerFactory
     .put("foo",IndoEuropeanTokenizerFactory.INSTANCE);
fieldToTokenizerFactory
     .put("bar",new NGramTokenizerFactory(3,3));

TokenizerFactory defaultFactory
     = new RegExTokenizerFactory("\\S+");

TokenizerFactoryAnalyzer analyzer
     = new TokenizerFactoryAnalyzer(fieldToTokenizerFactory,
                                    defaultFactory);
```

First, we grab the text from the first command-line argument. Then we set up the mapping from field names to tokenizer factories, using an Indo-European tokenizer factory for field foo and a trigram tokenizer factory for field bar. The default tokenizer factory is based on a regex matching sequences of non-space characters. Finally, the analyzer is constructed from the map and default factory. The rest just prints the tokens and the results of the analysis.

The Ant target lp-lucene-tokens runs the command with the value of property text passed in as an argument.

```
> ant -Dtext="Jim ran." lp-lucene-tokens
Jim ran.
01234567

Field=foo                    Field=jib
  Pos Start   End             Pos Start   End
    1     0     3 Jim           1     0     3 Jim
    1     4     7 ran           1     4     8 ran.
    1     7     8 .

Field=bar
  Pos Start   End
    1     0     3 Jim
    1     1     4 im
...
    1     5     8 an.
```

We see the tokens produced in each field are derived in the case of foo from the Indo-European tokenzier factory, in the case of bar from the trigram tokenizer factory, and in the case of the feature jar, the default tokenizer factory. In all cases, the position increment is 1; LingPipe's tokenization framework is not set up to account for position offsets for phrasal search.

3.9 Tokenizations as Objects

In our examples so far, we have used the iterator-like streaming interface to tokens.

3.9.1 Tokenizer Results

The `Tokenizer` base class itself provides two ways to get at the entire sequence of tokens (and whitespaces) produced by a tokenizer.

The method `tokenize()` returns a string array containing the sequence of tokens produced by the tokenizer.

```
Tokenizer tokenizer = tokFactory.tokenizer(cs,0,cs.length);
String[] tokens = tokenizer.tokenize();
```

The method `tokenize(List<String>,List<String>)` adds the lists of tokens to the first list specified and the list of tokens to the second list. The usage pattern is

```
Tokenizer tokenizer = tokFactory.tokenizer(cs,0,cs.length);
List<String> tokens = new ArrayList<String>();
List<String> whitespaces = new ArrayList<String>();
tokenizer.tokenize(tokens,whitespaces);
```

In both cases, the results are only sequences of tokens and whitespaces. The underlying string being tokenized nor the positions of the tokens are part of the result.

3.9.2 The `Tokenization` Class

The class `Tokenization` in `com.aliasi.tokenizer` represents all of the information in a complete tokenization of an input character sequence. This includes the sequence of tokens, the sequence of whitespaces, the character sequence that was tokenized, and the start and end positions of each token.

Constructing a `Tokenization`

There are three constructors for the `Tokenization` class. Two of them, `Tokenization(char[],int,int,TokenizerFactory)` and `Tokenization(CharSequence,TokenizerFactory)`, take a tokenizer factory and a sequence of characters as input, and store the results of the tokenizer produced by the tokenizer factory. The third constructor takes all of the components of a tokenization as arguments, the text, tokens, whitespaces, and token positions, `Tokenization(String,List<String>,List<String>,int[],int[])`.

The constructors all store copies of their arguments so that they are not linked to the constructor arguments.

Getters for `Tokenization`

Once constructed, a tokenization is immutable. There are getters for all of the information in a tokenization. The method `text()` returns the underlying text that was tokenized. The method `numTokens()` returns the total number of tokens (there will be one more whitespace than tokens). The methods `token(int)` and `whitespace(int)` return the token and whitespace at the specified position. The methods `tokenStart(int)` and `tokenEnd(int)` return the starting offset (inclusive) of the token and the ending position (exclusive0 for the token at the specified index.

There are also methods to retrieve sequences of values. The method `tokenList()` returns an unmodifiable view of the underlying tokens and `tokens()` returns a copy of an array of the tokens. The methods `whitespaceList()` and `whitespaces()` are similar.

Equality and Hash Codes

Two tokenization objects are equal if their character sequences poroduce the same strings, they have equivalent token and whitespace lists, and the arrays of start and end positions for tokens are the same. Hash codes are defined consistently with equality.

Serialization

A tokenization may be serialized. The deserialized object is a `Tokenization` that is equal to the first.

Thread Safety

Tokenizations are immutable and completely thread safe once constructed.

Demo of `Tokenization`

The demo class `DisplayTokenization` provides an example of how the `Tokenization` class may be used. The work is in the `main()` method, which begins as follows.

```
TokenizerFactory tokFact
     = new RegExTokenizerFactory("\\p{L}+");

Tokenization tokenization = new Tokenization(text,tokFact);

List<String> tokenList = tokenization.tokenList();
List<String> whitespaceList = tokenization.whitespaceList();
String textTok = tokenization.text();
```

It begins by creatig a tokenizer factory from a regex that matches arbitrary sequences of letters as tokens. Then it creates the `Tokenization` object using the text from the command-line argument and the tokenizer factory. Then we collect up the list of tokens, list of whitespaces and underlying text. The command continues by looping over the tokens.

```
for (int n = 0; n < tokenization.numTokens(); ++n) {
     int start = tokenization.tokenStart(n);
     int end = tokenization.tokenEnd(n);
     String whsp = tokenization.whitespace(n);
     String token = tokenization.token(n);
```

For each token, it extracts the start position (inclusive), end position (exclusive), whitespace before the token, and the token itself. These are printed. After the loop, it prints out the final whitespace.

```
String lastWhitespace
     = tokenization.whitespace(tokenization.numTokens());
```

The Ant target `tokenization` runs the demo command with the value of property `text` as the command-line argument.

```
> ant -Dtext="John didn't run. " tokenization

tokenList=[John, didn, t, run]
whitespaceList=[,  , ',  , . ]
textTok=|John didn't run. |

John didn't run.
01234567890123456
0         1

   n start   end  whsp      token
   0     0     4    ||     |John|
   1     5     9   | |     |didn|
   2    10    11   |'|        |t|
   3    12    15   | |      |run|
lastWhitespace=|. |
```

We have added vertical bars around the text, tokens and whitespaces to highlight the final space. The spans here are only for the tokens. The whitespaces span the interstitial spaces between tokens, so their span can be computed as the end of the previous token to the start of the next token. The last whitespace ends at the underlying text length.

Because tokens are only defined to be contiguous sequences of letters, the apostrophe in *didn't* and the final period appear in the whitespace. The period only appears in the final whitespace.

Chapter 4

Suffix Arrays

Suffix arrays provide a convenient data structure for finding all of the repeated substrings of arbitrary length in texts. They also have applications to indexing. They are used for applications ranging from plagiarism detection to language modeling to indexing for phrase searches.

LingPipe provides suffix arrays that operate at either the character level or the token level. At the token level, there is a further class that manages collections of documents. Applications include plagiarism detection, language modeling, indexing

4.1 What is a Suffix Array?

A simple example will suffice to illustrate the basic workings of suffix arrays. Suppose we have the string *abracadabra*, presented here with the positions in the sequence numbered.

```
abracadabra
012345678901
0         1
```

A suffix array represents the suffixes of a string. For instance, the example string *abracadabra* has the following suffixes, given with their starting position in the string.

Pos	*Suffix*	*Pos*	*Suffix*	*Pos*	*Suffix*
0	*abracadabra*	4	*cadabra*	8	*bra*
1	*bracadabra*	5	*adabra*	9	*ra*
2	*racadabra*	6	*dabra*	10	*a*
3	*acadabra*	7	*abra*		

A suffix array is nothing more than the set of suffixes sorted. The suffixes are represented by their starting positions. For our running example, we have the following sorted list of suffixes.

Idx	Pos	Suffix
0	10	*a*
1	7	*abra*
2	0	*abracadabra*
3	3	*acadabra*
4	5	*adabra*
5	8	*bra*
6	1	*bracadabra*
7	4	*cadabra*
8	6	*dabra*
9	9	*ra*
10	2	*racadabra*

The suffix array itself is just the array of first-character positions for the suffixes. In our example, the suffix array is the following array of `int` values.

```
{ 10, 7, 0, 3, 5, 8, 1, 4, 6, 9 , 2 }
```

Note that the suffix itself may be retrieved given the underlying string and the position of the first character.

The utility of a suffix array is that it brings together suffixes that share common prefixes. For instance, it is easy to see that there are two suffixes that start with the sequence *bra*, one starting at position 8 and one starting at position 1. We can also see that there are five substrings starting with *a* and two starting with *abra*.

Every substring of our original string is representable as a prefix of a suffix. For instance, the substring *ca* spanning from position 7 (inclusive) to 9 (exclusive) is the two-character prefix of the suffix *cadabra* starting at position 7.

Thus we can easily find all the matching substrings in a string. For instance, if we want to find all the instances of strings of length three or more that occur two or more times in the text, we just scan down the suffix array, where we will find *abr* and *bra* both occur twice.

LingPipe provides a class for creating suffix arrays of characters from texts, suffix arrays of tokens from texts and a tokenizer factory, and suffix arrays of tokens from a collection of named documents and a tokenizer.

4.2 Character Suffix Arrays

LingPipe implements suffix arrays where the symbols are characters in the class `CharSuffixArray` in the package `com.aliasi.suffixarray`. We cover token suffix arrays in the next section).

4.2.1 Demo Code Walkthrough

We have provided a demo in class `CharSuffixArrayDemo` in this chapter's package. The demo class does little more than build a suffix array and then walk over it extracting the suffixes in order. The relevant code, after assigning the input argument to a string `text` is

```
CharSuffixArray csa = new CharSuffixArray(text);

for (int i = 0; i < csa.suffixArrayLength(); ++i) {
     int pos = csa.suffixArray(i);
     String suffix = csa.suffix(pos,Integer.MAX_VALUE);
```

The code uses the constructor without a specification on the maximum length. Then it walks through all of the positions, up to the length of the suffix array. From each, it extracts the position in the underlying string and the suffix running from that position to the end of the string. We have elided the print statements, as usual.

The second part of the demo extracts all of the repeated substrings. As such, it starts with a loop over the length of matching substrings, then extracts them inside.

```
for (int len = csa.suffixArrayLength() - 1; len > 1; --len) {
     List<int[]> matchSpans = csa.prefixMatches(len);
     if (matchSpans.size() == 0) continue;
     for (int[] matchSpan : matchSpans) {
         int first = matchSpan[0];
         int last = matchSpan[1];
         for (int i = first; i < last; ++i) {
             int pos = csa.suffixArray(i);
             String match = csa.suffix(pos,len);
```

We start from the maximum possible match, which is one less than the length of the string, and repeat for all lengths above 1. Within the loop, we call the `prefixMatches()` method on the suffix array with the specified length. This returns a list of spans represented as integer arrays. If the size is zero, there are no repeated substrings of the specified length, and we continue to the next iteration of the loop.

If there are matches, we iterate over the arrays of positions. For each such position, we extract the start and end position, assigning them to local variables for readability. Finally, we loop over the matching positions from first (inclusive) to last (exclusive), and for each match, pull out the position, then generate the string corresponding to the match. We do not show the print routines for these values, but show examples in the next section.

4.2.2 Running the Demo

We've provided an Ant target `char-demo` to run the demo. The property `text` specifies the text for which to build a suffix array. Using our running example, we have

```
> ant -Dtext=abracadabra char-demo
```

```
abracadabra
012345678901
0         1

Idx  Pos  Suffix          Idx  Pos  Suffix
  0   10  a                 6    1  bracadabra
  1    7  abra              7    4  cadabra
  2    0  abracadabra       8    6  dabra
  3    3  acadabra          9    9  ra
  4     5  adabra               10    2  racadabra
  5     8  bra
```

As before, the index is the (implicit) index into the suffix array, the position indicates the position in the original string, and the suffix is the range of characters from the position to the end of the original string.

After writing out the basic suffix array, the demo carries on with writing out the positiosn of all repeated substrings of length greater than one.

```
start=  7 len=  4  abra       start=  7 len=  2  ab
start=  0 len=  4  abra       start=  0 len=  2  ab

start=  7 len=  3  abr        start=  8 len=  2  br
start=  0 len=  3  abr        start=  1 len=  2  br

start=  8 len=  3  bra        start=  9 len=  2  ra
start=  1 len=  3  bra        start=  2 len=  2  ra
```

For instance, this indicates that the substring *abra* appeared twice, once starting at position 7 and once starting at position 0.

Although there are only pairs of matches here, the number can quickly grow with repetitive strings. The worst case situation for overlaps is with a sequence of the same character.

```
> ant -Dtext=yyyyy char-demo
yyyyy
012345

Idx   Pos  Suffix      Idx   Pos  Suffix
  0     4  y             3     1  yyyy
  1     3  yy            4     0  yyyyy
  2     2  yyy


start=  1 len=  4  yyyy       start=  3 len=  2  yy
start=  0 len=  4  yyyy       start=  2 len=  2  yy
                              start=  1 len=  2  yy
start=  2 len=  3  yyy        start=  0 len=  2  yy
start=  1 len=  3  yyy
start=  0 len=  3  yyy
```

In this example, the suffixes are merely sorted by length, and there are two matches of length four, three of length three, and four of length two.

4.3 Token Suffix Arrays

LingPipe's class `TokenSuffixArray`, in package `suffixarray`, supports token-based suffix arrays. Token suffix arrays work exactly the same way as character suffix arrays, only at the granularity of tokens.

4.3.1 Advantages of Token Arrays

There are three main advantages to using token-based suffix arrays. First, they are more efficient at representing texts and finding long substrings, as the suffix array itself will not be as large. The saving depends on the average token length. With the overhead of the tokenization object and the tokens themselves, a token suffix array may wind up being larger in memory than a character suffix array. But it should be faster to construct and faster to find long matches.

The second gain is in understandability of matches. Usually in language-processing applications, we only care about matches including all or none of a token. Substring matching involving partial tokens is rarely useful at an application level.

The third gain is in the ability to relax the comparison criteria. For instance, tokenizers can case normalize, stop list, dispose of whitespace and punctuation, and so on. This allows greater control over the matching.

4.3.2 Token-Based Array Demo

The class `TokenSuffixArrayDemo` in this chapter's package, `suffixarray`, illustrates the basic usage pattern for token suffix arrays.

Code Walk Through

The command-line input is set to the `String` variable `text`, and then a suffix array based on tokens is constructed as follows.

```
TokenizerFactory tf
     = IndoEuropeanTokenizerFactory.INSTANCE;
tf = new LowerCaseTokenizerFactory(tf);
Tokenization tok = new Tokenization(text,tf);
TokenSuffixArray tsa = new TokenSuffixArray(tok);
```

We first access the singleton tokenizer factory for Indo-European languages, then wrap it in a lower-case filter that downcases all of the text in the tokens (see Chapter 3 for more on tokenizers and tokenizer factories). We then construct the tokenization itself using the text and the tokenizer factory. The `Tokenization` encapsulates the raw text, the list of tokens extracted, and each token's start and end position in the raw text. This allows us to map the tokens back to their positions in the underlying text (see Section 3.9.2 for details of the `Tokenization` class). Finally, we construct the token suffix array itself using just the tokenization, which defaults to unbounded suffix lengths and the default document separator (we cover document separators in Section 4.4).

We see the tokenization in action in the next section of the code, which accesses the tokenization from the suffix array and prints out its tokens and their positions.

```
Tokenization tokenization = tsa.tokenization();
for (int i = 0; i < tokenization.numTokens(); ++i) {
     String token = tokenization.token(i);
```

```
    int start = tokenization.tokenStart(i);
    int end = tokenization.tokenEnd(i);
```

Next, we iterate through the suffixes themselves.

```
for (int i = 0; i < tsa.suffixArrayLength(); ++i) {
    int suffixArrayI = tsa.suffixArray(i);
    String suffix = tsa.substring(i,Integer.MAX_VALUE);
```

Here we consider all positions up to the length of the suffix array, pulling out the value of the suffix array, which is an index into the tokens in the tokenization.

The final section of the demo finds subsequences of tokens of a specified length that match in the text.

```
for (int len = 20; --len > 0; ) {
    List<int[]> prefixMatches = tsa.prefixMatches(len);
    for (int[] match : prefixMatches) {
        for (int j = match[0]; j < match[1]; ++j) {
            int textPos = tsa.suffixArray(j);
            String suffix = tsa.substring(j,len);
```

As with the character suffix array, we consider different lengths and for each length, find all the matches using the `prefixMatches()` method on token suffix arrays. Each match is an array of integers recording the start (inclusive) and end (exlcusive) position in the suffix array. Note that these positions are contiguous in the suffix array because the suffix array is sorted; this is why we can just loop incrementing the variable `j`. Finally, given a position in the suffix array, we find the position in the text and the corresponding suffix. As with our other demos, we've elided the print statements.

Running the Demo

The Ant target `token-demo` runs the demo, with property `text` providing the text input.

```
> ant -Dtext="John ran HOME and Mary ran home." token-demo
TEXT:
John ran HOME and Mary ran home.
012345678901234567890123456789012
0         1         2         3

TOKENS
  0 (  0,  4) john        4 ( 18, 22) mary
  1 (  5,  8) ran         5 ( 23, 26) ran
  2 (  9, 13) home        6 ( 27, 31) home
  3 ( 14, 17) and         7 ( 31, 32) .
```

The first part of the output just reports the tokens and their start (inclusive) and end (exclusive) positions in the text.

After this, we see the suffix array itself, with the index into the basic list of tokens and the suffix itself.

```
SUFFIX ARRAY (idx,array,suffix)
    0   7 .
    1   3 and Mary ran home.
    2   6 home.
    3   2 HOME and Mary ran home.
    4   0 John ran HOME and Mary ran home.
    5   4 Mary ran home.
    6   5 ran home.
    7   1 ran HOME and Mary ran home.
```

Note that the sorting is on the tokens, not the suffixes, so that *HOME*, *John*, and *Mary* sort as if they were lower case. Finally, we see the matching substrings, this time at a token level.

```
MATCHING SUBSTRINGS
  len   sa  txt  suffix         len   sa  txt  suffix
    2    6    5  ran home         1    6    5  ran
    2    7    1  ran HOME         1    7    1  ran

    1    2    6  home
    1    3    2  HOME
```

Note that the comparison is also done by token, so taht *ran home* and *ran HOME* are considered a match. This time, we've also printed the length 1 matches.

4.4 Document Collections as Suffix Arrays

The third class implemented in LingPipe's `suffixarray` package is for managing collections of documents in a suffix array. This class is based on an underlying token suffix array whose text is derived by concatenating all the documents together with a distinguished separator. This process is all handled internally in the document suffix array.

4.4.1 Demo

The document suffix array demo is in the file `DocSuffixArrayDemo` in this chapter's package. The demo will walk over a directory recursively and add the text of every plain file to the document collection. It will then index these documents in a suffix array so as to support searching for shared substrings.

Code Walkthrough

A document suffix array is constructed from a mapping from string-based document identifiers to text strings. We start with a path assigned to file variable `dir` and an integer match length `len`. We create a map to hold the identifier-to-document mapping and then call the static helper method `addFiles()` to create it from the directory.

```
Map<String,String> idToDocMap = new HashMap<String,String>();
addFiles(dir,idToDocMap);
```

The addFiles() function is defined recursively in the obvious way, using the path to the file as the file identifier and the character content of the file as its text.

```
static void addFiles(File path, Map<String,String> idToDocMap)
     throws IOException {

     if (path.isDirectory()) {
         for (File subpath : path.listFiles())
             addFiles(subpath, idToDocMap);
     } else if (path.isFile()) {
         String fileName = path.toString();
         String text = Files.readFromFile(path,"ASCII");
         text = text.replaceAll("\\s+"," "); // norm whitespace
         idToDocMap.put(fileName,text);
     }
}
```

The result is that we have a map from document identifier strings to the text of the document.

Next, we create a tokenizer factory that lowercases tokens extracted by the Indo-European tokenizer and filters out any non-word tokens (where words are defined as containing only ASCII alpha-numeric characters).

```
TokenizerFactory tf = IndoEuropeanTokenizerFactory.INSTANCE;
tf = new LowerCaseTokenizerFactory(tf);
tf = new RegExFilteredTokenizerFactory(tf,Pattern.compile("\\w+▷
▷"));
```

With the document mapping and tokenizer factory, we are ready to build a document suffix array.

```
String boundaryToken = "eeooff";
int maxSuffixLength = Integer.MAX_VALUE;
DocumentTokenSuffixArray dtsa
     = new DocumentTokenSuffixArray(idToDocMap, tf,
                                    maxSuffixLength,
                                    boundaryToken);
```

First, we set a boundary token of *eeooff*. Note that this boundary token must tokenize to itself in the tokenizer, or the constructor will throw an exception. We also set a maximum suffix length to be the maximum possible length.

Once we've constructed the document suffix array, we pull out the underlying token suffix array using the method suffixArray(), then find and print the matches.

```
TokenSuffixArray tsa = dtsa.suffixArray();

List<int[]> prefixMatches = tsa.prefixMatches(len);
```

```
for (int[] match : prefixMatches) {
     for (int j = match[0]; j < match[1]; ++j) {
         String matchedText = tsa.substring(j,len);
         int textPos = tsa.suffixArray(j);
         String docId = dtsa.textPositionToDocId(textPos);
```

The same method, `prefixMatches(int)`, is used to pull out the matches. The primary utility of the document suffix array is to manage the document identifiers and map matches back to them, as seen in the following code that loops over the matches.

As with the token-based suffix arrays, we get a contiguous range of indexes into the suffix array, here assigned to integer value `j`. We then pull the text that was matched out just as evore, using the `substring()` method on token suffix arrays. We also find the position of the token in the underlying text using the method `suffixArray()`, which returns the value of the suffix array for its argument. We then use this position to pull out the identifier for the document using the document suffix array method `textPositionToDocId()`. As usual, we have elided the print statements and boilerplate.

Running the Demo

First, we need to gather data. We have used the test section of the 20 newsgroups corpus for that purpose; see Section D.2 for download instructions and a description of the corpus. The documents are newsgroup postings, so they contain a large degree of overlapping text because of the heavy use of quoting previous messages. We assume the data have been unpacked into the directory `data/20news/20news-bydate-test`, which we have placed in the top-level for the book.

The Ant target `doc-demo` runs the demo, using the properties `len` for the required match length and `dir` for a path to the directory containing the documents.

```
>        ant -Dlen=100 -Ddir=../../data/20news/20news-bydate-test
doc-demo
```

```
Extracting files from path=C:\lpb\data\20news\20news-bydate-test
Match length=100
# of docs=7532
# of tokens=2298193
```

There are roughly 7500 documents and 2.3 million tokens, and it takes a little less than 30 seconds and 2GB of memory to build the array on my aging workstation using 64-bit Java 1.6.

It then starts dumping out matches, which are prohibitively long, so we truncate their tails here for readability:

```
rec.sport.hockey\53942  0 0 0.00 0 0 0 0 1.000 WAS Byron Dafoe ...
rec.sport.hockey\54274  0 0 0.00 0 0 0 0 1.000 WAS Byron Dafoe ...
...
```

These are the first two results, which are reports of player statistics in two different posts on the `rec.sport.hockey` newsgroup. The text printed out is the text spanned by the matching tokens in the original document. Given the various access methods, it'd also be possible to retrieve the tokens that actually matched.

4.5 Implementation Details

Suffix arrays are represented internally very directly as an array of `int` values. During construction, an array of `Integer` positions is created and sorted using a `Comparator` for suffixes. This boxing of `int` values as `Integer` positions requires an object per position in the suffix array and also an `int` at the point the array is being written out, both of which are relatively costly. [1]

Thus the overall construction time is no worse than $\mathcal{O}(n \log n)$ comparisons, where n is the length of the string. The current implementation uses a very naive comparator that may require up to $\mathcal{O}(n)$ operations to compare suffixes where `n` is the number of matching characters in the prefixes of the suffixes. Texts with long repeated substrings, like our earlier example *yyyyyy*, present worst-case scenarios.[2]

[1] In the future, we may replace this crude implementation with a more efficient version that is able to sort the `int` values in a suffix array directly.

[2] A more efficient comparator implementation would dynamically track the number of matching prefix characters in its divide-and-conquer step — merge sort is particularly easy to understand in this way. We would then start all comparisons after the matching characters in the subsegment of data being sorted.

Chapter 5

Symbol Tables

Strings are unwieldy objects — they consume substantial memory and are slow to compare, making them slow for sorting and use as hash keys. In many contexts, we don't care about the string itself *per se*, but only its identity. For instance, document search indexes like Lucene consider only the identity of a term. Naive Bayes classifiers, token-based language models and clusterers like K-means or latent Dirichlet allocation do not care about the identity of tokens either. Even compilers for languages like Java do not care about the names of variables, just their identity.

In applications where the identity of tokens is not important, the traditional approach is to use a symbol table. A symbol table provides integer-based representations of strings, supporting mappings from integer identifiers to strings and from strings to identifiers. The usage pattern is to add symbols to a symbol table, then reason with their integer representations. So many of LingPipe's classes operate either explicitly or implicitly with symbol tables, that we're breaking them out into their own chapter.

5.1 The `SymbolTable` Interface

The interface `SymbolTable` in `com.aliasi.symbol` provides a bidirectional mapping between string-based symbols and numerical identifiers.

5.1.1 Querying a Symbol Table

The two key methods are `symbolToID(String)`, which returns the integer identifier for a string, or -1 if the symbol is not in the symbol table. The inverse method `idToSymbol(int)` returns the string symbol for an identifier; it throws an `IndexOutOfBoundsException` if the identifier is not valid.

The method `numSymbols()` returns the total number of symbols stored in the table as an integer (hence the upper bound on the size of a symbol table is `Integer.MAX_VALUE`).

Modifying a Symbol Table

Like the `Collection` interface in `java.util`, LingPipe's symbol table interface specifies optional methods for modifying the symbol table. In concrete implementations of `SymbolTable`, these methods either modify the symbol table or throw an `UnsupportedOperationException`.

The method `clear()` removes all the entires in the symbol table. The method `removeSymbol(String)` removes the specified symbol from the symbol table, returning its former identifier or -1 if it was not in the table.

The method `getOrAddSymbol(String)` returns the identifier for the specified string, adding it to the symbol table if necessary.

5.2 The `MapSymbolTable` Class

Of the two concrete implementations of `SymbolTable` provided in LingPipe, the map-based table is the most flexible.

5.2.1 Demo: Basic Symbol Table Operations

The class `SymbolTableDemo` contains a demo of the basic symbol table operations using a `MapSymbolTable`. The `main()` method contains the following statements (followed by prints).

```
MapSymbolTable st = new MapSymbolTable();

int id1 = st.getOrAddSymbol("foo");
int id2 = st.getOrAddSymbol("bar");
int id3 = st.getOrAddSymbol("foo");

int id4 = st.symbolToID("foo");
int id5 = st.symbolToID("bazz");

String s1 = st.idToSymbol(0);
String s2 = st.idToSymbol(1);

int n = st.numSymbols();
```

The first statement constructs a new map symbol table. Then three symbols are added to it using the `getOrAddSymbol()` method, with the resulting symbol being assigned to local variables. Note that `"foo"` is added to the table twice. This method adds the symbol if it doesn't exist, so the results returned are always non-negative.

Next, we use the `symbolToID()` methods to get identifiers and also assign them to local variables. This method does not add the symbol if it doesn't already exist. Finally, we retrieve the number of symbols.

The Ant target `symbol-table-demo` runs the command.

```
> ant symbol-table-demo

id1=0           s1=foo
id2=1           s2=bar
```

```
id3=0           n=2
id4=0
id5=-1
```

Note that `id1` and `id3` are the same, getting the identifier 0 for string `"foo"`. This is also the value returned by the `symboltoID()` method, as shown by the value of `id4`. Becase we used `symbolToID("bazz")` rather than `getOrAddSymbol("bazz")`, the value assigned to `id5` is -1, the special value for symbols not in the symbol table.

The `idToSymbol()` method invert the mappings as shown by their results. Finally, note there are 2 symbols in the symbol table when we are done, `"foo"` and `"bar"`.

5.2.2 Demo: Removing Symbols

The class `RemoveSymbolsDemo` shows how the map symbol table can be used to modify a symbol table. Its `main()` method is

```
st.getOrAddSymbol("foo");
st.getOrAddSymbol("bar");
int n1 = st.numSymbols();

st.removeSymbol("foo");
int n2 = st.numSymbols();

st.clear();
int n3 = st.numSymbols();
```

We first add the symbols `"foo"` and `"bar"`, then query the number of symbols. Then we remove `"foo"` using the method `removeSymbol()` and query the numbe of symbols again. Finally, we clear the symbol table of all symbols using the method `clear()`, and query a final time.

We can run the demo with the Ant target `remove-symbol-demo`,

```
> ant remove-symbol-demo

n1=2     n2=1     n3=0
```

By removing symbols, we may create holes in the run of identifiers. For instance, the identifier 17 may have a symbol but identifier 16 may not. This needs to be kept in mind for printing or enumerating the symbols, which is why there are symbol and identifier set views in a map symbol table (see below).

5.2.3 Additional `MapSymbolTable` Methods

The implementation `MapSymbolTable` provides several methods not found in the `SymbolTable` interface.

Boxed Methods

The map symbol table duplicates the primitive-based SymbolTable methods with methods accepting and returning Integer objects. This avoids round trip automatic unboxing and boxing when integrating symbol tables with collections in java.util.

The methods getOrAddSymbolInteger(String), symbolToIDInteger(String) return integer objects rather than primitives. The method idToSymbol(Integer) returns the symbol corresponding to the integer object identifier.

Symbol and Identifier Set Views

The method symbolSet() returns the set of symbols stored in the table as a set of strings, Set<String>. The method idSet() returns the set of IDs stored in the table as a set of integer objects, Set<Integer>.

These sets are views of the underlying data and as such track the data in the symbol table from which they originated. As the symbol table changes, so do the sets of symbols and sets of identifiers.

These sets are immutable. All of the collection methods that might modify their contents will throw unsupported operation exceptions.

Providing immutable views of underlying objects is a common interface idiom. It allows the object to maintain the consistency of the two sets while letting the sets reflect changes in the underlying symbol table.

We provide a demo in SymbolTableViewsDemo. The main() method uses the following code.

```
MapSymbolTable st = new MapSymbolTable();
Set<String> symbolSet = st.symbolSet();
Set<Integer> idSet = st.idSet();
System.out.println("symbolSet=" + symbolSet
                   + " idSet=" + idSet);

st.getOrAddSymbol("foo");
st.getOrAddSymbol("bar");
System.out.println("symbolSet=" + symbolSet
                   + " idSet=" + idSet);

st.removeSymbol("foo");
System.out.println("symbolSet=" + symbolSet
                   + " idSet=" + idSet);
```

The two sets are assigned as soon as the symbol table is constructed. We then print out the set of symbols and identifiers. After that, we add the symbols "foo" and "bar" and print again. Finally, we remove "foo" and print for the last time.

The Ant target symbol-table-views runs the demo,

```
> ant symbol-table-views-demo

symbolSet=[] idSet=[]
```

```
symbolSet=[foo, bar] idSet=[0, 1]
symbolSet=[bar] idSet=[1]
```

As advertised, the values track the underlying symbol table's values.

Unmodifiable Symbol Table Views

In many cases in LingPipe, models will guard their symbol tables closely. Rather than returning their actual symbol table, they return an unmodifiable view of the entire table. The method `unmodifiableView(SymbolTable)` returns a symbol table that tracks the specified symbol table, but does not support any of the modification methods.

This method employs the same pattern as the generic static utility method `<T>unmodifiableSet(Set<? extends T>)` in the utility class `Collections` from the built-in package `java.util`; it returns a value of type `Set<T>` which tracks the specified set, but may not be modified. This collection utility is used under the hood to implement the symbol and identifier set views discussed in the previous section.

Thread Safety

Map symbol tables are not thread safe. They require read-write synchronization, where all the methods that may modify the underlying table are considered writers.

Serialization

Map symbol tables are serializable. They deserialize to a `MapSymbolTable` with the same behavior as the one serialized. Specifically, the identifiers do no change.

5.3 The `SymbolTableCompiler` Class

The `SymbolTableCompiler` implementation of `SymbolTable` is a legacy class used in some of the earlier LingPipe models. We could have gotten away without the entire `com.aliasi.symbol` package and just dropped `MapSymbolTable` into `com.aliasi.util`. Instead, we abstracted an interface and added a parallel implementation in order to maintain backward compatibility at the model level.

Once a symbol is added, it can't be removed. Symbol table compilers do not support the `remove()` or `clear()` methods; these methods throw unsupported operation exceptions.

5.3.1 Adding Symbols

The symbol table compiler implements a very unwieldy usage pattern. It begins with a no-argument constructor. The usual `getOrAddSymbol(String)` method from the `SymbolTable` interface will throw an unsupported operation exception.

Instead, the method addSymbol(String) is used to add symbols to the table, returning true if the symbol hasn't previously been added.

The really broken part about this class is that it doesn't actually implement SymbolTable properly until after its been compiled.

5.3.2 Compilation

A symbol table compiler implements LingPipe's Compilable interface (for more information, see the section in the I/O chapter of the companion volume, *Text Processing in Java*).

5.3.3 Demo: Symbol Table Compiler

The demo class SymbolTableCompilerDemo implements a main() method demonstrating the usage of a symbol table compiler. The method begins by setting up the compiler and adding some symbols

```
SymbolTableCompiler stc = new SymbolTableCompiler();
stc.addSymbol("foo");
stc.addSymbol("bar");

int n1 = stc.symbolToID("foo");
int n2 = stc.symbolToID("bar");
int n3 = stc.symbolToID("bazz");
```

It sets some variables that result from attempting to get the symbols again at this point.

Next, we compile the symbol table using the static utility method compile() from LingPipe's AbstractExternalizable class. We use the original symbol table then the compiled version to convert symbols to identifiers.

The Ant target symbol-table-compiler-demo runs the command.

```
> ant symbol-table-compiler-demo

n1=-1     n2=-1     n3=-1
n4=1     n5=0     n6=-1
n7=1     n8=0     n9=-1

st.getClass()=class com.aliasi.symbol.CompiledSymbolTable
```

The symbols are all unknown before compilation, returning -1 as their identifier. After compilation, both the compiler and the compiled version agree on symbol identifiers. We also print the class for the result, which is the package-protected class CompiledSymbolTable. This class does implement SymbolTable, but the get-or-add method is not supported.

One nice property of the class is that we are guaranteed to have identifiers numbered from 0 to the number of symbols minus 1. We are also guaranteed to get symbols stored in alphabetical order. Note that the symbol "bar" is assigned identifier 0 even though it was added after the symbol "foo".

Be careful with this class. If more symbols are added, then the table is compiled again, identifiers will change. This is because it relies on an underlying sorted array of symbols to store the symbols and look up their identifiers.

5.3.4 Static Factory Method

The static utility method `asSymbolTable(String[])` constructs a symbol table from an array of symbols. It will throw an illegal argument exception if there are duplicates in the symbol table. Perhaps a `SortedSet` would've been a better interface here.

Chapter 6

Character Language Models

A *language model* is a statistical construct that assigns probabilities to strings of symbols. Traditionally, these strings are either sequences of bytes, sequences of characters, or sequences of tokens. LingPipe provides language-model implementations for sequences of characters and sequences of tokens (see Chapter 3 for more on tokenization in LingPipe). In this chapter, we introduce the basic interfaces and character language models. In Chapter 7, we introduce token language models.

The bytes, characters or tokens modeled by language models may represent arbitrary sequences of symbols. With token-based models, the symbol set need not even be finite. For instance, we can model XML documents as sequences of characters, proteins as sequences of amino acids, or texts as sequences of tokens. In this chapter, we are going to focus on texts represented as sequences of Unicode characters. the text itself may come from anywhere, including e-mail or instant messages, blogs, newspapers, fiction novels, emergency room discharge summaries, people names or names of companies, etc.

The basic operation of language models in LingPipe is to provide them training data consisting of a sequence of texts. After they are trained, language models may be used to estimate the probability of texts that were not in the training data.

6.1 Applications of Language Models

LingPipe uses character language models as the basis of implementations of several interfaces, including classifiers, taggers, chunkers, and spelling correctors. There are further applications for token language models discussed in Chapter 7. In all cases, these applications use one or more language models to estimate different kinds of text. For instance, in language-model based classification, there is a language model for each category, and texts are classified into the category which assigns them the highest probability. In chunkers, separate language models might be constructed for person names, company names, place names, and for text that's not part of a name; these are then combined to analyze new text for names of entities. For spelling correction, a single language model characterizes the correct spellings and is used to weigh alternative corrections against each other.

Yet another application of language models is in characterizing languages or sublanguages. For instance, we can compare the entropy of news stories in the *New York Times* with biomedical abstracts in MEDLINE, the surprising result being that MEDLINE is more predictable at the text level than the *Times*). We can also compare the entropy of Chinese and English, measuring the average information content of a Chinese character (of which there are tens of thousands) with the information content of an English character (of which there are dozens).

6.1.1 Applications Beyond LingPipe

There are many uses of language models that are not supported by LingPipe. Yet the language modeling components are the same and LingPipe supports appropriate interfaces.

One of the major applications of language models is in speech recognition, where they play the same role as the language model in spelling correction.

Another application of language models is compression. Using a technique such as *arithmetic coding*, language models provide state-of-the-art compression. Although LingPipe does not implement an arithmetic coder for compression, we will discuss the connection to compression when we discuss language model evaluation. It turns out there is a deep connection between predictability expressed probabilistically and compressibility.

6.2 The Basics of N-Gram Language Models

LingPipe's character- and token-based language models are based on n*-grams*. An n-gram is nothing more than a sequence of n symbols. In this chapter, we consider character-based n-grams; in Chapter 7, we consider tokens as symbols. For instance, the triple (*a*, *b*, *c*) is a character trigram (3-gram), whereas the pair (*in*, *Brooklyn*) is a token bigram (2-gram).

The basis of an n-gram language model is storing counts for n-grams seen in a training corpus of text. These are then used to infer the probability of a new sequence of symbols. They key technical feature of n-gram models is that they model the probability of a character in a sequence based only on the previous $n-1$ characters. Models such as these with finite memories are called *Markov chains*.

The chain rule allows us to model the probability of a sequence of characters based on a model of a single character following a sequence of characters before it. For instance, with the character sequence *tooth* using 3-grams (trigrams), we decompose the total probability of a sequence into the product of a sequence of character probabilities, with

$$p(\mathit{tooth}) = p(t) \times p(o|t) \times p(o|\mathit{to}) \times p(t|\mathit{oo}) \times p(h|\mathit{ot}). \tag{6.1}$$

To prevent the possibilty of underflow from multiplying long sequences of numbers less than 1, we almost always work on the log scale. This also has the pleasant property of converting multiplication to addition, yielding

$$\log p(tooth) = \log p(t) + \log p(o|t) + \log p(o|to) + \log p(t|oo) + \log p(h|ot). \quad (6.2)$$

6.3 Character-Level Language Models and Unicode

LingPipe's character-based language models are models of sequences of Java `char` values, which represent UTF-16 byte pairs (see the section on byte representations for primitive types in the introductory Java section of the companion volume, *Text Processing in Java*. When LingPipe's language models were coded, every Unicode code point could be coded in a single UTF-16 byte pair. As we discussed, this is enough to cover Unicode's basic multilingual plane (BMP), which includes most characters for most languages.[1]

6.4 Language Model Interfaces

There is a rich set of interfaces for language models and related support classes in the LingPipe package `com.aliasi.lm`. The basic interface is `com.aliasi.lm.LanguageModel` interface, and other interfaces are nested within `LanguageModel`.

6.4.1 Predictive: `LanguageModel`

The top-level language model interface, `LanguageModel`, defines methods for estimating probabilities of sequence of characters, which is also known as prediction. It defines two methods for estimating the log (base 2) probability of a sequence of characters, one based on character array slices and one based on the character sequence interface.

```
double log2Estimate(char[] cs, int start, int end);
double log2Estimate(CharSequence cs);
```

The first method uses an indexed character array to define the slice of characters from `cs[start]` to `cs[end-1]`. For most methods, the character slice method is more efficient and the character-sequence method is implemented by first converting the sequence to an array of characters.

The reason estimates are returned as logarithms is that we would otherwise run into underflow. If characters have an average probabilty of 0.25 (this is roughly the case for open-domain English text), the probability of a sequence of

[1]As discussed in the section on character primitives in the companion volume, *Text Processing in Java*, Unicode 4.1, introduced characters with code points beyond the BMP. To code such characters requires two 16-bit `char` values, known as a surrogate pair. For the purposes of LingPipe's language models, characters are 16 bit UTF-16 values and a code point beyond the BMP is considered two characters. What this will mean in practice is that LingPipe's language models will assign non-zero probability to sequences of `char` values that do not correspond to legal UTF-16 encodings, and thus don't correspond to Unicode strings. In the case of spelling correction, we have to be careful not to suggest corrections that are illegal sequences of `char` values. In other cases, we just lose some modeling efficiency (in the statistical sense of efficiency, not the run-time sense).

600 tokens has a $0.25^{600} = 2^{-1200}$ probabilty, which is too small to fit into a Java double-precision floating point value (the minimum value for which is 2^{-1074}; see the section on Java primitive types in the companion volume, *Text Processing in Java*).

6.4.2 Trainable: `LanguageModel.Dynamic`

Language models that may be trained by supplying example text implement the interface `LanguageModel.Dynamic`. This interface extends the corpus interface `ObjectHandler<CharSequence>`, which specifies the method

```
void handle(CharSequence cs);
```

The character sequence is considered as training data and used to fit the model parameters. See Section 2.1 for more information on `ObjectHandler` and how it fits into corpus-based usage patterns.

There are four other methods named `train()`, the first two of which simply allow training with character sequences.

```
void train(CharSequence cs);
void train(CharSequence cs, int count);
```

The first method is a legacy method duplicating the functionality of `handle()`. The second method takes an integer count value which determines the number of times the character sequence is used for training. Although implemented more efficiently, a training call with a count is equivalent to calling the basic training method the count number of times.

The last two methods just repeat the character sequence methods with character slices.

6.4.3 Conditional Probabilities: `LanguageModel.Conditional`

Conditional language models predict the probability of a character given the previous characters. The interface `LanguageModel.Conditional` specifies the method

```
double log2ConditionalEstimate(CharSequence);
```

This method returns the probabilty of the last character in the sequence given the initial characters in the sequence. For instance, `log2ConditionalEstimate("abcd")` returns the probability of the character *d* following the sequence of characters *abc*.

There is also a method working on character slices. It doesn't make sense to call this method with an empty sequence.

The character language models all support this method, but it is not supported by the token-based language models, which work token by token, not character by character.

6.4.4 Sequence versus Process Language Models

Language models are divided into two types, sequence models and process models. Conceptually, sequence models distribute probability to all strings of all lengths. As such, they may be used to compare the probabilities of words of different lengths. Process models form the basis of LingPipe's taggers and chunkers.

In contrast, process models distribute probability only among strings of a given length. Process models may only be used to compare the probabilities of two strings of the same length. LingPipe implements sequence models on top of process models as described below. Process models also form the basis of spelling correction and like sequence models, may be used as the basis of classifiers.

The distinction between sequence and process models is implemented using *marker interfaces*. A marker interface does not specify any method signatures. The most well-known marker interface in Java itself is `java.io.Serializable`, though `java.util.EventListener` is also a marker interface.

Bounded: `LanguageModel.Sequence`

The interface `LanguageModel.Sequence` is just a marker interface, meaning that it does not specify any new methods other than that inherited from its super-interface, `LanguageModel`.

Sequence language-model implementations are normalized so that the sum of the probability of all strings of all lengths is one.[2] In symbols, this amounts to

$$\sum_{n \geq 0} \sum_{\texttt{cs.length()} == \texttt{n}} 2^{\texttt{log2Prob(cs)}} = 1. \tag{6.3}$$

Unbounded: `LanguageModel.Process`

The second type of language model is a process language model. Such a language model implements the marker interface `LanguageModel.Process`. A process language model treats a sequence of characters as being generated as a kind of random process that generates a character at a time without a distinct beginning or end.

Because they don't model the beginning or end of a string, process language models are normalized by length so that the sum of the probabilities of all strings of a given length is one. In symbols, for every `n`, we have

$$\sum_{\texttt{cs.length()} == \texttt{n}} 2^{\texttt{log2Prob(cs)}} = 1. \tag{6.4}$$

[2]Because it's just a marker interface, the requirement is conventional rather than expressed in Java's type language. This is similar to interface requirements on collections in the standard `java.util` library.

6.5 Process Character Language Models

As we saw in the last section, language models have a particularly simple interface. We just give them text to train and then they can model new text.

The simplest language model implementation supplied by LingPipe is `NGramProcessLM`, in package `com.aliasi.lm`. This class implements several interfaces, including several of the basic language model interfaces, `LanguageModel`, the unbounded interface, `LanguageModel.Process`, the trainable interface `LanguageModel.Dynamic`, and conditional prediction interface `LanguageModel.Conditional` (see Section 6.4 for descriptions of these interfaces).

In addition to the LM interfaces, the `NGramProcessLM` class also implements `ObjectHandler<CharSequence>` (see Section 2.1 for a description of the handler interface). This allows language models to be integrated into the general parser/handler/corpus interfaces (see Chapter 2 for more details).

Furthermore, process character LMs are both serializable (see the sections on object I/O in the I/O chapter in the companion volume, *Text Processing in Java*). As we will see later in this section, compilation produces a very efficient, but no longer dynamic version of the language model, whereas serialization will allow it to be written out and read back in.

6.5.1 Process Character Language Model Demo

Perhaps the simplest way to get a feel for how language models work is to code one up. We provide an example in the class `ProcessLmDemo` in this chapter's package, `com.lingpipe.book.charlm`.

Code Walkthrough

As with all of our demos, it's set up as a simple `main()` method that reads its arguments from the command line. Specifically, we need an integer n-gram length (`ngram`), a character sequence on which to train (`textTrain`) and a character sequence on which to test (`textTest`), which are supplied as the first three arguments. The training and evaluation are then almost trivial.

```
NGramProcessLM lm = new NGramProcessLM(ngram);
lm.handle(textTrain);
double log2Prob = lm.log2Estimate(textTest);
```

We start by creating a new instance of an `NGramProcessLM` supplying the n-gram length to the constructor. We then take the training text string and pass it to the language model's `handle(CharSequence)` method, which adds the text to the training data. Finally, we compute the log (base 2) of the probaiblity of the test text using the language model's `log2Estimate(CharSequence)` method.

Running the Demo

The Ant target `process-demo` runs the command, using the property `ngram` for n-gram length, the property `text.train` for the training text, and the property `text.test` for the test text. It is thus run as follows.

```
> ant -Dngram=5 -Dtext.train="abracadabra" -Dtext.test="candelabra"
process-demo
```

```
ngram=5    train=|abracadabra|    test=|candelabra|
log2 p(test|train)=-69.693
```

The command-line arguments are first printed (with vertical bars to denote the boundaries of the strings) and then the log probability of the test sequence given the training data is shown.

One way of thinking of language models is as measuring a kind of text similarity. The more similar the test text is to the training text, the higher its probability.

If the test text is very predictable given the training text, the log2 probability will be higher (higher log probabilities have smaller absolute values, because logarithms of values in $[0, 1]$ are always negative). For instance, consider the following case.

```
> ant -Dngram=2 -Dtext.train="ababababab" -Dtext.test="ababab"
process-demo
```

```
ngram=2    train=|ababababab|    test=|ababab|
log2 p(test|train)=-3.060
```

6.5.2 Length Effects

Shorter texts will usually have lower probabilities in a process language model. That's because the sum of probabilities of all strings of a fixed length sum to 1, such as all seven-character strings. There are many many more seven-character strings than six-character strings. The number of strings of a given length grows exponentially (as long as the alphabet contains more than a single character). It's important to keep in mind that we shouldn't be comparing probabilities of strings of different lengths using a process language model. Conveniently, LingPipe's built-in applications of language models ensure that the probabilities are used appropriately.

As an example of the length effect, consider the same training text as above with a test text that's a subsequence of the test text above:

```
> ant -Dngram=2 -Dtext.train="ababababab" -Dtext.test="abab"
process-demo
```

```
ngram=2    train=|ababababab|    test=|abab|
log2 p(test|train)=-2.419
```

Note that -2.419 is greater than -3.060. In linear probability terms, that's $2^{-2.419} = 0.225$, whereas $2^{-3.060} = 0.120$. In other words, the model, after training, says that *abab* is almost twice as likely as *ababab*.

For process language models, which don't model boundaries, prefixes of a string always have higher probabilities. This is not necessarily the case for arbitrary substrings, nor will it be the case for the boundary language models we consider in the next section.

Characters not in the Training Data

A language model will assign probabilities to any texts of any length. The characters don't need to have been seen in the training data. For instance, consider our first example above, where the characters *e*, *n*, and *l* were new to the training data. Characters that were not seen in the training data tend to have fairly low probabilities. For instance, consider

```
> ant -Dngram=2 -Dtext.train="abababababab" -Dtext.test="xyxy"
process-demo
```

```
ngram=2    train=|ababababab|    test=|xyxy|
log2 p(test|train)=-71.229
```

Again, it's clear why we use the log scale — 2^{-71} is a very low probability indeed.

Effect of n-Gram Length

The n-gram length is the most important of the tuning parameters; we consider the rest of which we turn to in the next section. A longer n-gram will provide a tighter model of the training text than shorter n-grams. The longer the n-gram, the higher the probability assigned to the training text itself as well as to very similar text (again measuring by n-gram frequency). For instance, consider

```
> ant -Dngram=2 -Dtext.train="abcdef" -Dtext.test="abcdef"
process-demo
```

```
ngram=2    train=|abcdef|    test=|abcdef|
log2 p(test|train)=-11.334
```

and

```
> ant -Dngram=3 -Dtext.train="abcdef" -Dtext.test="abcdef"
process-demo
```

```
ngram=3    train=|abcdef|    test=|abcdef|
log2 p(test|train)=-10.884
```

The improvement is relatively small. If we had repeated the training text many times or if it had been longer, the effect would be stronger. Here, if you actually go up to 4-grams, the probability estimate gets worse. The reason for this is the default smoothing parameters, which we discuss in Section 6.7.

One thing to beware of is that increasing the n-gram size tends to have a dramatic impact on the amount of memory required. This is apparent when you consider that for basic English, there are only a few dozen characters and thus only a few dozen unigrams (1-grams). But there are thousands of bigrams (2-grams), and millions of 5-grams and 6-grams. Even with 40 characters, there are over 2.5 million potential 4-grams. Luckily, natural language text is fairly predictable, so the set of n-grams actually observed tends to grow more slowly than this worst case may lead you to believe. Furthermore, the number of n-grams in any given text is always bounded by the length of the text, because each position contributes at most one n-gram of a given length.

6.6 Sequence Character Language Models

In this section, we describe the second kind of character language model available in LingPipe, the sequence models. Sequence models are applied when there are either boundary or length effects. They provide a properly normalized probability model to compare the probabilities of strings of different lengths; the process models only allow comparison of probabilities of strings of the same length. The sequence models in LingPipe are built on top of process language models and use exactly the same interfaces for training and evaluation.

Sequence character language models are the basis for generating words in LingPipe's hidden Markov model taggers and chunkers. They are also used for smoothing token generation in token-based language models. Sequence character language models are also used to model sequences for spell checking. Like process language models, sequence models may also be used as the basis of classifiers.

The right language model to use depends on the data. If there are boundary effects and the sequences are relatively short, such as words or sentences as opposed to full documents, use sequence language models. If you need to compare strings of different lengths, use sequence language models. Otherwise, use process models.

6.6.1 Boundary and Length Effects

The process character language models, which we saw in the last section, treat the stream of characters as an ongoing process with no start and no end. One consequence of the process model is that the initial and final characters in a language model are not treated any differently than those in the middle.

Suppose we wanted to build a language model for words, or for sentences. Note that a subsequence of a word is not necessarily a word; for instance, *saril*, which is a substring of *necessarily*, is not an English word. Being like a word is graded, though, with *saril* being more word like than the substring *bseq* of *subsequence*.

Although it seems obvious, it is important statistically that each word or sentence is of finite length (though there is, in principle, no bound on this length other than the ability of our interlocuters to understand us and the amount of time we have). Words in most languages are not uniform from start to end. For instance, in English, there are suffixes like *ing* and *s* that appear far more often at the ends of words than at the beginning of words.

Another effect arises at the syllable level in English from pronounciation. Syllables tend to follow the *sonority hierarchy*, meaning that sonority tends to rise toward the middle of the syllable and fall at the end. For instance, vowels like *o* are the most sonorous, then nasals like *n* and *m*, then consonants like *r*, then stops like *t*. For instance, the one syllable sequences *tram* and *mart* are words, and both have sonorities that increase toward the voewl and then fall. In contrast, *rtam* is not pronounceable in English, with the usual reason given that it violates the syllable sonority principle, because *r* is more sonorous than *t*. Different langauges have different tolerances for syllabel sonority. On the one hand, Japanese typically only allows open syllables, which means no final consononants other than the most sonorant, the nasals like *n*). At the other extreme, languages like Slovenian don't even require a full vowel in a syllable and stack up consonants between syllables in a way that is hard for speakers of languages like Japanese or even English to pronounce.

In English sentences, we tend to find the first word capitalized more often than would be expected if the generation of characters was completely uniform. At the other end, sentence terminating punctuation characters, like the period and question mark, show up more frequently at the end of sentences than elsewhere. Perhaps more importantly for statistical modeling follows from this, namely that non-punctuation characters tend not to show up at the ends of sentences in well-edited text.

6.6.2 Coding Sequences as Processes

A standard trick used in both computer languages and mathematical models for converting a process-type model into a bounded sequence-type model is to add distinguished end-of-string characters. For instance, this is the technique used by the C programming language to encode strings. The byte zero (0x00) is used to encode the end of a string. Thus (ASCII) strings in C are stored in arrays one byte longer than the string itself. Similarly, lines in many file-oriented formats are delineated by end-of-line characters, such as the carriage return (U+000D). In order to present multiple sequences, characters such as the comma are used in comma-separated values (CSV) file formats used for spreadsheets. These commas are not part of the strings themselves, but act as separators.[3]

To encode sequence models as processes, we add a distinguished end-of-string character. It is treated like another character in that at each point, there is a probability of generating the end-of-string character given a sequence of previous characters. After genearing the end-of-string character, the sequence is terminated and no more characters may be generated. This leads to a bound on the otherwise ongoing character generation process. If the probabilty of generating the next character includes the possiblity of generating the end-of-string character, then the result is a bounded model that's properly normalized over all strings. That is, the sum of the probabilities of all strings sums to 1, so we have a proper model of strings. The process model generates sequences in which the sum of the probabilities of strings of a given length was 1. The upshot of this is that there's no way to use the process language models to compare string probabilities to each other when the strings are different lengths.

The use of a distinguished end-of-string character also handles boundary effects. Because we have to generate the end-of-string character given the suffix of the word, we can model the fact that strings are more likely to end following some character sequences than others. For instance, if we use *$* as our end-of-string marker (fllowing regular expression notation), then we might find *runs$* to

[3]In standard comma-separated value files, strings that contain commas may be quoted. Then, to include quotes, an escape mechanism is used, typically using a sequence of two double quotes within a quoted string to represent a quote. Getting the details of these encodings right is essential for preserving the true values of strings.

be more likely than *ru$* because of the probability of generating *$* given *runs* can be much higher than the probabilty of generating *$* given *ru$*; in symbols, we can have[4]

$$p(\$|runs) >> p(\$|ru). \tag{6.5}$$

We also use a distinguished begin-of-string character which we use to condition the probabilities for the string's prefix. Suppose we use ˆ as our begin-of-string character. What we do is reduce the probability of generating a string like *runs* to that of generating *runs$* given ˆ, thus setting

$$p_{\text{seq}}(runs) = p_{\text{proc}}(runs\$|\hat{}\,). \tag{6.6}$$

This defines the probability of *runs* in the sequence model, $p_{\text{seq}}()$, in terms of a conditional probability calculation in a process model, $p_{\text{proc}}()$.

6.6.3 Sequence Character Language Model Demo

The demo code is in the class file `SequenceLmDemo` in this chapter's package, `com.lingpipe.book.charlm`. The code is almost identical to the process LM demo introduced in Section 6.5.1.

Code Walkthrough

There are three command line arguments, populating the `ngram` integer, `csvTrain` as a string and `textTest` as a string.

For the sequence language model demo, the training data is assumed to be a sequence of texts separated by commas (with no allowance for commas in the strings for this demo). The work is done in constructing a trainable instance of the boundary language model, training on each text, then estimating the probability of the test text.

```
NGramBoundaryLM lm = new NGramBoundaryLM(ngram);
for (String text : csvTrain.split(","))
    lm.handle(text);
double log2Prob = lm.log2Estimate(textTest);
```

The method `split(String)` defined in `java.lang.String` splits the string into substrings at positions matching the string separator argument. Note that the separators are removed from the strings. It would be more efficient, given that we are splitting on a character, to do this explicitly and use the `train()` method taking an array slice; that way, there is no copying at all.

As usual, we have not shown the print statements or the boilerplate for the `main()` method or arguments.

[4]Although not precise in most contexts, the notation $x >> y$ indicates that x is much greater than y on some implicit scale. Typically this notation is only used when x is at least one or more orders of magnitude (factors of ten) greater than y.

Running the Demo

The Ant target `seq-demo` runs the demo, with arguments specified as properties `ngram` for n-gram length, `csv.train` for the comma-separated training texts, and `text.test` for the test text. For example,

```
>          ant -Dngram=4 -Dcsv.train="runs,jumps,eating,sleeping"
-Dtext.test="jumps" seq-demo
```

```
ngram=4   train=|runs|jumps|eating|sleeping|   test=|jumps|
log2 p(test|train)=-9.877
```

Thus the log (base 2) probability of seeing the string *jumps* in the model trained on the four specified words is roughly -10, which translates to 2^{-10}, or about 1/1000 on the linear scale. The reason the probability is so low is due to the smoothing in the model (see Section 6.7).

In contrast to the process model, in the sequence model, prefixes are not always more probable than longer strings. For instance, consider *jump* instead of `jumps`.

```
>          ant -Dngram=4 -Dcsv.train="runs,jumps,eating,sleeping"
-Dtext.test="jump" seq-demo
```

```
ngram=4   train=|runs|jumps|eating|sleeping|   test=|jump|
log2 p(test|train)=-13.037
```

Under this model (4-grams with default parameters) with the specified four training instances, the probability of *jumps* is about 8 times as likely as the probability of *jump* (2^{-13} is approximately 1/8000).

One of the most useful parts of the sequence model is how it generalizes endings and beginnings. For instance, *running* is estimated to be fairly probable (log probabilty -21.5). Even an unseen word such as *blasting* (log probability -46.4) is modeled as more likely than substrings like *blast* (log probability -46.6), because the model has learned that words are likely to end in *ing*.

Configuring the Boundary Character

The boundary character in the bounded language models may be specified by the client. Because the begin-of-sentence character is not generated, the same character may be used for both boundaries without loss of generality. The default for the boundary character is the reserved Unicode character U+FFFF. The boundary character must be set in the constructor, as it remains immutable during training and/or compilation.

6.7 Tuning Language Model Smoothing

Language models provide two parameters for tuning in addition to the length of n-grams. One of these controls the degree to which lower-order n-grams are blended with higher-order n-grams for smoothing. The second controls the number of characters in the base (zero order) model. We describe both in this section.

Operationally, the tuning parameters may be set at any time before compilation. These will dynamically change the behavior of the dynamic language models. But once a model is compiled, the tuning parameters may not be changed. This suggests a strategy of training a large model, then evaluating the tuning parameters on held out data (not part of training) to choose which is best.

6.7.1 The Problem with Maximum Likelihood

Recall that maximum likelihood estimation provides estimates for model parameters in such a way as to maximize the probability of the training character sequence(s). These estimates, for our simple language models, would be based on ratios of empirical counts. In particular, they will provide zero probabilty to a character given a context if that character was never seen in that context in the training data. The problem with this is that we are likely to run into subsequences of characters in new test data that were not in the training data. We would like to assign these unseen sequences non-zero probabilities so that every documents we observe in testing are assigned non-zero probabilities. If we don't, our higher-level applications of language models to tagging or spell checking or classification would almost certainly break down.

For example, suppose we're trying to model the probability of the letter *r* following the sequence *the new cha* with a 7-gram model. In a pure maximum likelihood 7-gram, the probability of seeing the letter *r* will be given by dividing the number of times *ew char* occurred in the training corpus by the number of times *ew cha* showed up followed by any character. For instance, there might be two instances of *ew chai*, perhaps as a part of *new chair* (furniture) or *new chai* (beverage) and one of *ew chas*, perhaps as part of *grew chastened*, in the corpus. In this case, the probability of the next letter being *i* is estimated at 2/3 and the probability of the next letter being *s* at 1/3. Note that there is no probability left over for the possibility of *r* being next. The maximum likelihood estimate for an n-gram model assigns zero probability to any string containing an n-gram that was not seen during training! While this may not be a problem in a short-context genetic modeling problem, it is a huge problem for natural language texts, where there is a long tail of rare n-grams which causes n-grams not seen during training to pop up in test sets. An easy example is numbers. If we have 8-grams, only 8-digit numbers showing up in the training set are assigned non-zero probability during prediction (testing). There are 10^8, that is 100 million, 8-digit numbers.

6.7.2 Witten-Bell Smoothing

To get around the novel n-gram problem, LingPipe employs a common language modeling technique known as *interpolative smoothing* to ensure that every sequence of characters is assigned a non-zero probability.

LingPipe uses an approached to interpolative smoothing is a slight generalization of the general method developed by Ian Witten and Timothy Bell in the context of text compression.[5] The idea behind Witten-Bell smoothing for n-grams involves blending order n predictions with order $n-1$ predictions, all the way down to a basic uniform distribution at order 0. Thus even if we haven't ever seen *ew char* in the corpus, we will still assign it a non-zero probability. That probability will be a weighted average of the 7-gram estimate of *r* given *ew cha*, the 6-gram estimate of *r* given *w cha*, the 5-gram estimate of *r* given *cha* (with an initial space), the 4-gram estimate of *r* given *cha*, the 3-gram estimate of *r* given *ha*, the 2-gram estimate of *r* given *a*, the 1-gram estimate of *r* given (empty context). Even if the higher-order n-grams provide zero probability estimates, the lower-order n-grams are likely to have been trained on the suffix of the sequence. But even if the character is novel, we bottom out in a uniform distribution, as we describe in the next section.

6.7.3 Uniform Base Distribution

This recursion bottoms out at the uniform estimate, which we consider a 0-gram estimate for the sake of terminological consistency. In the uniform estimate, each character is given equal probabilty. If there are 256 possible characters, as in Latin 1, then each has a 1/256 probability estimate in the uniform base case.

Because we have a uniform base case, LingPipe's character language models require the number of possible characters to be specified in the constructor. The default is `Character.MAX_VALUE`, which is $2^{16}-1$, which is a slight overestimate of the number of legal Unicode UTF-16 values (see the section on primitive data types in Java in the companion volume, *Text Processing in Java*). The number of characters can be set lower if we know we are dealing with ASCII or Latin 1 or another restricted subset of Unicode characters.

As long as we choose the number of characters to be greater than or equal to the number of unique characters seen in the combined training and test set, the probability estimates will be proper (that is, sum to 1 over all strings for sequence models and over all strings of a given length for process models).

6.7.4 Weighting the n-Grams

Interpolated n-gram models take a weighted average of the predictions of (maximum likelihood estimated) n-grams, $(n-1)$-grams, $(n-2)$-grams, and so on, eventually bottoming out with the uniform distribution over characters. What we haven't said is how the weights for each model are determined. We'll provide precise mathematical definitions in Section 6.13. For now, we will provide a more descriptive explanation.

The first thing to note is that the weighting is context dependent in that it depends on the characters in the context from which we are predicting the next character. That is, if we are predicting the next character from context *ew char*, we will likely have different weightings of the n-grams than if we predict from context *t of th*.

[5] Witten, Ian H. and Timothy C. Bell. 1991. The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory* **37**(4).

The weight assigned to the order n-gram model in the interpolated model is based on two factors determined from the context for the prediction and one parameter in the constructor. For deciding how heavily to weight an n-gram context of previous characters in prediction, we take into account how many times that context has been seen in the training data. The more we have seen a given context of previous characters in the training data, the more heavily its prediction is weighted. Basically, this just says that when we have more data, we can trust the prediction more.

The second factor for weighting n-grams is how many different outcomes we've seen given that context. If we have a context with many different outcomes, we will weight it less for prediction.

In addition to these contextual factors, LingPipe generalizes Witten and Bell's original method of interpolation with a parameter that controls the weighting between higher-order contexts and lower-order contexts. This parameter, which is named `lambdaFactor` in the code, may be optionally supplied during construction. In Witten and Bell's original approach, the lambda factor was implicitly set to 1.0, which is rarely an optimal value. This interpolation factor determines how much to weight higher-order contexts relative to lower-order contexts. The higher the value of this factor, the more agressively we smooth. In other words, with high values of the lambda factor, the more heavily the lower-order n-grams are weighted. The interpolation factor defaults to the length of the n-gram, which seems to work well in a large number of contexts.

6.8 Underlying Sequence Counter

The boundary language models are based on the sequence models. The sequence model probabilities are based on keeping a count of character subsequences. For both boundary and sequence models, the object that manages the counts is accessible through the method `substringCounter()`. This returns an instance of `TrieCharSeqCounter`. The name reflects the data structure used, a *trie*.

With the sequence counter, raw counts of arbitrary substrings in the training data may be queried and updated. It is unlikely that a client of the language model class will need to directly modify the underlying counter by any operation other than pruning (see Section 6.10).

All of the counts are based on `long` values. The sequence counter efficiently stores values for low count sequences using shorter integer representations.[6]

6.9 Learning Curve Evaluation

In this section, we provide a demo implementation of process languauge models that operate online, one character at a time. As each character in a text is visited, it is predicted given the characters that have come before. After that, it is added

[6]Carpenter, Bob. 2005. Scaling high-order character language models to gigabytes. *ACL Software Workshop.*

to the training data. This allows us to plot a learning curve that shows how our language model is doing.

6.9.1 Connection to Compression

It turns out that using a technique known as arithmetic compression (which LingPipe does not support), it is possible to use the probabilistic predictions of a language model to perform compression of arbitrary sequences of bytes, such as files. The key result is that for every character c we encounter after context s, we can represent it using a number of bits equal to $\log_2 p(c|s)$. That is, the better we can predict the next character, as measured by the probability assigned to it, the better we can perform compression. This makes sense, as this is a measure of predictability, and predictable texts are easily compressible no matter what compression technique is used (language models, Zip-style, Burrows-Wheeler style, etc.)

Given the fundamental relation between prediction and compression, the number of bits required to compress a sequence is just the sum of the log (base 2) probability estimates for the symbols in the sequence.

Average, or total log probability is also the standard (and most useful) overall measure of language model performance. Because we can learn on the fly, we can plot out the learning curve of average bits per symbol as we move through the sequence. By inspecting the resulting learning curves, we can tell how the model improves with more training data. In every LingPipe application, such as tagging, classification and spelling correction, language models with better held-out predictive accuracy provide better accuracy to the application of which they are a part.

6.9.2 Code Walkthrough

This demo introduces a few new tools from LingPipe, which we will explain as we go. Two of these new methods are language-model specific, including a method to train a single character given a context and a method to estimate the probability of a character given the previous characters.

Because we are thinking of our text as a streaming sequence of characters, we use a process language model.

As usual, we have a number of parameters supplied to the `main()` method by the command line. In order, we assume an integer `ngram` for the n-gram length, an integer `numChars` for the total number of unique characters possible (not necessarily observed) in the train and test sets, a double `lambdaFactor` for the interpolation ratio, and a file `dataDir` for the directory in which to find the test texts, arranged with one test per file. This will allow you to easily test your own data by just pointing this demo at the directory containing the text.

To start, we iterate over the files in the data directory, extracting their characters and setting up a language model with the parameters specified on the command line.

```
for (File file : dataDir.listFiles()) {
```

```
char[] cs = Files.readCharsFromFile(file,"ISO-8859-1");

NGramProcessLM lm
    = new NGramProcessLM(ngram,numChars,lambdaFactor);

OnlineNormalEstimator counter
    = new OnlineNormalEstimator();
```

The characters are extracted using LingPipe's static utility method `readCharsFromFile`, which is in the `com.aliasi.util.Files`. Note that we have specified the character encoding ISO-8859-1, aka Latin1. The reason we do this is that Latin1 characters stand in a one-to-one relationship with bytes. By specifing Latin1, each character corresponds to a single byte with the same value. This lets us model arbitrary byte sequences using characters.

We then set up an instance of `OnlineNormalEstimator`, which is a class in `com.aliasi.stats` that computes running averages and variances, as we will see in the next block.

Next, we iterate over the characters in the text, first estimating each character's probability given the previous characters, then adding the character to the training data (given its context).

```
for (int n = 0; n < cs.length; ++n) {
    double log2Prob
        = lm.log2ConditionalEstimate(cs,0,n + 1);

    lm.trainConditional(cs, Math.max(0,n - ngram),
                        n, Math.max(0,n - 1));
```

First, we estimate the log (base 2) probability of the next character given the characters we have already seen. This method takes a character array (`cs`), a start position (`0`), and an end position (`n+1`). The value returned is the probability of the n-th character (`cs[n]`) given the previous characters (`cs[0]` through `cs[n-1]`). We use the value `0` as the start position because the method is clever enough to only use as long a context for prediction as it stores.

Next, we add the character to the training data. This is done with the model method `trainConditional()`, which is supplied with several arguments. First, `cs`, is the character slice. Next is the starting position from which to train. Because we are using n-grams, this is the current position minus the n-gram length, but given that it cannot be less than 0, we use `max(0,n - ngram)` as our starting position. The third argument, here `n`, specifies the final position which is trained. The fourth and final argument, here `max(0,n - 1)`, indicates the end of the context. Here, we just train a single character at a time, so the context ends one position before the character we're training (`n - 1`).

Finally, we increment our average and deviation accumulator and extract the current average and standard deviation.

```
    counter.handle(log2Prob);
    double avg = counter.mean();
    double sd = counter.standardDeviationUnbiased();
```

This involves calling the `handle(double)` method on the online normal estimator, which adds the predicted log probability to the set of observed probabilities.

After that, we pull the current sample mean (otherwise known as the average) and the sample standard deviation (with an unbiased estimate) from the online normal estimator. These sample means and deviations are based on the characters we have seen so far.

As usual, we have suppressed the various print statements.

6.9.3 Running the Online Learning Curve Demo

The Ant target `learning-curve` calls the demo, with parameters supplied by properties `ngram` for the maximum n-gram size, `num.chars` for the total number of unique characters, `lambda.factor` for the interpolation ratio, and `data.dir` for the directory in which the data is found.

For test data, we use the Canterbury corpus (which we describe in Section D.1). The Canterbury corpus is a standard corpus introduced by Ross Arnold and Timothy Bell (of Witten and Bell smoothing) to evaluate compression algorithms. The Canterbury corpus contains a mixture of document types, including novels, plays and poetry, as well as C and Lisp source code, a Unix man page, a Microsoft Excel binary spreadsheet (`.xls` format), and executables for the SPARC architecture. In Section D.1, we describe the files more fully along with their sizes.

```
>               ant -Dngram=6 -Dnum.chars=256 -Dlambda.factor=6.0
-Ddata.dir=../../data/canterbury learning-curve
```

```
PROCESS LM PARAMETERS
    ngram=6
    numchars=256
    lambdaFactor=6.0
    data.dir=C:\lpb\data\canterbury

Processing File=..\..\data\canterbury\alice29.txt
   log2Prob[c[0]]=-8.000        mean=-8.000   sd= 0.000
   log2Prob[c[5000]]=-3.018     mean=-3.352   sd= 2.692
   log2Prob[c[10000]]=-3.382    mean=-2.972   sd= 2.575
   log2Prob[c[15000]]=-0.011    mean=-2.798   sd= 2.559
   log2Prob[c[20000]]=-5.706    mean=-2.701   sd= 2.533
...
   log2Prob[c[140000]]=-0.852   mean=-2.113   sd= 2.351
   log2Prob[c[145000]]=-2.414   mean=-2.105   sd= 2.348
   log2Prob[c[150000]]=-0.410   mean=-2.102   sd= 2.350
   log2Prob[c[152088]]=-24.140  mean=-2.103   sd= 2.352

Processing File=..\..\data\canterbury\asyoulik.txt
   log2Prob[c[0]]=-8.000        mean=-8.000   sd= 0.000
   log2Prob[c[5000]]=-0.386     mean=-3.397   sd= 2.462
   log2Prob[c[10000]]=-3.013    mean=-3.072   sd= 2.400
...
   log2Prob[c[120000]]=-1.099   mean=-2.342   sd= 2.309
   log2Prob[c[125000]]=-0.047   mean=-2.337   sd= 2.310
   log2Prob[c[125178]]=-0.001   mean=-2.337   sd= 2.310
...
```

We've instrumented the code so that every 5000 characters it prints out the prediction for that character and the running mean and standard deviation from the set of samples so far. We see that for file `alice29.txt`, which is the text of Lewis Carroll's book *Alice in Wonderland*, the final average log (base 2) probability per character after all 152,088 characters have been processed is -2.103. This corresponds to about 2 bits per character, which is a typical result for language models of text given this little training data (a book is not much on the scale of the text available for training, such as years of newswirte or MEDLINE, which run to gigabytes, or the web, which runs to terabytes if not petabytes).

Also note the rather large sample standard deviation, which indicates that there is a vast range of probability estimates. For instance, character 15,000 is predicted with log probabilty -0.011, which is almost certainty, whereas character 20,000 is predicted with probabilty -5.7, which isn't much better than chance given the set of characters.

Processing the entire corpus takes less than a minute on my aging workstation. The final results for the various files are as follows (see Section D.1 for details on the contents of the files).

File	*Bytes*	*Avg*	*SD*	*GZip*
`alice29.txt`	152,089	-2.103	2.352	2.85
`asyoulik.txt`	125,179	-2.337	2.310	3.12
`cp.html`	24,603	-2.285	2.725	2.59
`fields.c`	11,150	-2.020	2.584	2.24
`grammar.lsp`	3,721	-2.434	2.945	2.65
`kennedy.xls`	1,029,744	-1.753	3.632	1.63
`lcet10.txt`	426,754	-1.882	2.299	2.71
`plrabn12.txt`	481,861	-2.204	2.200	3.23
`ptt5`	513,216	-0.829	2.484	0.82
`sum`	38,240	-2.702	3.451	2.67
`xargs.1`	4,227	-2.952	2.649	3.31

We have also presented compressions results based on GZip, expressed as the number of bits per byte required for the compressed file.[7] Note that for language data, our 6-gram model with default parameters is far superior to GZip, with GZip being close or slightly better on some of the binary data, like the spreadsheet.

These are not optimal parameters for our model, but rather defaults. We recommend using 5-grams or 6-grams for English text in situations where there is relatively little training data (as in these examples). We also recommend setting the interpolation ratio to the n-gram length. We encourage you to try differeent parameters, especially for n-gram length and interpolation ratio; you should leave the number of characters fixed at 256, which is the maximum number of Latin1 characters.

[7]Given on the Canterbury corpus page at `http://corpus.canterbury.ac.nz/details/cantrbry/RatioByRatio.html`.

6.10 Pruning Counts

One problem with the naive application of n-gram character language models is that the size of the model increases with the amount of training data.[8] Although the model size increases sublinearly due to repeated subsequences,[9] it can grow very large relative to available storage with long n-grams.

One way to keep the memory constrained is to *prune* the underlying counts from time to time (the etymology of the term is that it's typically used for tree-like data structures, such as the ones used for language models). Pruning works by removing all strings whose count falls below some minimum value. If there are millions of such contexts, removing the ones with low count often has very little effect on the overall probabilities assigned by the model. One reason for this is that the low counts tend to be associated with long sequences that are low frequency. These tend to have shorter sequences which are almost as representative, but slightly higher frequency. Another reason is that the low frequency counts are for low frequency training events which tend to occur less often in new data than high frequency training events; if not, there's something wrong with the training data that's making it unlike the test data.

6.11 Compling and Serializing Character LMs

Both implementations of character language models, `NGramProcessLM` and `NGramBoundaryLM`, implement Java's `Serializable` interface (see the sections on Java serialization and LingPipe compilation in the I/O chapter of the companion volume, *Text Processing in Java*).

6.11.1 Compilation

Like many of LingPipe's statistical models, the character language model implementations come in two flavors. First, there are the two classes we've discussed already, `NGramProcessLM` and `NGramBoundaryLM`, which implement handlers for character sequences for training. These two classes are both compilable (see the section on object compilation in the I/O chapter of the companion volume, *Text Processing in Java*). Compiling an instance of one of these classes produces instances of `CompiledNGramProcessLM` and `CompiledNGramBoundaryLM`. Although based on shared abstract superclasses, these abstract bases are not public because there is a sufficiently rich set of interfaces (see Section 6.4 for the complete list).

[8]This is a general property of so-called *non-parameteric models*. This terminology is confusing, because non-parameteric models rather than being parameter free, have an unbounded number of parameters. The unboundedness comes from the fact that such models store the data they are trained on. Another example of a non-parametric model in LingPipe is K-nearest-neighbor classifification, which stores all of the training instances to compare to the test instances.

[9]Sublinear growth means that the model size is less than a constant factor times the size of the training data, meaning less than $\mathcal{O}(n)$. The reason storage grows non-linearly is due to repeated n-grams, which must repeat with a finite. As a non-linguistic example, binary search grows sublinearly with size, being $\mathcal{O}(\log n)$.

Compiled character language models are much much faster than the dynamic language models that allow training. The reason for this is threefold. First, all of the smoothing is done at compile time, so only a single n-gram need be evaluated, the one with the longest matching context. Second, a suffix tree data structure is used to make it efficient to find the longest suffix for sequences of evaluations. Third, all of the logarithms are precomputed, so only addition is required at run time (see Equation 6.2).

Compiled language models only support the estimation interfaces, though they support both the joint (sequence of characters) and conditional (one character given previous characters) forms of estimation. The only other method they support is `observedCharacters()`, which returns an array containing all of the characters that have been seen (including the boundary character for sequence models).

The classes that depend on character language models use the dynamic forms of language models for training and the compiled form in compiled models.

6.11.2 Serialization

Character language models are also serializlable. The deserialized version is an instance of the same class as the language model that was serialized, and will have exactly the same set of stored n-grams and predictions.

A typical use case is to train a language model on some data, then compile it. But that does not allow further training. If the model is also serialized, then it may be deserialized, trained with further data, then compiled for use and serialized for later training.

6.12 Thread Safety

As with most of LingPipe's classes, language models are thread safe under read-write synchronization. Specifically, any number of operations that read from language models, including all of the estimation and sequence counter operations may be run concurrently with each other. Write operations on the other hand, like training, pruning, and setting configuration parameters like the interpolation ratio, must operate exclusively of all other read or write operations.

6.13 The Mathematical Model

The mathematics of character language models are straightforward. We'll begin with the process language model and move onto the bounded language model.

6.13.1 Strings

First we'll establish some notation for talking about strings mathematically. Suppose we have an alphabet Σ of characters.[10] Let Σ^n be the set of all sequences of

[10]For character language models, Σ will consist of UTF-16 byte pairs. If we stay within the basic multilingual plane (BMP) of Unicode, these will correspond to Unicode characters (code points). If

such characters of length n, which are called strings. In symbols,

$$\Sigma^n = \{(s_1, \ldots, s_N) \mid s_n \in \Sigma \text{ for } 1 \leq n \leq N\}. \tag{6.7}$$

Given a string, we often write $|s| = n$ if $s \in \Sigma^n$ and say that s is of length n.

Next, we take Σ^* to be the set of all strings, defined by

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n. \tag{6.8}$$

We will use the symbol ϵ is used for the *empty string*, which is defined as the unique sequence of length 0.

The fundamental operation over strings is *concatenation*. The concatenation of two strings $s = (s_1, \ldots, s_M)$ of length M and $t = (t_1, \ldots, T_N)$ of length N is taken to be the

$$s \cdot t = (s_1, \ldots, S_M, t_1, \ldots, T_N). \tag{6.9}$$

Strings have several pleasant algebraic properties. The empty string is the identity for concatenation, so that

$$\epsilon \cdot s = s \cdot \epsilon = s. \tag{6.10}$$

Concatenation is associative, so that

$$(s \cdot t) \cdot u = s \cdot (t \cdot u). \tag{6.11}$$

Thus the order of concatenation of a sequence of strings doesn't matter. Together, these two properties define what is known as a *monoid*. Note that the natural numbers ($\mathbb{N}$) under addition (+) form a monoid with identity zero (0). By analogy to the natural numbers, if s is a string, we write s^n for the string consisting of n concatenations of s. Thus $s^0 = \epsilon$ is the empty string.

Of course, unlike the case for arithmetic addition, concatenation is not a symmetric operation, so that we don't, in general, have $s \cdot t = t \cdot s$.

6.13.2 The Chain Rule

The basic idea of n-gram language models is to use the chain rule, which is derivable from the definition of joint and conditional probabilities, to factor the probabiltiy of a string into the product of the probabilities of each of its characters given the characters that came before it. In symbols, suppose we have a string $s_1, \ldots, s_N$ of length N. The chain rule tells us we can compute its probability as

$$p(s_1, \ldots, s_k) = p(s_1) \times p(s_2|s_1) \times \cdots \times p(s_N|s_1, \ldots, s_{N-1}). \tag{6.12}$$

6.13.3 Markov Chains

The point of using a limited-context-length n-gram is that you only look back at most $(n-1)$ characters. To specify that our models have limited context, we truncate the contexts in the chain rule, giving us

$$p(s_N|s_1, \ldots, s_{N-1}) = p(s_k|s_{k-n+1}, \ldots, s_{k-1}). \tag{6.13}$$

the characters used go beyond the BMP, some of these may correspond to the high or low half of a surroage pair.

Note that unlike the chain rule, which holds for any probabilities at all in full generality, the use of limited contexts is a modeling decision. We know that this decision is only an approximation for natural language, which exhibits much longer range dependencies. For instance, if a person named *Tony* is mentioned in an article, the name is likely to show up again much later in the document.

Combining the chain rule in Equation 6.12 with our limited context model Equation 6.13, we have

$$p(s_1, \ldots, s_k) = p(s_1) \times p(s_2|s_1) \times \cdots \times p(s_k|s_{k-n+1}, \ldots, s_{k-1}). \tag{6.14}$$

For instance, if we have a 2-gram model and string *abra*, our model factors the probabilty of *abra* as

$$p(abra) = p(a) \times p(b|a) \times p(r|b) \times p(a|r). \tag{6.15}$$

Assuming that the contexts we use for prediction are finitely bounded, meaning that predicting the next character involves only a fixed finite window, we have what is known as a *Markov model*. A particular sequence of characters generated from such a model is then called a *Markov chain*. In a Markov model of character string generation, each character is generated based on finitely many previous characters. Note that being Markovian doesn't say how the probabilty of the next character is estimated, but only that it depends on at most a bounded finite history.

6.13.4 Training Data Sequence Counts

Now suppose we have seen training data in the form of a set of M strings $\sigma_1, \ldots, \sigma_M$. Given such a training set, let $\mathrm{cnt}(s)$ be the number of times the string s appeared as a substring of one of the training strings σ_m. For instance, if the training strings are *abe* and *abracadabra*, the substring *ab* shows up 3 times, once in the first string and twice in the second.

Let $\mathrm{excnt}(\sigma)$ be the count of all extensions by one character of the string σ, that is

$$\mathrm{excnt}(\sigma) = \sum_{s' \in \Sigma} \mathrm{cnt}(\sigma \cdot s). \tag{6.16}$$

For instance, in the training data, the count of *ra* is 2, but the extension count of *ra* is only 1, because the second training string ends with *ra* which is not extended.

6.13.5 Maximum Likelihood Estimates

We'll use the notation $p_{\mathrm{MLE}}(s_n|s_1, \ldots, s_{n-1})$ for the maximum likelihood estimate of the conditional probabilty of a seeing the character s_n given the previous $(n-1)$ characters $s_1, \ldots, s_{n-1}$. The maximum likelihood estimate model is derived in the usual way, by counting in the training data, giving us

$$p_{\mathrm{MLE}}(s_n|s_1, \ldots, s_{n-1}) = \frac{\mathrm{cnt}(s_1, \ldots, s_{n-1}, s_n)}{\mathrm{excnt}(s_1, \ldots, s_{n-1})}. \tag{6.17}$$

That is, we count up all the times we've seen the character in question following the string and divide by the number of times the string was extended by any character.

These maximum likelihood estimates will almost certainly assign zero probability to many characters in even relatively modest length n-grams.

6.13.6 Witten-Bell Smoothing

We follow Witten and Bell's approach to smoothing these maximum likelihood estimates through interpolation. The basic definition is recursive, interpolating the maximum likelihood estimate at length n with the smoothed estimate of length $(n-1)$. In symbols, we let $\lambda_{s_1,\ldots,s_{n-1}} \in [0,1]$ be the weight of the maximum likelihood estimate for predicting the next character given that the previous characters were $s_1, \ldots, s_{n-1}$. The recursive case of the definition is then given by

$$\begin{aligned} p(s_n|s_1,\ldots,s_{n-1}) = \lambda_{s_1,\ldots,s_{n-1}} \times p_{\mathrm{MLE}}(s_n|s_1,\ldots,s_{n-1}) \\ + (1-\lambda_{s_1,\ldots,s_{n-1}}) \times p(s_n|s_1,\ldots,s_{n-2}). \end{aligned} \tag{6.18}$$

On each application of this rule, the context is shortened by one character.

Next, we need to define the interpolation ratios $\lambda_s \in [0,1]$ for arbitrary strings $s \in \Sigma^*$. Witten and Bell define the interpolation ratio by setting

$$\lambda_s = \frac{\mathrm{excnt}(s)}{\mathrm{excnt}(s) + (L \times \mathrm{numext}(s))}, \tag{6.19}$$

where $L \geq 0$ is the interpolation parameter specified in the language model constructor, $\mathrm{excnt}(s)$ is defined as in Equation 6.16, and $\mathrm{numext}(s)$ is defined as the number of unique characters were observed following the context s,

$$\mathrm{numext}(s) = |\{c \mid \mathrm{cnt}(s \cdot c) > 0\}| \tag{6.20}$$

where $|x|$ is the cardinality (number of members) of the set x.

The recursion bottoms out by interpolating with the uniform distribution $p_{\mathrm{U}}(s_n)$. If K is the total number of characters, then the uniform distribution over characters is defined by

$$p_{\mathrm{U}}(s_n) = 1/K. \tag{6.21}$$

Thus the recursion bottoms out by interpolating the maximum likelihood unigram model with the uniform distribution,

$$p(s_n) = \lambda_\epsilon \times p_{\mathrm{MLE}}(s_n) + (1-\lambda_\epsilon) \times p_{\mathrm{U}}(s_n), \tag{6.22}$$

where the interpolation ratio λ_ϵ is defined as usual.

6.13.7 Process Language Models

The language models we have just defined following Witten and Bell's definitions are process language models.[11] For process language models, if we let $p(s)$ be

[11] The terminology, which is not standard in the language-modeling literature, derives from the fact that generating the sequence of characters forms what is known as a *random process.*

the probability of a string s, then

$$\sum_{s \in \Sigma^n} p(s) = 1. \tag{6.23}$$

6.13.8 Sequence Language Models

Unlike the process language models, for which the probabilities of all strings of a given length sum to 1, a bounded language model is a proper model of arbitrary character sequences. Thus a bounded language model, with probability $p(s)$, is such that

$$\sum_{s \in \Sigma^*} p(s) = 1. \tag{6.24}$$

The only difference between this equation and Equation 6.23 is that the sum is over all strings, Σ^*, rather than over strings of a fixed length n, Σ^n.

Thus we think of sequence language models as generating finite sequences of characters then stopping; a process language model goes on generating characters indefinitely.

We follow the fairly standard, though often undocumented, procedure of definining bounded language models in terms of process language models. Suppose we are definining sequences over the character set Σ. We will choose a new character, b, called the boundary character, such that $b \notin \Sigma$. We then define a process language model over the extended alphabet $\Sigma \cup \{b\}$ and use it to define our sequence language model distribution.

Despite adding a new boundary symbol b, our probabilities are still taken to be defined over strings in Σ^*. Specifically, we define the probability of a string $s = s_1, \ldots, s_M \in \Sigma^*$ as

$$p_B(s_1, \ldots, s_M) = p_B(s_1, \ldots, s_M, b | b), \tag{6.25}$$

where $p_B()$ is the sequence language model probability distribution and $p()$ is the underlying process language model. In words, we start with the boundary character b as context, then generate our string $s = s_1, \ldots, s_M$, then generate a final boundary character b. Note that we chose s from the unaugmented set of strings Σ^*.

Because we now have a termination condition for our character generating process, it is fairly straightforward to verify that the process generates a probability distribution normalized over all strings, as shown in Equation 6.24. We start with the boundary character, though don't generate it, which is why it's in the context in Equation 6.25. We then continue generating, at each point deciding whether to generate the boundary character b and stop or to generate a non-boundary character and keep going. Because the sum of each next character in the augmented alphabet $\Sigma \cup \{b\}$ sums to 1.0, the normalization in Equation 6.24 holds.

Chapter 7

Tokenized Language Models

In addition to the character-level language models described in Chapter 6, LingPipe provides token-level language models. These may be configured to predict arbitrary texts including whitespace, or may be used just for predicting sequences of tokens.

7.1 Applications of Tokenized Language Models

Like character language models, some applications for which were discussed in Section 6.1, tokenized language models may be used as the basis for for classification. An additional applciation of tokenized language models is for the discovering of new or interesting phrases or collocations. For significant phrase extraction, a background language model characterizes a fixed comparison text, such as all the text collected from newspapers in a year, and a foreground model characterizes some other text, such as the text collected this week; differential analysis is then done to find phrases that appear more frequently than one would expect in the foreground model, thus modeling a kind of “hot topic” detection.

7.2 Token Language Model Interface

In addition to the interfaces discussed in Section 6.4, there is an interface for tokenized language models.

7.2.1 Tokens: `LanguageModel.Tokenized`

For language models based on tokens, the interface `LanguageModel.Tokenized` supplies a method for computing the probability of a sequence of tokens,

```
double tokenLog2Probability(String[] toks, int start, int end);
```

This returns the log (base 2) probability of the specified slice of tokens. As usual, the start is inclusive and the end exclusive, so that the tokens used are `toks[start]` to `toks[end-1]`. There is also a method to return the linear probability, which is just the result of exponentiating the log probability.

Unlike character-based probabilities, this token probability only counts the probability of the tokens, not the whitespaces (or other non-token characters) surrounding them.

A further disconnect for tokenized language models is that they may use a tokenizer that reduces the signal, conflating some of the inputs. In that case, what is being modeled is the probability of the sequence of tokens, not the raw character sequence. For example, any two character sequences producing the same tokens will produce the same token probability even if they are different strings.

Chapter 8

Spelling Correction

Chapter 9

Classifiers and Evaluation

We are going to introduce the classifier interface, discuss what classifiers do, and then show how to evaluate them. In subsequent chapters, we consider a selection of the classifier implementations offered in LingPipe.

9.1 What is a Classifier?

A classifier takes inputs, which could be just about anything, and return a classification of the input over a finite number of discrete categories. For example, a classifier might take a biomedical research paper's abstract and determine if it is about genomics or not. The outputs are "about genomics" and "not about genomics."

Classifiers can have more than two outputs. For instance, a classifier might take a newswire article and classify whether it is politics, sports, entertainment, science and technology, or world or local news; this is what Google and Bing's news services do.[1] Other applications of text classifiers, either alone or in concert with other components, range from classifying documents by topic, identifying the language of a snippet of text, analyzing whether a movie review is postive or negative, linking a mention of a gene name in a text to a database entry for that gene, or resolving the sense of an ambiguous word like *bank* as meaning a savings institution or the sloped land next to a stream (it can also mean many other things, including the tilt of an airplane, or even an adjective for a kind of shot in billiards). In each of these cases, we have a known finite set of outcomes and some evidence in the form of text.

9.1.1 Exclusive and Exhaustive Categories

In the standard formulation of classification, which LingPipe follows, the categories are taken to be both exhaustive and mutually exclusive. Thus every item being classified has exactly one category.

One way to get around the exhaustiveness problem is to include an "other" category that matches any input that doesn't match one of the other categories.

[1] At `http://news.google.com` and `http://news.bing.com`.

The other category is sometimes called a “sink” or “trash” category, and may be handled differently than the other categories during training.

Exclusivity is more difficult to engineer. While it is possible to allow categories that represent more than one outcome, if we have n base categories, we’ll have $\binom{n}{2}$ unique pairs of categories. The combinatorics quickly gets out of hand.

If an item needs to be classified for two cross-cutting categorizations, we can use two classifiers, one for each categorization. For instance, we might classify MEDLINE citations as being about genomics or not, and about being about clinical trials or not. The two binary classifiers produce four possible outcomes. Another example of this would be to use two binary classifiers for sentiment, one indicating if a text had positive sentiment or not, and another indicating if it had negative sentiment or not. The result is a four-way classification, with a neutral article having neither positive nor negative sentiment, a mixed review having both positive and negative sentiment, and a positive or negative review having one or the other.

The latent Dirichlet allocation (LDA) model we consider in Chapter 14 assigns more than one category to each document, under the assumption that every document is a mixture of topics blended according to some document-specific ratio that the model infers.

9.1.2 Ordinals, Counts, and Scalars

A classifier is an instance of what statisticians call categorical models. The outcome of a classification is one of finitely many categories, not a continuous scalar value like a floating-point number.

Classifiers deal with categorical outcomes. Ordinal outcomes are like categorical outcomes, only they come with an order. Examples of ordinal outcomes include rating movies on a $\{1, 2, 3, 4, 5\}$ scale, answering a survey question with strongly-disagree, disagree, neutral, agree, or strongly agree, and rating a political attitude as left, center, or right.

In many cases, we will be dealing with counts, which are non-negative integers, which are also called “natural numbers”. Examples of count variables include the number of times a word appears in a document, the length of a document’s title in characters, and the number of home runs a baseball player gets in a season. Classifiers like naive Bayes convert word count data into categorical outcomes based on a probability model for documents (see Chapter 10).

Scalar outcomes are typically continuous. Examples of continuous scalar variables include a person’s height, the length of the vowel sequence in milliseconds in the pronunciation of the word *lion*, or the number of miles between two cities.

Rating movies on an ordinal 1-5 scale doesn’t allow ratings like 1.793823. Half-star ratings could be accomodated by including additional discrete outcomes like 1.5, 2.5, 3.5 and 4.5, leaving a total of 9 possible ratings. When there are 100 possible ratings, it makes less sense to treat the outcome as ordinal. Often, it is approximated by a continuous scalar variable.

Models that predict or estimate the values of ordinal, count and scalar values all have different evaluation metrics than categorical outcomes. In this section, we focus exclusively on categorical outcomes.

9.1.3 Reductions to Classification Problems

Many problems that don't at first blush appear to be classification problems may be reduced to classification problems. For instance, the standard document search problem (see the chapter on the Lucene search library in the companion volume, *Text Processing in Java*) may be recast as a classification problem. Given a user query, such as `[be-bim-bop recipe]`, documents may be classified as relevant or not-relevant to the search.

Another popular reduction is that for ranking. If we want to rank a set of items, we can build a binary classifier for assessing whether one item is greater than another in the ordering. In this case, it is then a challenge to piece back together all these binary decisions to get a final rank ordering.

Ordinal classification problems may also be reduced to classifications. Suppose we have a three-outcome ordered result, say $N \in \{1, 2, 3\}$. We can develop a pair of binary classifiers and use them as an ordinal three-way classifier. The first classifier will test if $N < 2$ and the second if $N < 3$. If the first classifier returns true, the response is 1, else if the second classifier returns true the response is 2, else the response is 3. Note that if the first classifier returns true and the second false, we have an inconsistent situation, which we have resolved by returning 1.

Just about any problem that may be cast in a hypothesize-and-test algorithm may use a classifier to do the testing, with a simple binary outcome of accept or reject. For instance, we can generate possible named-entity mention chunkings of a text and then use a binary classifier to evaluate if they are correct or not.

Such reductions are often not interpetable probabilistically in the sense of assigning probabilities to possible outcomes.

9.2 Kinds of Classifiers

LingPipe provides several varieties of classifiers. These are ordered in terms of the amount of information they provide as the result of classification. In this section, we describe them in order from least informative output to most informative.

The reason there are multiple classification classes is that different classification techniques have different capabilities.

Each type of classifier comes with its own evaluation class which is tailored to the information provided by that kind of classifier.

9.2.1 First-Best Classifiers and Classifications

A simple first-best classifier need only return its best guess for the category of each input item. Some applications only allow a single best guess, and thus some evaluations are geared toward evaluating only a classifiers' first-best guess for an input.

A first-best classifier will implement the interface `BaseClassifier<E>`, providing classifications of type `E` objects. The interface defines a single method, `classify(E)`, which returns a first-best classification for the specified input.

The return type of the classify method is `Classification`. The `Classification` base class implements a single method `bestCategory()`, which returns a string representing the category. While this could've been done with a return type consisting of just a string, the base class `Classification` is going to be extended to richer types of classification, which would not be possible with `String`.

A `Classification` instance is constructing from the string corresponding to its best category, using constructor `Classification(String)`.

Evaluation of first-best classifiers is carried out using the class `BaseClassifierEvaluator`. Base evaluators track confusion matrices, which represent the full empirical response profile of a first-best classifier by category. From a confusion matrix, it's possible to derive all sorts of other evaluations, which are described in the Javadoc for the evaluator and exemplified later in this chapter.

9.2.2 Ranking Classifiers and Classifications

A ranking classifier returns the possible categories in rank order of their match to the input. The top ranked answer is the first-best result. We still assume exhaustive and exclusive categories, but the classfier supplies its second-best guess, third-best guess and so on.

A ranking classifier implements the interface `RankedClassifier` which extends `BaseClassifier`. In addition to returning the best category, a ranked classifier is able to return additional guesses along with their ranks. The idea is that the higher the rank, the more confident the classifier. Because there is more than one result, a ranked classifier implements a method `size()` returning an integer indicating the number of categories in the classification. The method `category(int)` returns the string corresponding to the category at a specific rank. Assuming the size is non-zero, the method call `category(0)` will return the same value as `bestCategory()`. If there are at least two categories, the call `category(1)` returns the second-highest ranking classification by the classifier, and so on.

A ranked classifier is constructed from a sequence of categories ordered from best guess to worst guess using the constructor `RankedClassification(String[])`.

A ranked classifier is evaluated using the `RankedClassifierEvaluator`. The ranked evaluator extends the base evaluator the same way that a ranked classifier extends a base classifier and a ranked classification extends a first-best classification. A ranked classifier evaluator adds ranked evaluation statistics like the mean reciprocal rank of the true answer.

9.2.3 Scored Classifiers and Classifications

A scoring classifier goes one step further and assigns a (floating point) score to each categories. These may then be sorted to provide a ranking and a first-best result, with higher scores taken to be better matches. For example, LingPipe's implementations of averaged perceptrons returns scored results.

A scored classifier implements the interface ScoredClassifier, which extends RankedClassifier. In addition to returning sizes and categories at ranks, the method score(int) returns a double-precision floating-point score for the result at the specified rank. These scores should be decreasing as the rank decreases. That is, score(0) should be less than or equal to score(1) and so on.

A ScoredClassification is constructed from parallel arrays of of categories and scores using the constructor ScoredClassification(String[],double[]). The scores must be sorted in decreasing order; if they are not, the constructor throws an illegal argument exception. There is a utility method create(List<ScoredObject<String») that creates a scored classification from an unsored list of scored strings. The ScoredObject class, in LingPipe's package com.aliasi.util is such that an instance of ScoredObject<String> implements a method getObject() returning a string and a method score() returning a double-precision floating-point score.

A scored classifier is evaluated with a ScoredClassifierEvaluator, which extends the ranked evaluator class. The results of a scored classifier evaluation rare stored in a ScoredPrecisionRecallEvaluation object. This evaluation class supports results like precision-recall and ROC curves, recall at specific ranks, and other evaluations depending on scores.

9.2.4 Conditional Classifiers and Classifications

Scores for categories given an input are often normalized so that they represent an estimate of the conditional probability of a category given the input. This is the case for LingPipe's logistic regression classifiers and k-nearest neighbors classifiers. For instance, a classifier might see a MEDLINE citation about testing for a disease with a known genetic component and estimate a 40% probability it is about genomics and 60% probability that it is not about genomics. Because the outcomes are exhaustive and exclusive, the probabilities assigned to the categories must sum to 1.

A conditional classifier implements the interface ConditionalClassifier, which extends ScoredClassifier. It adds a method conditionalProbability(int) for returning the conditional probability of the result at the specified rank. It also contains a method conditionalProbability(String) which returns the probability estimate for a specified category.

There are several constructors for conditional classifiers. The basic constructor is ConditionalClassifier(String[],double[]), which takes a pair of parallel arrays representing the categories and their conditional probabilities, in order. The sum of the probabilities must be 1. An optional third argument may be provided to specify the tolerance of the probabilities summing to 1. A second constructor is Conditional(String[],double[],double[]), with three parallel arrays, the first floating point one of which contains scores and the second of which contains conditional probabilities. The scores do not need to be the same as the conditional probabilities, but both must be presented in descending order.

If the scores or probabilities are not in descending order, an illegal argument exception is raised. An exception is also raised if the arrays do not ahve the same length.

There are two convenience static factory method for creating conditional classifications. The first, `createLogProbs(String[],double[])` takes probabilities on the log scale and normalizes them to proper probabilities by exponentiating and sorts them appropriately.[2] The second static factory method has the same signature with a different name, `createProbs(String[],double[])`. This takes unnormalized probabilities on the linear scale, which must be non-negative, normalizes them to sum to 1, sorts them, then returns the corresponding conditional classifier.[3]

A conditional classifier is evaluated using a `ConditionalClassifierEvaluator`, which extends the scored classifier evaluator class. A conditional classifier provides average conditional probabilities as well as conditional one-versus-all evaluations.

9.2.5 Joint Classifiers and Classifications

In the case of generative statistical models, such as naive Bayes classifiers or hidden Markov models, the score represents the joint probabilty of the output category and the input being classified. Given the joint probabilities, we can use the rule of total probability to compute conditional probabilities of categories given inputs.

A joint classifier implements the interface `JointClassifier`, which extends `ConditionalClassifier`. The reason it can extend the conditional classifier class is that given joint probabilities, it's possible to calculate conditional probabilities, but not vice-versa. The joint classifier interface defines a classify method that returns a `JointClassification`, which extends the `ConditionalClassification` class.

The basic constructor is `JointClassification(String[],double[])`, where the first array is the array of categories and the second the array of log (base 2) joint probabilities. The arrays must be the same length and the probabilities must be in descending order. A second constructor, `JointClassification(String[],double[],double[])`, takes separate score and joint probability arrays in the same way as we saw for conditional classifiers. The joint probabilities are used to define conditional probabilities in the usual way.[4]

[2]The normalization is the usual softmax operation. With K categories with log probability scaled scores x_k, we set $\Pr(\text{cat} = k) \propto \exp(x_k)$, so that

$$\Pr(\text{cat} = k) = \frac{\exp(x_k)}{\sum_{k'=1}^{K} \exp(x_{k'})}.$$

[3]The normalization is the usual rescaling. With K categories with linear probability scaled scores y_k, we set $\text{pr}(\text{cat} = k) \propto y_k$, so that

$$\Pr(\text{cat} = k) = \frac{y_k}{\sum_{k'=1}^{K} y_{k'}}.$$

[4]If we have K categories with log (base 2) joint probabilities and w is the input, let $y_k =$

A joint classifier is evaluated using an instance of `JointClassifierEvaluator`, which extends the conditional classifier evaluator. Additional functionality includes joint probability evaluations. These are the traditional evaluation measures for joint statistical models and are based on the probabilty assigned to the training or test instances by the model.

9.3 Gold Standards, Annotation, and Reference Data

For classification and other natural language tasks, the categories assigned to texts are designed and applied by humans, who are notoriously noisy in the semantic judgments for natural language.

For instance, we made up the classification of genomics/non-genomics for MEDLINE citations. In order to gather evaluation data for a classifier, we would typically select a bunch of MEDLINE citations at random (or maybe look at a subset of interest), and label them as to whether they are about genomics or not.

At this point, we have a problem. If one annotator goes about his or her merry way and annotates all the data, everything may seem fine. But as soon as you have a second annotator try to annotate the same data, you will see a remarkable number of disagreements over what seem like reasonably simple notions, like being "about" genomics. For instance, what's the status of a citation in which DNA is only mentioned in passing? What about articles that only mention proteins and their interactions? What about an article on the sociology of the human genome project? These boundary cases may seem outlandish, but try to label some data and see what happens.

In the Message Understanding Conference evaluations, there were a large number of test instances involving the word *Mars* used to refer to the planet. The contestants' systems varied in whether they treated this as a location or not. The reference data did, but there weren't any planets mentioned in the training data.

9.3.1 Evaluating Annotators

The simplest way to compare a pair of annotations is to look at percentage of agreement between the annotators.

Given more than two annotators, pairwise statistics may be calculated. These may then be aggregated, most commonly by averaging, to get an overall agreement statistic. It is also worth inspecting the counts assigned to categories to see if any annotators are biased toward too many or too few assignments to a particular category.

$\log_2 \Pr(\mathrm{cat} = k, \mathrm{in} = w)$ be the log (base 2) joint probability of category k and input w. where c_k is a category and w_k is the object being classified, $\Pr(\mathrm{cat} = k|\mathrm{in} = w) \propto 2^{y_k}$ and we normalize as with any probabilities that are known up to a proportion, by

$$\Pr(\mathrm{cat} = k|\mathrm{in} = w) = \frac{2^{y_k}}{\sum_{k'=1}^{K} 2^{y_{k'}}}.$$

In general, pairs of annotators may be evaluated as if they were themselves classifiers. One is treated as the reference (gold standard) and one as the response (system guess), and the response is evaluated against the reference.

It is common to see Cohen's κ statistic used to evaluate annotators; see Section 9.4.3 for more information.

9.3.2 Majority Rules, Adjudication and Censorship

Data sets are sometimes created by taking a majority vote on the category for items in a corpus. If there is no majority (for instance, if there are two annotators and they disagree), there are two options. First, the data can be censored, meaning that it's removed from consideration. This has the adverse side effect of making the actual test cases in the corpus easier than a random selection might be because the borderline cases are removed. Perhaps one could argue this is an advantage, because we're not even sure what the categories are for those borderline cases, so who cares what the system does with them.

The second approach to disagreements during annotation is adjudication. This can be as simple as having a third judge look at the data. This will break a tie for a binary classification problem, but may not make the task any clearer. It may also introduce noise as borderline cases are resolved inconsistently.

A more labor intensive approach is to consider the cases of uncertainty and attempt to update the notion of what is being annotated. It helps to do this with batches of examples rather than one example at a time.

The ideal is to have a standalone written coding standard explaining how the data was annotated that is so clear that a third party could read it and label the data consistently with the reference data.

9.4 Confusion Matrices

One of the fundamental tools for evaluating first-best classifiers is the confusion matrix. A confusion matrix reports on the number of agreements between a classifier and the reference (or "true") categories. This can be done for classification problems with any number of categories.

9.4.1 Example: Blind Wine Classification by Grape

Confusion matrices are perhaps most easily understood with an example. Consider blind wine tasting, which may be viewed as an attempt by a taster to classify a wine, whose label is hidden, as to whether it is made primarily from the syrah, pinot (noir), or cabernet (sauvignon) grape. We have to assume that each wine is made primarily from one of these three grapes so the categories are exclusive and exhaustive. An alternative, binary classification problem would be to determine if a wine had syrah in it or not.

Suppose our taster works their way through 27 wines, assigning a single grape as a guess for each wine. For each of the 27 wines, we have assumed there is a true answer among the three grapes syrah, pinot and cabernet. The resulting

		Response			
		cabernet	*syrah*	*pinot*	
	cabernet	9	3	0	*12*
Reference	*syrah*	3	5	1	*9*
	pinot	1	1	4	*6*
		13	*9*	*5*	***27***

Fig. 9.1: *Confusion matrix for responses of a taster guessing the grape of 27 different wines. The actual grapes in the wines make up the reference against which the taster's responses are judged. Values within the table indicate the number of times a reference grape was guessed as each of the three possibilities. Italicized values outside the box are totals, and the bold italic 27 in the lower right corner is the total number of test cases.*

confusion matrix might look as in Figure 9.1. Each row represents results for a specific reference category. In the example, the first row represents the results for all wines that were truly cabernets. Of the 12 cabernets presented, the taster guessed that 9 were cabernets, 3 were syrah, and none were guessed to be a pinot. The total number of items in a row is represented at the end in italices, here *12*. The second row represents the taster's resuls for the 9 syrahs in the evaluation. Of these, 5 were correctly classified as syrahs, whereas 3 were misclassified as cabernets and one as a pinot. The last row is the pinots, with 4 correctly identified and 1 misclassified as cabernet and 1 as syrah. Note that the table is not symmetric. One pinot was guessed to be a cabernet, but no cabernets were guessed to be pinots.

The totals along the right are total number of reference items, of which there were 12 cabernets, 9 syrahs, and 6 pinots. The totals along the bottom are for responses. The taster guessed 13 wines were cabernets, 9 were syrahs, and 5 were pinots. The overall total in the lower right is 27, which is sum of the values in all the cells, and equal to the total number of wines evaluated.

Contingency matrices are particular kinds of contingency tables which happen to be square and have the same outcome labels on the rows and columns. In Section 9.8, we suvey the most popular techniques culled from a vast literture concerning the statistical analysis of contingency tables in general and confusion matrices in particular.

9.4.2 Classifier Accuracy and Confidence Intervals

The overall accuracy is just the number of correct responses divided by the total number of responses. A correct response for an item occurs when the response category matches the reference category. The count of correct responses are thus on the diagonal of the confusion matrix. There were 9 correct responses for reference cabernets, 5 for syrahs, and 4 for pinots, for a total of 18 correct responses. The total accuracy of the classifier is thus $18/27 \approx 0.67$.

Assuming that the reference items occur in the same distribution as they will in further test cases, the overall accuracy is an estimate of the probability that a classifier will make the correct categorization of the next item it faces. If further test data are distributed differently than our evaluation data, we can apply post-stratification, as described in Section 9.10.

We will also measure accuracy on a category-by-category basis, as we explain below when we discuss one-versus-all evaluations in Section 9.4.8.

Because overall accuracy is a simple binomial statistic, we can compute a normal approximation to a 95% confidence interval directly (see Section B.1.2 for an explanation of the formula). For our example, the 95% interval is approximately plus or minus 0.18 and the 99% interval approximately plus or minus 0.23.[5] As usual, with so few evaluation cases, we don't get a tight estimate of our system's accuracy.

In the case of classifier evaluations (and in many other situations in statistics), the width of confidence intervals, assuming the same distribution of results, is inversely proportional to the square root of the number of cases. For instance, to cut the size of confidence intervals in half, we need four times as much data. Concretely, if we had four times as much data in the same proportions as in our example, our 95% confidence intervals would be plus or minus 0.09.

9.4.3 κ Statistics for Chance-Adjusted Agreement

There is a family of statistics called κ statistics that are used to provide an adjustment to accuracy or agreement statistics based on how likely agreement would be "by chance". All of the κ statistics are defined in the same way, using the formula:

$$\kappa = \frac{a - e}{1 - e} \tag{9.1}$$

where a is a response's total accuracy, and e is accuracy achievable by guessing randomly. For example, if we have a system that has accuracy $a = 0.8$, if the chance accuracy is $e = 0.2$, then $\kappa = (0.8 - 0.2)/(1 - 0.2) = 0.75$.

In general, κ statistics are smaller than accuracy, $\kappa \leq a$, with equality only in the limit of zero random accuracy, $e = 0$.

The minimum possible value for κ is when the system is completely inaccurate, with $a = 0$, producing $\kappa = -e/(1-e)$, which works out to the negative odds for a randomly chosen item to be guessed correctly at random.

There are three forms of κ statistic in popular use, varying by the way in which e is defined. Accuracy a is always as we computed it in the previous section.

Cohen's κ

Cohen's κ assumes that the response is chosen at random based on the respondent's proportion of answers. For instance, in our runing example, we can read this off the bottom row, with the response being cabernet in 13/27 cases, syrah in 9/27 cases, and pinot 5/27 cases. If we assume that the reference values are chosen at random the same way, this time reading 12/27 cabernet, 9/27 syrah

[5]Because we're using a normal approximation, we might get restuls like 95% plus or minus 10%, leading to confidence intervals like [85%,105%], which extend beyond the possible values of a parameter like accuracy.

and 6/27 pinot from the right-most totals column. Thus our expected chance of agreement is

$$\left(\frac{13}{27} \times \frac{12}{27}\right) + \left(\frac{9}{27} \times \frac{9}{27}\right) + \left(\frac{5}{27} \times \frac{6}{27}\right) \approx 0.3663. \tag{9.2}$$

(We use so many decimal places here to compare two different versions of κ below.)

For Cohen's κ, the highest random accuracy e arises when the proportion of answers of each category match in the reference and response.

Siegel and Castellan's κ

A popular alternative to Cohen's version is Siegel and Castellan's κ, which uses the average of the reference and response proportions in place of the individual proportions to compute the chance accuracy.

In the Siegel and Castellan version of κ, we average the reference and response proportions. For instance, with a reference proportion of 12/27 and response proportion of 13/27 for cabernet, the average is 12.5/27. The calculation then becomes

$$\left(\frac{12.5}{27}\right)^2 + \left(\frac{9}{27}\right)^2 + \left(\frac{5.5}{27}\right)^2 \approx 0.3669. \tag{9.3}$$

In general, Siegel and Castellan's expected agreement will be higher than with Cohen's calculation.

Byrt, Bishop and Carlin's κ

If we assume our chance accuracy is a coin flip, with $e = 0.5$, we get Byrt et al.'s κ statistic, which works out to

$$\kappa = 2a - 1 \tag{9.4}$$

where a is the accuracy. The minimum value, for 0% accuracy, is -1, the maximum value, for 100% accuracy is 1, and the breakeven poin is 50% accuracy, yielding a κ value of 0.

9.4.4 Information-Theoretic Measures

We can use the empirical counts in a confusion matrix to estimate both the individual distributions of responses and references, as we used in the κ statistics, as well as their joint distribution, as represented by the entire confusion matrix. The basic definitions for information-theoretic concepts are reviewed in Section B.5.

With these estimates in hand, we can use any of the information-theoretic measures to consider the individual distributions or joint distributions. Specifically, we can compute the entropy of the reference or response distribution, or the conditional entropy of one given the other, or the cross entropy. We can also compute divergence statistics between the distributions.

Conditional Entropy

Of all the information-theoretic measures, we will focus on conditional entropy (see Section B.5.3), as it is the most informative for classification. It tells us how much information knowing the reference value gives us about the response.

In evaluating a classifier, we are really interested in how much the classifier reduces the undertainty in the outcome. Without the classfier, the uncertainty is measured by the entropy of the reference distribution, as calculated above. With the classifier in place, we need to measure the entropy of the outcome given the response of the classifier.

Let's focus on the response in the case of wines whose reference category is pinot. In those cases, of which there were 6, the classifier classified 1 as cabernet, 1 as syrah, and 4 as pinot. This corresponds to a discrete distribution with probabilities 1/6, 1/6 and 4/6. Calculating the entropy for this outcome, we get

$$\left(\frac{1}{6} \times \log_2 \frac{1}{6}\right) + \left(\frac{1}{6} \times \log_2 \frac{1}{6}\right) + \left(\frac{4}{6} \times \log_2 \frac{4}{6}\right) \approx 1.25. \tag{9.5}$$

We will refer to this value as the conditional entropy of the outcome given the reference category is pinot and observing the classifer's response. The value is 0.81 for cabernet and 1.35 for syrah. We then weight these values by the reference counts of each of these categories, and divide by the total number of counts, to produce

$$\frac{1}{27}\left((12 \times 0.81) + (9 \times 1.35) + (6 \times 1.25)\right) = 1.09 \tag{9.6}$$

The units are bits, and as usual, they're on a log scale. A value of 1.09 bits means that we've reduced the three-way decision to a $2^{1.09}$, or 2.1-way decision.

We can read the reference entropy the same way. Its value is 1.53, meaning that the base classification problem is effectively a $2^{1.53}$ or 2.9-way decision. The value would work out to a 3-way decision if the probabilty of seeing a cabernet test case was the same as for seeing a pinot or syrah test case. The value lower than 3 arises because of the 12/9/6 distribution of the data. If it were more skewed, entropy would be lower.

Mutual Information

A related measure to conditional entropy is mutual information (see Section B.5.4). It measures the difference in entropy in the entire table between guessing the response category randomly and guessing it based on the conditional distribution.

9.4.5 χ^2 Independence Tests

Pearson's χ^2 independence test is for the hypothesis that the the reference and response categories for an item were assigned independently (see Section B.4.1). In the blind-wine tasting case, this would correspond guessing the wine before tasting it by simply generating a random category based on the taster's distribution of answers. This is the same kind of independent reference and response generation that underlies the expected correct answers by chance used in the definition of Cohen's κ statistic (see Section 9.4.3) and mutual information (see Section 9.4.4).

Pearson's X^2 statistic (see Section B.4.1) is used for the χ^2 test, with higher values of X^2 being less likely to have arisen from random guessing. In our running example, the value of the statistic is 15.5, which is quite high.

To compute p values, we need to determine where X^2 falls in the cumulative distribution of χ^2_ν, where ν is the degrees of freedom (here $(K-1)^2$ for K categories). For our example, the chi square statistic X^2 is 15.5. The R function function `phchisq()` for the cumulative density of χ^2 may be used to compute p-values by:

```
> 1 - pchisq(15.5,df=4)

[1] 0.003768997
```

The way to interpret a p value is as the probability under the null hypothesis (here independence) of seeing as extreme a value as was seen (here 15.5) in the distribution at hand (here the χ^2 distribution with 4 degrees of freedom). In other words, our p value of 0.0038 means that there is less than a 0.4% chance that two independent distributions were in as good agreement as what we observed in our table. This is usually considered a low enough p value to reject the null hypothesis that the two results were independent.[6]

LingPipe is also able to report Pearson's mean-square contingency statistic φ^2, Cramér's V statistic of association, and Goodman and Kruskal's index of predictive association, all of which are defined in Section 9.8.2

9.4.6 The `ConfusionMatrix` Class

A confusion matrix is represented in LingPipe as an instance of the class `ConfusionMatrix`, which is in package `com.aliasi.classify`.

Constructing a Confusion Matrix

A confusion matrix minimally requires an array of categories for construction, with constructor `ConfusionMatrix(String[])`. It may optionally take a matrix of counts, with constructor `ConfusionMatrix(String[],int[][])`.

Getters and Increments

We can retrieve (a copy of) the categories as an array using `categories()`. The evaluation keeps a symbol table under the hood (see Chapter 5) to support mappings from categories to their indices. This is available indirectly through the method `getIndex(String)`, which returns the index for a category in the array of category names or -1 if there is no such category.

[6]We never accept the null hypothesis. With a high p value we simply fail to reject it.

It's also possible to retrieve (a copy of) the entire matrix, using `matrix()`, which returns an `int[][]`.

We can get the count in a particular cell using `count(int,int)`, where the first index is for the reference category. If we want to get the count by category, we can use the `getIndex()` method first.

We can increment the values in the confusion matrix by reference and response category, or by index. The method `increment(int,int)` takes a reference and response category index and increments the counts by one. The method `increment(String,String)` takes the names of the categories. There is also a method `incrementByN(int,int,int)`, which takes two category indices and an amount by which to increment.

Often classifiers do not perform equally well for all categories. To help assess performance on a per-category basis, we can look at the category's row or column of the confusion matrix. The confusion matrix class also supports one-versus-all evaluations, as discussed in Section 9.4.8.

9.4.7 Demo: Confusion Matrices

In the rest of this section, we will illustrate the use of confusion matrices to compute the statistics defined earlier in this section. The demo is in class `ConfusionMatrixDemo`.

Code Walkthrough

The work is all in the `main()` method, which starts by allocating the contents of the confusion matrix and then the matrix itself, with data corresponding to our running blind-wine classication example.

```
String[] cats = new String[] {
     "cabernet", "syrah", "pinot"
};

int[][] cells = new int[][] {
     { 9, 3, 0 },
     { 3, 5, 1 },
     { 1, 1, 4 }
};

ConfusionMatrix cm = new ConfusionMatrix(cats, cells);
```

After that, it's just a bunch of getters and assigns, which we will print at the enc of the code. First, the categories, total count, total correct and accuracies, with confidence intervals,

```
String[] categories = cm.categories();

int totalCount = cm.totalCount();
int totalCorrect= cm.totalCorrect();
double totalAccuracy = cm.totalAccuracy();
double confidence95 = cm.confidence95();
double confidence99 = cm.confidence99();
```

Next, the three versions of the κ statistic,

```
double randomAccuracy = cm.randomAccuracy();
double randomAccuracyUnbiased = cm.randomAccuracyUnbiased();
double kappa = cm.kappa();
double kappaUnbiased = cm.kappaUnbiased();
double kappaNoPrevalence = cm.kappaNoPrevalence();
```

Then the information theoretic measures,

```
double referenceEntropy = cm.referenceEntropy();
double responseEntropy = cm.responseEntropy();
double crossEntropy = cm.crossEntropy();
double jointEntropy = cm.jointEntropy();
double conditionalEntropy = cm.conditionalEntropy();
double mutualInformation = cm.mutualInformation();
double klDivergence = cm.klDivergence();

double conditionalEntropyCab = cm.conditionalEntropy(0);
double conditionalEntropySyr = cm.conditionalEntropy(1);
double conditionalEntropyPin = cm.conditionalEntropy(2);
```

Finally, we have the χ^2 and related statistics and the λ statistics,

```
double chiSquared = cm.chiSquared();
double chiSquaredDegreesOfFreedom
     = cm.chiSquaredDegreesOfFreedom();
double phiSquared = cm.phiSquared();
double cramersV = cm.cramersV();

double lambdaA = cm.lambdaA();
double lambdaB = cm.lambdaB();
```

All of these statistics are explained in Section 9.8.

Running the Demo

The Ant target `confusion-matrix` runs the demo. There are no command-line arguments, so there's nothing else to provide.

```
> ant confusion-matrix

categories[0]=cabernet
categories[1]=syrah
categories[2]=pinot

totalCount=27
totalCorrect=18
totalAccuracy=0.6666666666666666
confidence95=0.17781481095759366
confidence99=0.23406235319928148
```

	Resp	
Ref	*cab*	*not*
cab	9	3
not	4	11

	Resp	
Ref	*syrah*	*not*
syrah	5	4
not	4	14

	Resp	
Ref	*pinot*	*not*
pinot	4	2
not	1	20

Fig. 9.2: *The one-versus-all confusion matrices defined by joining categories of the* 3×3 *confusion matrix shown in Figure 9.1.*

```
randomAccuracy=0.36625514403292175
randomAccuracyUnbiased=0.3669410150891632
kappa=0.4740259740259741
kappaUnbiased=0.4734561213434452
kappaNoPrevalence=0.33333333333333326

referenceEntropy=1.5304930567574824
responseEntropy=1.486565953154142
crossEntropy=1.5376219392005763
jointEntropy=2.619748965432189
conditionalEntropy=1.089255908674706
mutualInformation=0.39731004447943596
klDivergence=0.007128882443093773
conditionalEntropyCab=0.8112781244591328
conditionalEntropySyr=1.3516441151533922
conditionalEntropyPin=1.2516291673878228

chiSquared=15.525641025641026
chiSquaredDegreesOfFreedom=4.0
phiSquared=0.5750237416904084
cramersV=0.5362013342441477

lambdaA=0.4
lambdaB=0.35714285714285715
```

9.4.8 One-Versus-All Evaluations

We often want to evaluate the performance of a classifier on a single reference category. LingPipe's confusion matrix classes support this kind of single-category evaluation by projecting a confusion matrix down to a 2×2 confusion matrix comparing a single category to all other categories. Because our categories are exhaustive and exclusive, the reduced matrix compares a single category to its complement.

9.4.9 Example: Blind Wine Classification (continued)

Going back to our sample confusion matrix in Figure 9.1, we can render the results for the three different wines into the following three 2×2 confusion matrices, which we show in Figure 9.2. The one number that stays the same is the

	Resp	
Ref	*positive*	*negative*
positive	18	9
negative	9	45

Fig. 9.3: *The micro-averaged confusion matrix resulting from summing the one-versus-all matrices in Figure 9.2.*

number of correct identifications of the category. For instance, for syrah, the taster had 9 correct identifications. Thus there is a 9 in the cabernet-cabernet cell of the cabernet-versus-all matrix. The count for cabernet reference and non-cabernet response sums the two original cells for cabernet reference/syrah response and cabernet reference/pinot response, which were 3 and 0, yielding a total of 3 times that a cabernet was misclassified as something other than cabernet. Similarly, the cell for non-cabernet reference and cabernet response, representing non-cabernets classified as cabernets, is 4, for the 3 syrahs 1 pinot and the taster classified as cabernets. All other values go in the remaining cell of not-cabernet reference and not-cabernet response. The total is 11, being the sum of the syrah reference/syrah response, syrah reference/pinot response, pinot reference/syrah response, and pinot reference/pinot response. Note that this includes both correct and incorrect classifications for the original three-way categorization scheme. Here, the 11 classifications of non-cabernets as non-cabernets are treated as correct. The other two tables, for syrah-versus-all and pinot-versus-all are calculated in the same way.

9.4.10 Micro-Averaged Evaluations

Now that we have a 2×2 one-versus-all matrix for each category, we may sum them into a single matrix, which will be used for what are known as micro-averaged evaluations. The 2×2 micro-averaged confusion matrix for our running blind-wine tasting example is given in Figure 9.3. The micro-average confusion matrix is closely related to our original matrix (here Figure 9.1). The total count will be the number of categories (here 3) times the number of test cases (here 27, for a total count of 81). The number of true positives is just the number of correct classifications in the original matrix (here 18, for the three diagonal elements $9 + 5 + 4$). The number of false positives and false negatives will be the same, and each will be equal to the sum of the off-diagonal elements, here $3 + 1 + 1 + 3 + 0 + 1$). The true negative slot gets the rest of the total, which is just triple the original count (here $3 \times 27 = 81$) minus the number of correct answers in the original matrix (here 18) minus twice the number of errors in the original matrix (here $2 \times 9 = 18$, for a total of 45).

9.4.11 The `ConfusionMatrix` Class (continued)

The method `oneVsAll(int)` in `ConfusionMatrix` returns the one-versus-all statistics for the category with the specified index (recall that `getIndex(String)`

	Response	
Reference	*positive*	*negative*
positive	9 (TP)	3 (FN)
negative	4 (FP)	11 (TN)

Fig. 9.4: *A* 2 × 2 *confusion matrix for precision-recall analysis. In general, the rows are marked as references and the columns as reponses. The two categories are "positive" and "negative", with "positive" conventionally listed first.*

converts a category name to its index). The method `microAverage()` returns the micro-averaged confusion matrix. For our example, these are the three one-versus-all confusion matrices shown Figure 9.2 and the micro-averaged matrix shown in Figure 9.3.

These statistics are returned as an instance of `PrecisionRecallEvaluation`, which is also in the package `com.aliasi.classify`. A precision-recall matrix implements many of the same statistics as are found in a general confusion matrix, as well as a number of methods specific to 2×2 matrices with a distinguished positive and negative category. We turn to these in the next section.

9.5 Precision-Recall Evaluation

Precision-recall evaluations are based on a 2 × 2 confusion matrix with one category distinguished as "positive" and the other "negative" (equivalently "accept"/"reject", "true"/"false", etc.). As with confusion matrices, we distinguish the rows as the reference and the columns as the response.

For instance, in a search problem, the relevant documents are considered positive and the irrelevant ones negative. In a named entity chunking problem, spans of text that mention person names are positive and spans that don't mention person names are negative. In a blood test for a disease, having the disease is positive and not having the disease negative.

9.5.1 Example: Blind-Wine One-versus-All

An example of a confusion matrix suitable for precision-recall evaluations is shown in Figure 9.4. This is exactly the same matrix as shown in the cabernet-versus-all classifier in Figure 9.2. We have also added in names for the four distinguished slots. In the case the response is positive and the reference is positive, we have a true positive (TP). If the response is positive but the reference negative, we have a false positive (FP). Similarly, a negative response and negative reference is a true negative (TN) case, whereas a negative response with positive reference is a false negative (FN). So the second letter is "N" or "P" depending on whether the response is negative or positive. There are 13 positive responses and 14 negative responses, whereas there are 12 items that are actually positive in the reference and 15 that are actually negative.

9.5.2 Precision, Recall, Etc.

We call these special 2×2 confusion matrices precision-recall matrices because precision and recall are the most commonly reported statistics for natural language processing tasks.

The recall (or sensitivity) of a classifier is the percentage of positive cases for which the response is positive. The positive cases are either true positives (TP) if the response is positive or false negatives (FN) if the response is negative. Thus recall is given by

$$\text{rec} = \text{sens} = TP/(TP + FN). \tag{9.7}$$

In the example in Figure 9.4, TP=9 and FN=3, so recall is $9/(9+3) = 0.75$. Thus 75% of the positive cases received a positive response.

The specificity of a classifier is like sensitivity, only for negative cases. That is, specificity is the percentage of negative cases for which the response is negative. the negative cases are either true negatives (TN) if the response is negative or false positives (FP) if the response is positive. Specificity is thus defined to be

$$\text{spec} = TN/(TN + FP). \tag{9.8}$$

In Figure 9.4, TN=11 and FP=4, so specificity is $11/(11+4) \approx 0.73$.

Precision is not based on accuracy as a percentage of true cases, but accuracy as a percentage of positive responses. It measures how likely a positive response is to be correct, and in thus also known as (positive) predictive accuracy. Precision is defined by

$$\text{prec} = TP/(TP + FP). \tag{9.9}$$

In our example, TP=9, FP=4, so precision is $9/(9+4) \approx 0.69$. What counts for an error for precision is a false positive, whereas an error for recall is a false negative. Both treat true positives as positives.

The final member of this quartet of statistics is known as selectivity or negative predictive accuracy. Selectivity is like precision, only for negative cases. That is, it is the percentage of the false responses that are correct. The definition is

$$\text{selec} = TN/(TN + FN). \tag{9.10}$$

9.5.3 Prevalence

Another commonly reported statistic for precision-recall matrices is prevalence, which is just the percentage of positive reference cases among all cases. This is defined by

$$\text{prev} = (TP + FN)/(TP + FP + TN + FN) \tag{9.11}$$

Note that false negatives are reference positive cases for which the system returned a negative response.

If we know the prevalence of true cases as well as a classifier's sensitivity and specificity, we can reconstruct the entire confusion matrix,

$$\text{TP} = \text{prev} \times \text{sens}, \qquad \text{FN} = \text{prev} \times (1 - \text{sens}), \tag{9.12}$$

$$\text{FP} = (1 - \text{prev}) \times (1 - \text{spec}), \text{ and} \qquad \text{TN} = (1 - \text{prev}) \times \text{spec}. \tag{9.13}$$

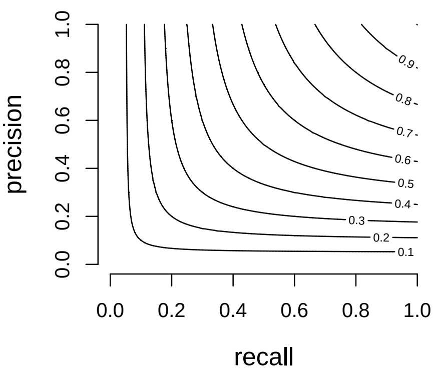

Fig. 9.5: *Contour plot of F measure as a function of recall and precision. Each contour line indicates a particular F measure value as marked. For any given precision and recall, we can see roughly the result F measure by which contour lines they're between. For instance, tracing a line up from 0.4 recall to 0.6 precision leaves us roughly equidistant between the* $F = 0.4$ *and* $F = 0.5$ *contour lines.*

9.5.4 F Measure

It is common to combine precision and recall into a single number. The traditional way to do this is using the (balanced) harmonic mean, which in the case of precision and recall, is known as F measure. The definition is

$$F = \frac{2 \times \text{prec} \times \text{rec}}{\text{rec} + \text{prec}}. \tag{9.14}$$

In Figure 9.5, we show a contour plot of F measure as a function of recall and precision.

Although precision and recall are the most commonly reported pair of statistics for natural language classification tasks, neither of them involve the true negative count. That is, they are only based on the TP, FP and FN counts, which involve either a positive reference category or positive response.

9.5.5 The `PrecisionRecallEvaluation` Class

The LignPipe class `PrecisionRecallEvaluation`, from the package `com.aliasi.classify`, encapsulates a precision-recall matrix and provides methods for the common statistics described in this section as well as the confusion matrix methods detailed the previous section.

Construction and Population

There are two constructors for a precision-recall evaluation instance. The first, `PrecisionRecallEvaluation()`, starts with zero counts in all cells, whereas `PrecisionRecallEvaluation(long,long,long,long)` starts with the specified number of TP, FN, FP, and TN counts respectively.

Once a precision-recall evaluation instance is constructed, the method addCase(boolean,boolean) adds another test case, where the first boolean argument represents the reference category and the second the response category. Positive responses are coded as boolean true and negatives as false. Thus calling addCase(true,true) adds a true positive, addcase(true,false) a false negative, addcase(false,false) a true negative, and addcase(false,true) a false positive.

Counts and Statistics

As for the confusion matrix implementation, the raw counts in the cells as well as row- and column-wise sums are available directly. The cell coutns are available through methods truePositive(), falsePositive(), trueNegative() and falseNegative().

The basic statistics are also named in the obvious way, with the three main methods being precision(), recall(), and fMeasure(). The precision-recall evaluation class ues the terms rejectionRecall() for specificity and rejectionPrecision() for selectivity, pointing out their duality with precision and recall by swapping positive/negative category naming.

The row totals are availabe as positiveReference() and negativeReference(), indicating the balance of positive and negative cases in the reference (often the test) set. Column totals for responses are positiveResponse() and negativeResponse(). The total number of items, that is the sum of all cells, is available through total().

The proportion of positive cases in the reference set is given by referenceLikelihood(), and the corresponding statistic for responses, namely the proportion of positive responses, as responseLikelihood().

All of the other statistics are defined in exactly the same way as for confusion matrices.

Demo: Precision-Recall Evaluations

We provide a demo of precision-recall evaluations in the class PrecisionRecallDemo. We'll use the one-versus-all classification matrix as we provided as a demo in Figure 9.2, which is the same as the result of running the confusion matrix demo in Section 9.4.7 and retrieving the one-versus-all evaluation for cabernet.

All of the work's done in the main() method, which accepts four arguments, representing the TP, FN, FP and TN counts respectively. These are parsed as long values and assigned to variables tp, fn, fp, and tn respectively. At this point, we can construct our evaluation and inspect the counts assigned.

```
PrecisionRecallEvaluation eval
    = new PrecisionRecallEvaluation(tp,fn,fp,tn);

long tpOut = eval.truePositive();
long fnOut = eval.falseNegative();
long fpOut = eval.falsePositive();
long tnOut = eval.trueNegative();

long positiveRef = eval.positiveReference();
long positiveResp = eval.positiveResponse();

long total = eval.total();
```

We may then retrieve the various precision- and recall-like statistics from the evaluation, including *F* measure (and prevalence).

```
double precision = eval.precision();
double recall = eval.recall();
double specificity = eval.rejectionRecall();
double selectivity = eval.rejectionPrecision();

double fMeasure = eval.fMeasure();

double prevalence = eval.referenceLikelihood();
```

Finally, we pull out some of the statistics that are available; the rest are defined in the Javadoc and explained in the confusion matrix demo.

```
double accuracy = eval.accuracy();
double accuracyStdDev = eval.accuracyDeviation();

double kappa = eval.kappa();

double chiSq = eval.chiSquared();
double df = 1;
```

We can run the demo using the Ant target `precision-recall`. It provides the values of properties `tp`, `fn`, `fp`, and `tn` as the first four arguments. For instance, we can call it with the values from the syrah-versus-all evaluation.

```
> ant -Dtp=9 -Dfn=3 -Dfp=4 -Dtn=11 precision-recall

tpOut=9    fnOut=3    fpOut=4    tnOut=11
positiveRef=12    positiveResp=13    total=27
precision=0.692    recall=0.75    fMeasure=0.719
specificity=0.733    selectivity=0.785    prevalence=0.444
accuracy=0.740    accuracyStdDev=0.084
kappa=0.479    chisq=6.23
```

We've reduced the number of digits by truncation and removed some line breaks to save space in the book.

9.6 Micro- and Macro-Averaged Statistics

So far, in considering confusion matrices, we only considered overall evaluation measures, such as accuracy, κ statistics, and χ^2 independence tests. In this section, we consider two ways of evaluating general classification results on a more category-by-category basis.

9.6.1 Macro-Averaged One Versus All

In our treatment of confusion matrices, we showed how the 3×3 confusion matrix in Figure 9.1 could be reduced the three 2×2 confusion matrices, shown in Figure 9.2.

Suppose take each of these 2×2 confusion matrices and consider its precision, recall and F measure. For instance, precision for the cabernet matrix is approximately 0.69, for syrah approximately 0.56, and for pinot noir 0.67. The macro-averaged precision is then given by averaging these values, the result of which is 0.64. We can similarly macro average any of the precision-recall evaluation statistics. Most commonly, we see macro-averaged values for precision, recall, and F measure.

Macro averaging has the property of emphasizing instances in low-count categories more than in high-count categories. For instance, even though there are only 6 actual pinots versus 12 actual cabernets, recall on the 6 pinots is weighted the same as recall on the 12 cabernets. Thus the pinot recall is weighted twice as heavily on a per-item basis.

Macro averaging is often used for evaluations of classifiers, with the intent of weighting the categories equally so that systems can't spend all their effort on high-frequency categories.

9.6.2 Micro-Averaged One Versus All

Micro-averaging is like macro-averaging, only instead of weighting each category equally, we weight each item equally. In practice, we can compute micro-averaged results over the 2×2 precision-recall matrix produced by summing the three one-versus-all precision-recall matrices. We showed an example of this in Figure 9.3, which is the result of simply adding up the three one-versus-all evaluations in Figure 9.2.

When we sum one-versus-all matrices, we wind up with a matrix with a number of true positives equal to the number of elements on the original matrix's diagonal. That is, the true positives in the micro-averaged matrix are just the correct answers in the original matrix, each of which shows up as a true positive in exactly one of the one-versus-all matrices (the one with its category).

In the micro-averaged confusion example in Figure 9.3, we have 18 true positives and 9 false positives, for a precision of exactly 2/3. Recall is also exactly 2/3. And so is F measure.

Micro-averaged results are often reported along with macro-averaged results for classifier evaluations. If the evaluators care about ranking classifiers, this provides more than one way to do it, and different evaluatiosn focus on different measures.

9.6.3 Demo: Micro- and Macro-Averaged Precision and Recall

A demo for micro- and macro-averaged results is in the class `MicroMacroAvg`. The code is all in the `main()` method, which expects no arguments. It starts by setting up a confusion matrix exactly as in our confusion matrix demo, so we don't show that. The calculation of micro- and macro-averaged results is as follows.

```
ConfusionMatrix cm = new ConfusionMatrix(cats, cells);

double macroPrec = cm.macroAvgPrecision();
double macroRec = cm.macroAvgRecall();
double macroF = cm.macroAvgFMeasure();

PrecisionRecallEvaluation prMicro = cm.microAverage();
double microPrec = prMicro.precision();
double microRec = prMicro.recall();
double microF = prMicro.fMeasure();
```

The macro-averaged results are computed directly by the confusion matrix class. The micro-averaged results are derived by first extracting a precision-recall matrix for micro-averaged results, then calling its methods.

For convenience, we also extract all the precision-recall evaluations on which the macro-averaged results were based and extract their precision, recall and F measures.

```
for (int i = 0; i < cats.length; ++i) {
     PrecisionRecallEvaluation pr = cm.oneVsAll(i);
     double prec = pr.precision();
     double rec = pr.recall();
     double f = pr.fMeasure();
```

We do not show the print statements.

The demo may be run from Ant using the target `micro-macro-avg`.

```
> ant micro-macro-avg

cat=cabernet prec=0.692 rec=0.750 F=0.720
cat=   syrah prec=0.556 rec=0.556 F=0.556
cat=   pinot prec=0.800 rec=0.667 F=0.727
Macro prec=0.683 rec=0.657 F=0.668
Micro prec=0.667 rec=0.667 F=0.667
```

From these results, it's also clear that we can recover the micro-averaged results by taking a weighted average of the one-versus-all results. For instance, looking at recall, we have one-versus-all values of 0.750 for cabernet, 0.556 for syrah, and 0.667 for pinot. If we average these results weighted by the number of instances (12 for cabernet, 9 for syrah, 6 for pinot), we get the same result,

$$0.667 = \frac{(12 \times 0.750) + (9 \times 0.556) + (6 \times 0.667)}{12 + 9 + 6}. \tag{9.15}$$

In most evaluations, micro-averaged results tend to be better, because performance is usually better on high-count categories than low-count ones. Macro-averaged results will be higher than micro-averaged ones if low count categories perform relatively better than high count categories.

9.6.4 Generalized Precision-Recall Evaluations for Information Extraction

Precision-recall evaluations are applicable to broader classes of problems than simple classifiers. In fact, the most popular applications of precision-recall evaluations are to information-finding problems. A typical example of this is named entity mention recognition. A named-entity mention is a sequence of text that refers to some object in the real world by name. For example, *Detroit* might refer to a city, *Denny McClain* to a person, and *Tigers* to an organization in the sentence *Denny McClain played for the Tigers in Detroit*.

Suppose we have a reference corpus of named entity mentions. We evaluate a named-entity mention chunking system by running it over the text and returning all the entity mentions found by the system. Given the reference answers, it's easy to classify each mention returned by the system as a true positive if the response matches a mention in the reference or a false positive if the response doesn't match a mention in the reference. Then, each mention in the reference that's not in the response is marked as a false negative. With the TP, FP, and FN statistics, it's straightforward to compute precision and recall.

We have so far said nothing about true negatives. The problem with true negatives for a named entity mention finding task is that any span of text may potentially refer to any type of entity, so there are vast numbers of true negatives. For instance, *played*, *McClain played*, *played for*, *McClain played for*, and so on, are all true negatives in our example for any type of entity to which they're assigned.

This explains why precision is a more popular measure of information extraction systems than specificity. For specificity, we need to know about the true negatives, whereas precision and recall are based only on the combined set of positive reference items and positive response items.

Similar precision-recall evaluations are applied to information retrieval (IR) systems. In the standard Cranfield evaluation model[7] for evaluating IR systems, a test (or reference) corpus consists of a sequence of queries and for each query, the set of relevant documents from a collection. A system then returns a number of documents for a query. Each of these returned document will be either a true positive (if it's marked as relevant in the reference corpus) or a false positive (if it's not marked as relevant). The relevant documents not returned are false negatives and the rest are true negatives. These evaluations are often extended to systems that provide ranked and scored results, as we discuss in the next section.

9.7 Scored Precision-Recall Evaluations

So far, we have assumed that the system's response to a classification problem is a single first-best guess at a category. Many different kinds of classifiers support returning more than one guess, typically with a score for each result determining

[7]Developed around the Cranfield test collection of 1400 documents in aeronautics in the 1960s.

a rank order. In many cases, the results are associated with conditional probability estimates of the response category given the item being classified; in other cases, scores have a geometric interpretation, such as the margin from the classification hyperplane for a perceptron or the cosine for TF/IDF rankings.

9.7.1 Ranks, Scores and Conditional Probability Estimates

For instance, given our blind-wine grape-guessing task, a taster may be presented with a wine, and respond that their first guess is cabernet, their second guess syrah, and their third guess pinot noir. That's a ranked response. In addition, they might further specify that cabernet has a score of -1.2, syrah a score of -1.3, and pinot a score of -2.0 (yes, scores may be negative; they still get sorted from largest, here -1.2, to smallest, -2.0, for purposes of ranking). That's what we call a scored response.

An even more refined response contains a conditional probability estimate of the output category given the input, such as 55% cabernet, 35% syrah, and 10% pinot noir. That's a conditional probability response.[8]

9.7.2 Comparing Scores across Items

Ranked results support ranking categories within the response to a single item being classified, such as a single wine. For instance, I may be guessing the grapes in two wines, and rank the first one syrah, cabernet, and pinot, and the second one pinot, syrah, and cabernet. Without scores, there's no way to tell if the taster is more confident about the guess of syrah for the first wine or pinot for the second wine.

With scores, we are sometimes able to compare a classifier's response across items. Usually we can't, though, because the scores will only be calibrated among the category responses for a single item. For instance, it doesn't usually make sense to compare scores for different documents across different searches in an information retrieval system like Lucene (see the chapter on Lucene in the companion volume, *Text Processing in Java*), because of all the document- and query-specific effects built into scoring.

Conditional probabilities, on the other hand, are naturally calibrated to be comparable across runs. If I have two wines, and for the first one I respond 55% cabernet, 35% syrah and 10% pinot, and for the second, 90% cabernet, 8% syrah, and 2% pinot, I can rank my choices. My best guess is that wine 2 is a cabernet, because I've estimated a 90% chance of being right. My second best guess is that wine 1 is a cabernet (55%), my third-best guess that wine 1 is a syrah (35%), my fourth-best guess that wine 1 is pinot (10%), my fifth-best that wine 2 is syrah (8%), and my least likely guess is that wine 2 is a pinot (2%).

In this section, we will assume that we're evaluating system responses in a situation where the scores (or conditional probabilities) are calibrated across different items. This lets us arrange all of our guesses for all of our items in order.

[8]There are also joint probability responses, from which conditional probabilities can be derived. The joint probabilities are less useful for evaluation, so we put off considering them until we see a classifier that uses them.

It is this ordering, an example of which we saw in the last paragraph, that forms the basis for scored precision recall evaluations.

9.7.3 The `ScoredPrecisionRecallEvaluation` Class

The LingPipe class `ScoredPrecisionRecallEvaluation`, in package `com.aliasi.classify`, implements precision-recall type statistics based on scored results.

Construction and Population

Most often, like precision-recall evaluations, scored precision-recall evaluations will be constructed automatically by encapsulated testing classes for each kind of system. LingPipe generates scored precision-recall evaluations when given appropriately probabilistic taggers, chunkers, spelling correctors, or classifiers. For now, we consider direct construction and manipulation to get a feeling for the evaluation class itself.

There is a single no-arg constructor, `ScoredPrecisionRecallEvaluation()`. The evaluation initially starts out with no data.

Responses are added to the evaluation using the method `addCase(boolean,double)`, which takes two arguments, the first being whether the result was correct, and second what the score was. Note that we are not keeping track of the individual categories, just their scores and whether they are correct or not.[9]

Going back to our example, of the previous section, suppose the first wine was a syrah, and the second a cabernet. Here we repeat the responses along with an indication of whether the response was true or not. For the first wine, we have 55% cabernet (false), 35% syrah (true), and 10% pinot noir (false), and for the second wine, 90% cabernet (true), 8% syrah (false), and 2% pinot noir (false). To populate the evaluation with these results requires six calls to `addCase()`, `addCase(false,0.55)`, `addCase(true,0.35)`, and `addCase(false,0.10)` for the three responses for the first wine, and similarly for the three results from the second wine.

If the responses are exhaustive for each item, this is all we have to do. The key observation is that the correct answers will show up in every set of responses. Unfortunately, this is often not the case. For instance, consider a named-entity chunker that returns the 100-best potential entity mentions for each sentence. If we know the true entity mentions, we can rank each of these results as true or false, based on whether the entity mention is in the reference standard or not. But there's no guarantee that the system will find all the entities in a sentence, even if it returns its 100-best guesses.

In the case where not every positive reference item is included in the system response, the method `addMisses(int)` lets you add in the number of positive reference items that were missed by the system's responses. For instance, suppose we have the following sentence, marked for named entities.

[9] To keep track of the actual items, use the basic precision-recall evaluation.

```
Johan van der Dorp took the train to den Haag.
01234567890123456789012345678901234567890123456
0         1         2         3         4
(0,18):PER   (37,45):LOC
```

The entities are marked by character offsets, with `(0,18):PER` indicating that the text spanning from character 0 (inclusive) to character 18 (exclusive), namely *Johan van der Dorp*, mentions a person. Simiarly, the other entity indicates that *den Haag* refers to a location. This is a tricky sentence for a system trained on English data without an extensive list of names or locations, because both names contain tokens that are not capitalized.

Given the input text, a named-entity mention chunker might return all found chunks with estimated probability above 0.01, which might lead to the following six chunks, for example.

```
( 4, 8):LOC  0.89     (41,45):ORG  0.43     (41,45):LOC  0.09
( 0, 5):PER  0.46     (41,45):PER  0.21     ( 0,18):PER  0.02
```

In this case, only the guess `(0,18):PER` is correct, with the other five being wrong. We call `addScore()` for each response, including the probability estimate and correctness, the latter of which is false for the first five responses and true for the last one. After adding the scores, we must call `addMisses(1)`, indicating that one of the reference entities, namely `(37,45):LOC`, was not included among the system responses.[10]

To evaluate information retrieval results, the system is used in the same way, with the method `addScore()` being called for each system response and `addMisses()` for the results that the system didn't return.

9.7.4 Demo: Scored Precision-Recall

We provide a demo of the scored precision-recall evaluation class in `ScoredPrecisionRecallDemo`. All of the work is in the `main()` method, which has no command-line arguments.

Construction and Population of Results

The `main()` method starts by constructing an evaluation and populating it.

```
ScoredPrecisionRecallEvaluation eval
     = new ScoredPrecisionRecallEvaluation();

eval.addCase(false,-1.21); eval.addCase(false,-1.80);
eval.addCase(true,-1.60); eval.addCase(false,-1.65);
eval.addCase(false,-1.39); eval.addCase(true,-1.47);
eval.addCase(true,-2.01); eval.addCase(false,-3.70);
eval.addCase(true,-1.27); eval.addCase(false,-1.79);

eval.addMisses(1);
```

[10]These low-level adds, for both found chunks and misses, is automated in the chunker evaluation classes. We're just using named entity spotting as an example here.

Rank	*Score*	*Correct*	*TP*	*TN*	*FP*	*FN*	*Rec*	*Prec*	*Spec*
0	−1.21	no	0	5	1	5	0.00	1.00	0.83
1	−1.27	**yes**	1	5	1	4	0.20	0.50	0.83
2	−1.39	no	1	4	2	4	0.20	0.33	0.67
3	−1.47	**yes**	2	4	2	3	0.40	0.50	0.67
4	−1.60	**yes**	3	4	2	2	0.60	0.60	0.67
5	−1.65	no	3	3	3	2	0.60	0.50	0.50
6	−1.79	no	3	2	4	2	0.60	0.43	0.33
7	−1.80	no	3	1	5	2	0.60	0.38	0.17
8	−2.01	**yes**	4	1	5	1	0.80	0.44	0.17
9	−3.70	no	4	0	6	1	0.80	0.40	0.00
10	miss	**yes**	5	0	6	0	1.00	0.00	0.00

Fig. 9.6: *Example of a scored precision-recall evaluation. The responses are shown with their correctness, sorted by score. There is a single missed positive item for which there was no response, so we include it at the bottom of the table without a score. In total, the number of positive cases is 5 and the total number of negative cases is 5. The ranks, numbered from 0, are determined by score. The counts of TP, TN, FP and FN and the corresponding recall* $(TP/(TP+FN))$*, precision* $(TP/(TP+FP))$*, and specificity* $(TN/(TN+FP))$ *values are shown.*

After construction, we add ten scored results, four of which are true positives and six of which are false positivies. We then add a single miss representing a reference positive that was not returned by the system.

Statistics at Ranks

The evaluation is most easily understood by considering the sorted table of evaluated responses, which may be found in Figure 9.6. The first column of the table indicates the rank of the response (numbering from zero), the second the score of the response, and the third whether it was a correct (true) response or not. For example, the top-ranked response has a score of -1.21 and is incorrect, whereas the second-best response has a score of -1.27 and is correct.

We use this table to generate confusion matrices for precision/recall for all of the results up to that rank. The next four columns represent the counts in the cells of the confusion matrix, TP, TN, FP, and FN, for each rank. These are the basis for the computation of the next three columns, which are recall, precision, and specificity, at rank N.[11] For recall, this is the fraction of all reference positive items that have been retrieved at the specified rank or before. The highest recall achieved for non-zero precision is only 0.80 because we added a missed item, for a total of five reference positive items. Every time we hit a correct answer, the recall (sensitivity) and precision go up and specificity stays the same.

Precision is calculated slightly differently, being the fraction of results returned so far that are correct. Thus the first value is zero, because we have $\text{TP} = 0$ and $\text{FP} = 1$ at this point, for a precision of $\text{TP}/(\text{TP}+\text{FP}) = 0$. At rank 1, we

[11]LingPipe numbers ranks from zero for the purpose of precision-recall and ROC curves, but numbers them from one for the purposes of computing the conventional precision-at-rank and reciprocal rank statistics.

have TP = 1 and FP = 1, for a precision of 0.5. By the time we get down to rank 9, we have TP = 4 and FP = 6, for a precision of 0.4. We pin the precision and specificity of results corresponding to calls to `addMisses()` to 0, because there is no indication of the number of false positives that may show up ahead of the misses.

Specificity (or rejection recall) is calculated more like recall than precision. Specifically, we assume that at each rank, we have correctly rejected all the items we haven't mentioned. So at rank 0, we have TN = 6 and FP = 1.

Inspecting Precision-Recall Curves

After constructing and populating the evaluation, the code in the `main()` method goes on to print out the precision-recall curve implied by the data and represented in the instance `eval` of a scored precision-recall evaluation.

```
boolean interpolate = false;
double[][] prCurve = eval.prCurve(interpolate);
for (double[] pr : prCurve) {
    double recall = pr[0];
    double precision = pr[1];
```

We have set the interpolation flag to false, and then walked over the resulting curve pulling out recall/precision operating points (and then printing them in code not shown).

The Ant target `scored-precision-recall` runs the demo, which takes no command-line arguments. If we run it, the following shows the results for uninterpolated and interpolated precision-recall curves (the repeated code to print out the interpolated results is not shown here; it only differs in setting the interpolation flag to `true`).

```
> ant scored-precision-recall
Uninterpolated P/R      Interpolated P/R
prec=1.00 rec=0.00      prec=1.00 rec=0.00
prec=0.50 rec=0.20      prec=0.60 rec=0.20
prec=0.33 rec=0.20      prec=0.60 rec=0.40
prec=0.50 rec=0.40      prec=0.60 rec=0.60
prec=0.60 rec=0.60      prec=0.44 rec=0.80
prec=0.50 rec=0.60      prec=0.00 rec=1.00
prec=0.43 rec=0.60
prec=0.38 rec=0.60
prec=0.44 rec=0.80
prec=0.40 rec=0.80
prec=0.00 rec=1.00
```

The uninterpolated results are just the precision and recall values from our table. Interpolation has the effect of setting the precision at a given recall point r to be the maximum precision at any operating point of the same or higher recall. For instance, the precision at recall point 0.4 is interpolated to 0.6, becaus ethe precision at recall point 0.6 is 0.6.

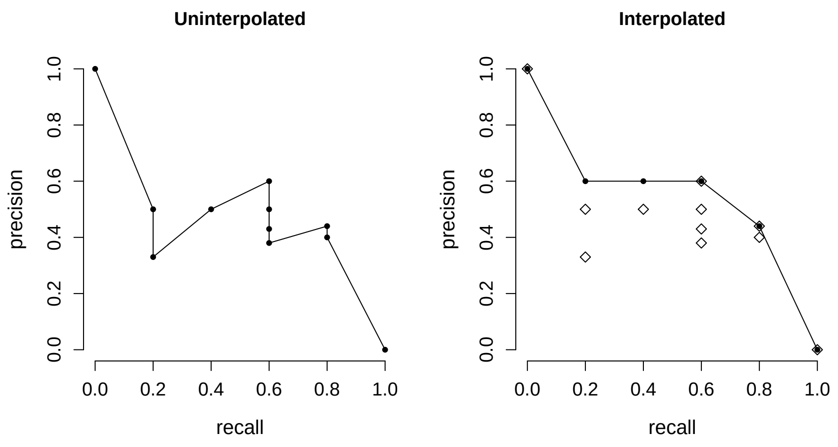

Fig. 9.7: *Uninterpolated and interpolated precision-recall curves for the data in Figure 9.6. In both graphs, the filled circles are the points on the curve. In the interpolated graphs, the diamonds correspond to points on the original, uninterpolated, curve.*

A graph of the uninterpolated and interpolated precision-recall curves are shown in Figure 9.7.

Inspecting ROC Curves

The code for retrieving receiver operating characteristic (ROC) curves is identical to that for precision/recall curves, only it uses the method `rocCurve(boolean)` rather than `prCurve(boolean)` and returns 1 − specificity/sensitivity pairs instead of recall/precision pairs.

```
interpolate = false;
double[][] rocCurve = eval.rocCurve(interpolate);
for (double[] ss : rocCurve) {
    double oneMinusSpecificity = ss[0];
    double sensitivity = ss[1];
```

The output from this block of code, and from the next block, which is identical other than setting the interpolation flag, is as follows.

```
Uninterpolated ROC         Interpolated ROC
1-spec=0.00 sens=0.00      1-spec=0.00 sens=0.00
1-spec=0.17 sens=0.00      1-spec=0.17 sens=0.20
1-spec=0.17 sens=0.20      1-spec=0.33 sens=0.60
1-spec=0.33 sens=0.20      1-spec=0.50 sens=0.60
1-spec=0.33 sens=0.40      1-spec=0.67 sens=0.60
1-spec=0.33 sens=0.60      1-spec=0.83 sens=0.80
1-spec=0.50 sens=0.60      1-spec=1.00 sens=1.00
1-spec=0.67 sens=0.60
1-spec=0.83 sens=0.60
1-spec=0.83 sens=0.80
```

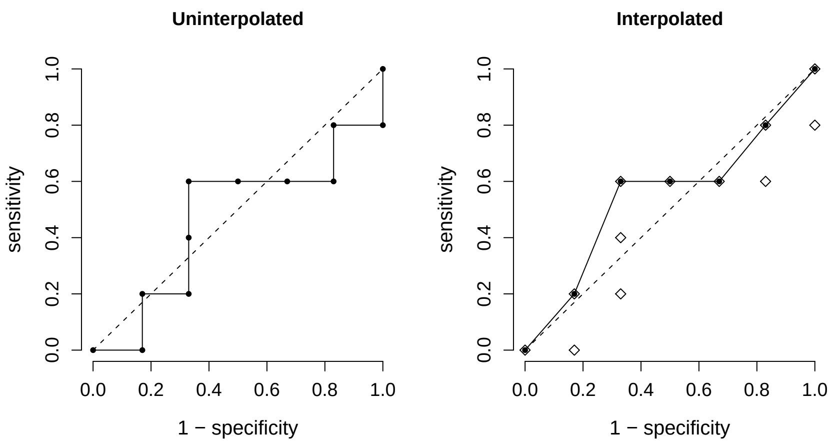

Fig. 9.8: *Uninterpolated and interpolated receiver-operating characteristic curves for the data in Figure 9.6. In both graphs, the filled circles are the points on the curve. In the interpolated graph, the diamonds correspond to points on the original, uninterpolated, curve. The dashed diagonal line is at chance performance.*

```
1-spec=1.00 sens=0.80
1-spec=1.00 sens=1.00
```

Interpolation works the same way as for precision-recall curves, removing dominated points with the same x-axis value. Graphs of the uninterpolated and interpolated ROC curves are shown in Figure 9.8.

The method `addNegativeMisses(int)` adds to the overall count of reference negative items. Because these will all be correctly rejected by not being returned, adding negative misses will primarily boost the specificity in ROC cuves.

We believe the precision-recall and ROC curves provide the best view of how a classifier works. It shows how tradeoffs may be made between precision and recall by setting different operating points for accepting a result. It is important to keep in mind, though, that these results are on evaluation data, and represent only an estimate of what results will be on unseen data. In practice, if results were achieved through tuning by cross validation, the precision-recall and ROC curves will be overly optimistic.

F Measure at Rank

LingPipe only returns the precision-recall operating points, not the *F* measure. To compute *F* measure, the static utility method `fMeasure(double,double,double)` may be used to compute the *F* measure of any two operating points. The three arguments are for the precision weight (which should be 1 as we have defined *F* measure here; see the next section for generalizations), and for precision and recall (though the order of precision and recall doesn't matter for computing *F* measure). So the call looks like `fMeasure(1.0,p,r)`, where the `1.0` just indicates that precision and recall are balanced, `p` is precision, and `r` is recall.

Precision at Rank *N*

For scored precision-recall evaluations, the method `precisionAt(int)` returns the precision at a specified rank. For example, with the data in Figure 9.6, the precision at rank N is determined by looking up the precision for that rank. Ranks are conventionally counted from 1, so that is how we count for the `precisionAt()` method.

The precision at every rank is calculated with the following code.

```
for (int n = 1; n <= eval.numCases(); ++n) {
     double precisionAtN = eval.precisionAt(n);
```

Note that our loop starts from 1 and uses a less-than-or-equal bound test to adjust for the fact that precision-at ranks start at 1. The output corresponds to the precisions in Figure 9.6.

```
Prec at  1=0.00     Prec at  5=0.60     Prec at  9=0.44
Prec at  2=0.50     Prec at  6=0.50     Prec at 10=0.40
Prec at  3=0.33     Prec at  7=0.43     Prec at 11=0.36
Prec at  4=0.50     Prec at  8=0.38
```

Maximum *F* Measure and BEP

Two other commonly reported operating points derived from the precision-recall curve are the maximum F measure and precision-recall breakeven point (BEP). Continuing our example, these are extracted as follows.

```
double maximumFMeasure = eval.maximumFMeasure();
double bep = eval.prBreakevenPoint();
```

Their values are output (code not shown) as

```
Maximum F =0.60       Prec/Rec BEP=0.44
```

These values are both computed from the uninterpolated curves and correspond to real points on the operating curve.

Break-even point is also known as R-precision, because it is equivalent to the precision at R, where R is the number of reference positive results. That's because if we have n true positives after R results, the the precision and recall are both n/R.

9.8 Contingency Tables and Derived Statistics

In this section, we provide a precise definition of contingency tables and discuss the computation and interpretation of statistics on contingency tables.[12]

A contingency table is an $M \times N$ matrix C of counts (i.e., $C_{m,n} \in \mathbb{N}$). A generic contingency table is shown in Figure 9.9. We use the notation $C_{m,+}$ for the sum

[12]This appendix is more advanced mathematically than the earlier presentation and assumes an understanding of probability distributions, densities, and hypothesis testing. See Appendix B for a refresher on the concepts and definitions needed to understand these more advanced sections.

	1	2	$\cdots$	N	*Total*
1	$C_{1,1}$	$C_{1,2}$	$\cdots$	$C_{1,N}$	$C_{1,+}$
2	$C_{2,1}$	$C_{2,2}$	$\cdots$	$C_{2,N}$	$C_{2,+}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
M	$C_{M,1}$	$C_{M,2}$	$\cdots$	$C_{M,N}$	$C_{M,+}$
Total	$C_{+,1}$	$C_{+,2}$	$\cdots$	$C_{+,N}$	$C_{+,+}$

Fig. 9.9: *A generic $M \times N$ contingency table C. Each entry $C_{m,n}$ represents a cell count. The values along the bottom and on the right side are totals of their respective columns and rows. The value in the lower right corner is the total count in the table.*

of the values in row m, $C_{+,n}$ for the sum of the values in column n, and and $C_{+,+}$ for the sum of the entire table. These values are defined by

$$C_{m,+} = \sum_{n=1}^{N} C_{m,n}, \qquad C_{+,n} = \sum_{m=1}^{M} C_{m,n}, \text{ and } \qquad C_{+,+} = \sum_{m=1}^{M} \sum_{n=1}^{N} C_{m,n}. \tag{9.16}$$

9.8.1 χ^2 Tests of Independence

The χ^2 distribution may be used to test the null hypothesis that that a pair of categorical random variables are generated independently.

Suppose we have K independent and identically distributed (i.i.d.) pairs A_k, B_k of categorical random variables taking values $A_k \in 1{:}M$ and $B_k \in 1{:}N$. Being identically distributed means the pairs share distributions, so that

$$p_{A_i,B_i} = p_{A_j,B_j} \text{ for } i, j \in 1{:}K. \tag{9.17}$$

Independence requires that the value of the pair A_i, B_i is independent of the value of the pair A_j, B_j if $i \neq j$.

So far, we have allowed for the possibility that A and B are not independent of each other. If A_k and B_k are independent, then

$$p_{A_k,B_k}(x, y) = p_{A_k}(x) \times p_{B_k}(y). \tag{9.18}$$

Because we've assumed the A_k, B_k pairs are i.i.d., they share distributions and hence are either all independent or all non-independent.

We define an $M \times N$ contingency table C by setting

$$C_{m,n} = \sum_{k=1}^{K} \mathrm{I}(A_k = m, B_k = n). \tag{9.19}$$

That is, $C_{m,n}$ is the number of times A_i was m and B_i was n. Note that C is a matrix random variable defined as a function of the random variables A and B.

Given a contingency table C, we can estimate the marginal distributions $p_A(m)$ and $p_B(n)$ by maximum likelihood using the observed counts in C (see Section B.3 for an overview of maximum likelihood estimation). Because p_A and p_B are multinomials, we assume $p_A = \mathsf{Bern}(\theta)$ and $p_B = \mathsf{Bern}(\phi)$ have Bernoulli distributions (i.e., coin flips) with chance-of-success parameters θ and ϕ (see

Section B.1.4 and Section B.1.1 for definitions of multinomial and Bernoulli distributions). The maximum likelihood estimates θ^*, ϕ^* of θ, ϕ given C are

$$\theta^*_m = \frac{C_{m,+}}{C_{+,+}} \quad \text{and} \quad \phi^*_n = \frac{C_{+,n}}{C_{+,+}}. \tag{9.20}$$

Pearson's independence test involves the statistic X^2, which is defined in terms of C by

$$X^2 = \sum_{m=1}^{M} \sum_{n=1}^{N} \frac{(C_{m,n} - E_{m,n})^2}{E_{m,n}} \tag{9.21}$$

where $E_{m,n}$ is the expected value of $C_{m,n}$ (given our estimates θ^* and ϕ^*) if A and B are independent,

$$E_{m,n} = C_{+,+} \times \theta^*_m \times \phi^*_n. \tag{9.22}$$

If A_k and B_k are independent, the distribution of the statistic X^2 is approximately χ^2 with $(M - 1) \times (N - 1)$ degrees of freedom (see Section B.4.1).[13] The usual rule of thumb is that the approximation is reliable if each of the expected counts $E_{m,n}$ is at least 5.

As usual, the classical hypothesis test rejects the null hypothesis with p-value α if the value of the test statistic X^2 is outside of the central and columns are independent if the X^2 statistic is outside of the central probabiilty interval of $1 - p$.

9.8.2 Further Contingency and Association Statistics

Pearson's Mean Square Contingency

Dividing Pearson's statistic X^2 (which we previously used for the χ^2 test) by the number of cases yields Pearson's φ^2 statistic.

$$\varphi^2 = \frac{X^2}{C_{+,+}} \tag{9.23}$$

The φ^2 statistic is known as the mean square contingency for the table C. As with X^2, larger values indicate more contingency.

Cramér's Degree of Association

Cramér's V statistic[14] is designed to measure the degree of association between the rows and columns in a general contingency table. The square of V is defined

[13]The proof is too complex, but we provide some hints as to its form. We have MN cells, so have $MN - 1$ degrees of freedom, of which we lose $M - 1$ and $N - 1$ because we are estimating θ^* and ϕ^*, which have one less than their dimensionality degrees of freedom, $M - 1$ and $N - 1$, for a total of $MN - 1 - (M + N - 2) = (M - 1)(N - 1)$ degrees of freedom (again, asymptotically). Because the $C_{m,n}$ are binomially distributed with success probability $\theta^*_m \times \phi^*_n$ and $C_{+,+}$ trials, the central limit theorem ensures asymptotic normality of $C_{m,n}$ as $C_{+,+}$ grows. The mean or expected value of the binomial is $E_{m,n}$, and a binomial's variance is equal to its mean. Rewriting the term inside the summation in the definition of the X^2 statistic as $((C_{m,n} - E_{m,n})/\sqrt{E_{m,n}})^2$ reveals that its just a squared z-score.

[14]H. Cramér. 1999. *Mathematical Methods of Statistics.* Princeton University Press.

by dividing φ^2 by the minimum of the number of rows and columns minus 1,

$$V^2 = \frac{\varphi^2}{\min(M, N) - 1}. \tag{9.24}$$

Obviously, the definition only makes sense if $M, N \geq 2$, and if $M = 2$ or $N = 2$, V^2 reduces to φ^2. As with φ^2 and X^2, larger values of V^2 indicate stronger associations.

Goodman And Kruskal's Index of Predictive Association

Goodman and Kruskal defined an asymmetric measure of predictive association which they called λ_B.[15] The value of λ_B is (an estimate of) the reduction in error likelihood from knowing the response category and using it to predict the reference category. It takes on values between 0 and 1, with higher values being better, 0 indicating independence and 1 perfect assocaition.

Given our $M \times N$ confusion matrix C, Goodman and Kruskal's index of predictive association λ_B is defined by

$$\lambda_A = \frac{(\sum_n R_n) - S}{C_{+,+} - S} \tag{9.25}$$

where S and R_n are defined by

$$S = \max_n C_{+,n} \quad \text{and} \quad R_n = \max_m C_{n,m}. \tag{9.26}$$

These statistics are unusual in using the maximum rather than matching. Dividing through by $C_{+,+}$ reveals a structure very much like the κ statistic (see Section 9.8.5), with e replaced by $S/C_{+,+}$ and with a replaced by $\sum_n R_n/C_{+,+}$.

The λ_B statistic is not symmetric between the rows and columns. If we transpose the matrix and compute the same statistic, we get $\lambda_A p$.

9.8.3 Information-Theoretic Measures

If we use the counts in C to estimate probability distributions using maximum likelihood, we may use any of the information-theoretic measures available to compare distributions (see Section B.5 for definitions).

For example, suppose we define random variables X and Y for the rows and columns, with joint distribution given by the maximum likelihood estimate given the contingency table C, which entails

$$p_{X,Y}(m, n) = \frac{C_{m,n}}{C_{+,+}}. \tag{9.27}$$

Given this defintion, the marginal distributions for X and Y are

$$p_X(m) = \frac{C_{m,+}}{C_{+,+}} \quad \text{and} \quad p_Y(n) = \frac{C_{+,n}}{C_{+,+}}. \tag{9.28}$$

[15]Goodman and Kruskal wrote three papers analyzing cross-classified data such as is found in contingency matrices, starting with Goodman, L. A. and W. H. Kruskal, 1954, Measures of association for cross classifications, Part I, *Journal of the American Statistical Association* **49**:732—764. With the same journal and title, Part II was published in 1959 in volume **53**, pages 123–163, and Part III in 1963 in volumne **58**, pages 310–364.

The conditional distributions for X given Y and vice-versa are

$$p_{X|Y}(m|n) = \frac{C_{m,n}}{C_{m,+}} \quad \text{and} \quad p_{Y|X}(m|n) = \frac{C_{m,n}}{C_{+,n}}. \tag{9.29}$$

With all of these definitions in hand, we can easily compute the various information theoretic measures, which are defined in Section B.5.

For instance, we can compute the entropy corresponding to the rows of the contingency table as $\mathrm{H}[X]$ and to the columns as $\mathrm{H}[Y]$. We can also compute the KL-divergences, symmetrized divergences and Jensen-Shannon divergence based on the marginal distributions p_X and p_Y implied by the contingency table.

With the joint and conditional estimates, we can tabulate the joint entropy $\mathrm{H}[X,Y]$, conditional entropies $\mathrm{H}[X|Y]$ and $\mathrm{H}[Y|X]$, as well as and mutual information $\mathrm{I}[X;Y]$.

9.8.4 Confusion Matrices

A confusion matrix is just a square ($N \times N$) contingency table where the variable represented by the rows and columns has the same set of categorical outcomes. Confusion matrices are often used to evaluate classifiers, taking the rows to represent reference categories and columns to represent system responses.

There is a wide range of statistics that have been applied to evaluating the performance of classifiers using confusion matrices. We repeat some of our earlier definitions in this chapter for completeness, providing fully specified definitions here for all of the statistics.

9.8.5 κ Statistics for Chance-Adjusted Agreement

Suppose we have an $N \times N$ confusion matrix C.

Cohen's κ statistic[16] is defined by

$$\kappa = \frac{a - e}{1 - e}, \tag{9.30}$$

where

$$a = \frac{\sum_{n=1}^{N} C_{n,n}}{C_{+,+}} \quad \text{and} \quad e = \sum_{n=1}^{N} \theta_n^* \times \phi_n^* = \frac{E_{n,n}}{C_{+,+}}. \tag{9.31}$$

In words, a is the percentage of cases in which there was agreement, in other words the total accuracy, and e is the expected accuracy if the reference and response are independent.

Siegel and Castellan's κ statistic[17] has the same definitional form as Cohen's, but the calculation of the expectation is based on averaging the reference and response to define

$$e = \sum_{n=1}^{N} \left(\frac{\theta_n^* + \phi_n^*}{2} \right)^2. \tag{9.32}$$

[16]Cohen, Jacob. 1960. A coefficient of agreement for nominal scales. *Educational And Psychological Measurement* **20**:37-46.

[17]Siegel, Sidney and N. John Castellan, Jr. 1988. *Nonparametric Statistics for the Behavioral Sciences.* McGraw Hill.

		Response	
		Positive	*Negative*
Reference	*Positive*	TP	FN
	Negative	FP	TN

Fig. 9.10: *A generic sensitivity-specificity matrix is a* 2×2 *confusion matrix with columns and rows labeled with "positive" and "negative.". The code P (positive) is assigned to reference positives and N (negative) to reference negatives, with T (true) assigned to correct responses and F (false) to incorrect responses.*

Byrt et al.'s κ statistic[18] completely ignores the adjustment for category prevalence found in e, taking an arbitrary 50% chance of agreement, corresponding to $e = 1/2$, and resulting in a definition of $\kappa = 2a - 1$.

9.8.6 Sensitivity-Specificity Matrices

A sensitivity-specificity matrix is a 2×2 confusion matrix where the binary categories are distinguished as "positive" and "negative" (or "true" and "false" or "accept" and "reject"). The cells have conventional names in a sensitivity-specificy matrix, which we show in Figure 9.10. The true positives (TP), false positives (FP) are for correct and incorrect responses when the reference category is positive, and true negative (TN) and false negative (FN) for correct and incorrect responses when the reference category is negative. False positives are also known as type-I errors or false alarms, and false negatives as misses or type-II errors.

Precision and Recall

For search and many classification evaluations, precision and recall are the most commonly reported metrics. Given a sensitivity-specificity matrix, precision and recall are calculated as

$$\text{precision} = \frac{TP}{TP + FP} \quad \text{and} \quad \text{recall} = \frac{TP}{TP + FN}. \tag{9.33}$$

Precision is the percentage of positive responses (TP + FN) that are correct, whereas and recall the percentage of positive reference items (TP + FP) that were assigned a positive response. Looking at the matrix, we see that recall is based on the top row and precision on the left column. In the next sections, we'll fill in the dual statistics.

Precision is sometimes called positive predictive accuracy, because it's the accuracy for items that get a positive prediction.

[18] Byrt, Ted, Janet Bishop and John B. Carlin. 1993. Bias, prevalence, and kappa. *Journal of Clinical Epidemiology* **46**(5):423–429.

Recall

	Pos	*Neg*
Pos	+	–
Neg		

Precision

	Pos	*Neg*
Pos	+	
Neg	–	

Rej. Recall

	Pos	*Neg*
Pos		
Neg	–	+

Rej. Precicision

	Pos	*Neg*
Pos		–
Neg		+

Fig. 9.11: *A graphical illustration of the transposition dualities involved in the definitions of recall (sensitivity) and precision (positive predictive accuracy) and in the definitions of rejection recall (specificity) and rejection precision (negative predictive accuracy). The symbol + indicates successfull attempts and – failed attempts. The statistic labeing the table is calculated as the percentage of attempts that were successful based on the value of the cells (see Figure 9.10).*

Sensitivity and Specificity

Sensitivity and specificity are more commonly used to evaluate sensitivity-specificity matrices (hence the name we gave them). These are defined by

$$\text{sensitivity} = \frac{TP}{TP + FN} \quad \text{and} \quad \text{specificity} = \frac{TN}{TN + FP} \tag{9.34}$$

Sensitivity is the same statistic as recall, being based on the top row of the matrix. One minus specificity is sometimes called fallout.

The easiest way to think of sensitivity and specificity is as accuracies on reference positives (TP + FN) and reference negatives (TN + FP), respectively.

Sensitivity and specificity are based on all four cells of the matrix, whereas precision and recall do not use the true negative count.

Transpose Duals and Rejection Precision

If we transpose the roles of the reference and response, we swap FN and FP counts, but TP and TN remain the same. Precision becomes recall and vice versa in the transposed matrix. Specificity, on the other hand, does not have a dual defined yet. To complete the picture, we introduce a statistic we call rejection precision, which is defined by

$$\text{rejection-precision} = \frac{TN}{TN + FP}. \tag{9.35}$$

This is the percentage of negative responses that are truly negative, and could also be called negative predictive accuracy.

To use terminology to match rejection precision, we could call specificity rejection recall. Rejection precision and rejection recall swap when the matrix is transposed in the same way as precision and recall. The dualities are clearest when viewed in table form, as laid out in Figure 9.11.

F Measure

A common measure used in information retrieval research is F measure, which is defined as the harmonic mean of precision and recall,

$$F = \left(\text{precision}^{-1} + \text{recall}^{-1}\right)^{-1} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}. \tag{9.36}$$

Sometimes, this measure is generalized with a weighting β to be

$$F_\beta = \frac{(1+\beta^2) \times \text{precision} \times \text{recall}}{\text{recall} + (\beta^2 \times \text{precision})}. \tag{9.37}$$

With this generalization, the original F measure in Equation 9.36 corresonds to the generalized F measure with $\beta = 1$.

F measure has the property, because it is based on a harmonic mean, of always being less than the simple average of precision and recall.

For some reason, F measure is the only commonly applied statistic based on the harmonic mean. You don't see people compute the harmonic mean of sensitivity and specificity or of rejection recall and rejection precision.

Jaccard Similarity Coefficient

The Jaccard similarity coefficient[19] provides a single measure of the quality of match between a response and a reference. The similarity coefficient J is defined by

$$J = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{FN}}. \tag{9.38}$$

It turns out to be very closely related to F measure, which may be rewritten by unfolding the defintion of precision and recall to

$$F = \frac{2 \times \text{precision} \times \text{recal}}{\text{precision} + \text{recall}} = \frac{(2 \times \text{TP})}{(2 \times \text{TP}) + \text{FP} + \text{FN}}. \tag{9.39}$$

This formulation reveals the relation of F measure to the Jaccard similarity coefficient; the difference is a factor of 2 boost to the true positives for F measure. The Jaccard similarity coefficient is less than F for non-degenerate matrices. Specifically, we will have $F > J$ if both $\text{TP} > 0$ and $\text{FP} + \text{FN} > 0$.

Fowlkes-Mallows Similarity Measure

Fowlkes and Mallows presented a measure of similarity for clusterings that may be adapted to sensitivity-specific matrices.[20,21] With a sensitivity-specificity matrix in hand, Fowlkes and Mallows used the geometric mean of precision and recall as their statistic,

$$B = \sqrt{\text{precision} \times \text{recall}} = \frac{\text{TP}}{\sqrt{(\text{TP} + \text{FN}) \times (\text{TP} + \text{FP})}}. \tag{9.40}$$

[19] Jaccard, Paul. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin de la Société Vaudoise des Sciences Naturelles* **37**:547–579.

[20] Fowlkes E. B. and C. L. Mallows. 1983. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association* **78**(384):553–584. `doi:10.2307/2288117`

[21] They first reduce hierarchical clusterings to flat clusterings (by setting a threshold on dendrogram height) then these flat clusterings to a sensitivity-specificity matrix. The set of cases is all pairs of objects being clustered. One clustering is set as the reference and one as the response. Reference positives are then defined as pairs of objects in the same cluster in the reference clustering and similarly for responses.

Yule's Q

Yule's Q statistic of association is based on the odds ratio

$$\rho = \frac{\text{sens}/(1-\text{sens})}{(1-\text{spec})/\text{spec}} = \frac{\left(\frac{\text{TP}}{\text{TP+FN}}\right) \Big/ \left(\frac{\text{FN}}{\text{TP+FN}}\right)}{\left(\frac{\text{FP}}{\text{FP+TN}}\right) \Big/ \left(\frac{\text{TN}}{\text{FP+TN}}\right)} = \frac{\text{TP}/\text{FN}}{\text{FP}/\text{TN}} = \frac{\text{TP} \times \text{TN}}{\text{FN} \times \text{FP}}. \tag{9.41}$$

The numerator is the odds of a positive item being correctly classified (sens) versus being incorrectly classified (1 − sens), and the denominator is the odds of a negative item being incorrectly classified (1 − spec) versus being correctly classified (spec). The numerator is the odds of the response being positive given that the reference is positive. The denominator is the odds of the response being positive given that the reference is negative. The ratio gives the increase in the odds of the response being negative given that the reference is negative.

Yule's Q statistic transforms the odds ratio ρ to the $[-1, 1]$ scale by

$$Q = \frac{\rho - 1}{\rho + 1} = \frac{(\text{TP} \times \text{TN}) - (\text{FN} \times \text{FP})}{(\text{TP} \times \text{TN}) + (\text{FN} \times \text{FP})}. \tag{9.42}$$

Yule's Y statistic dampens all these effects with square roots,

$$Y = \frac{\sqrt{(\text{TP} \times \text{TN})} - \sqrt{(\text{FN} \times \text{FP})}}{\sqrt{(\text{TP} \times \text{TN})} + \sqrt{(\text{FN} \times \text{FP})}}. \tag{9.43}$$

9.8.7 Collapsing Confusion Matrices for One-Versus-All

In evaluating classifiers and related systems, we often want to focus our analysis on a single category. We can do that by looking at that category in a confusion matrix. The alternative presented in this section instead reduces the confusion matrix to a sensitivity-specificity matrix for a single category. This is done by collapsing all the other categories by treating them as if they were the same.

Supose we have an $N \times N$ confusion matrix C, where $C_{n,m}$ is the count of the number of items with reference category n and response category m. We can pick a category $n \in 1{:}N$ and collapse the confusion matrix to a sensitivity-specificity matrix by making the following definitions. True positives and false positives are

$$\text{TP} = C_{n,n} \quad \text{and} \quad \text{FP} = C_{+,n} - C_{n,n}. \tag{9.44}$$

In words, the true positives correspond to reference instances of category n categorized as being category n. The false positives are items with response n and reference category other than n.

On the negative side, we have

$$\text{FN} = C_{n,+} - C_{n,n} \quad \text{and} \quad \text{TN} = C_{+,+} - C_{n,+} - C_{+,n} + C_{n,n}. \tag{9.45}$$

Here, the false negatives correspond to items which are of category n in the reference, but not in the response. The true negatives include all other cases, and works out to $\text{TN} = C_{+,+} - \text{TP} - \text{FP} - \text{FN}$.

Macro-Averaged Results

Statistics like overall accuracy computed on confusion matrices constitute what are known as micro-averaged results. The defining characteristic of micro-averaged statistics is that they are weighted by item, so that each item being evaluated contributes equally to the final statistics.

Macro averaging attempts to weight results so that each category has the same weight instead of every instance. The way this is usually carried out is to average one-versus-all results.

Suppose we have the following confusion matrix,

$$C = \begin{bmatrix} 85 & 5 & 10 \\ 3 & 20 & 2 \\ 1 & 1 & 3 \end{bmatrix}, \tag{9.46}$$

where as usual the rows are the reference and the columsn the response. This collapses into the three one-versus-all matrices.

$$D_1 = \begin{bmatrix} 85 & 15 \\ 4 & 26 \end{bmatrix}, \quad D_2 = \begin{bmatrix} 20 & 5 \\ 6 & 99 \end{bmatrix}, \text{ and } \quad D_3 = \begin{bmatrix} 3 & 2 \\ 12 & 113 \end{bmatrix}. \tag{9.47}$$

To compute the micro-averaged values, we just sum the three one-versus-all matrices position-wise to get

$$D_+ = \begin{bmatrix} 108 & 22 \\ 22 & 238 \end{bmatrix}. \tag{9.48}$$

The combined matrix always has the property that the false positive value is the same as the false negative count (here 22).

Micro-averaged statistics are calcluated directly from D_+. Macro-averaged statistics are calculated by first calculating the statistic on each D_n then averaging the results.

We can now compare micro- and macro-averaged statistics. For example, micro-averaged precision is 108/130 and micro-averaged recall is also 108/130, yielding a micro-averaged F measure of 108/130, or about 83%. Macro-averaged precision is the average of 85/100 (85%), 20/25 (80%) and 3/5 (60%), or about 75%. The low-count category 3 brings down the averages from the high count cases. This can work out the other way around, with macro-averaged values being higher than their micro-averaged counterparts.

9.8.8 Ranked Response Statistics

In natural language processing applications, especially ones related to search, system responses often take the form of ranked N-best lists.

For instance, a traditional web search engine might return the top N results, ranked in order of estimated relevance to the user's query.[22] Another example is a named-entity system that takes a biomedical journal article and returns a ranked list of identifiers for genes that were mentioned in the article. Yet another example would be automatically assigning keywords or tags to a blog entry or assigning MeSH terms to untagged MEDLINE citations.

[22]In practice for search, there is some benefit to balancing relevance and diversity in results.

Mathematical Formulation

In order to carry out ranked evaluations, we only need a ranked list of N responses along with an indication of whether they are correct or not. Mathematically, this is just a boolean vector $y \in \{0,1\}^N$. Very often the ordering is defined from an underlying score, which we might also have available as a parallel vector $x \in \mathbb{R}^N$.

Exhaustiveness for Evaluating Sensitivity

If we want to evaluate ranked sensitivity or recall, it's critical that we know how many positive answers there are in total. If we want to evaluate specificity, we also need to know how many negative answers there are in total.

For example, suppose there is a set T of 10 tags that might apply to a document. Now suppose we have a document and we know the relevant tags are $\{T_2, T_3, T_8\}$ of the total set of tags. Now suppose an automatic tagging system returns a ranked list of responses $\langle T_3, T_4, T_7, T_8\rangle$. Because the first and fourth ranked responses T_3 and T_8 are actually relevant, we take our boolean vector representation to be $y = \langle 1,0,0,1\rangle$.

By knowing that there is one additional reference tag not in the response, namely T_2, we are able to tell what percentage of all tags are found at each rank in the list of results.

One possibility would be to complete the response pessimistically, at least as far as evaluation goes. This means taking all the items which are not ranked and assuming the correct responses appear last. For instance, we'd extend our response vector $y = \langle 1,0,0,1\rangle$ representing four ranked responses, the first and fourth of which are correct, to eight ranked responses, $y = \langle 1,0,0,1,0,0,0,1\rangle$, with an additional correct response in the eighth (last) position. It might be better to average a bunch of random orderings of the non-responses, as that's what you'd expect to get if the system just returned randomly ordered results after its responses, but that approach would be computationally demanding. Instead, we will just truncate our evaluations at the extent of the user input and leave it for the reader to extrapolate to unseen responses.

Statistics at Rank n

Suppose we have an evaluated rank vector $y = \{0,1\}^J$ and we know that there are a total of T possible correct responses and F possible incorrect responses altogether.

The statistics at rank $n \in \mathbb{N}$, defined for $n < N$, are calcluated by converting the first n responses into a sensitivity-specificity matrix. The response positive cases have the obvious definitions

$$\text{TP}_n = \sum_{i=1}^{n} y_i \quad \text{and} \quad \text{FP}_n = \sum_{i=1}^{n} (1 - y_i). \tag{9.49}$$

To fill in the rest of the matrix, we know that everything we missed is a false negative and the rest are true negatives, giving us

$$\text{FN}_n = T - \text{TP}_n \quad \text{and} \quad \text{TN}_n = F - \text{FP}_n \tag{9.50}$$

As expected, we have $T + F = \mathrm{TP} + \mathrm{FP} + \mathrm{FN} + \mathrm{TN}$.

ROC and PR Curves

We now have a way to calculate a sensitivity-specificity matrix for each rank. The recieved operating characteristic (ROC) curve is defined to be the plot of sensitivity (on the vertical axis) versus one minus specificity (on the horizontal axis).

In addition to the ROC curves, it is common to see precision versus recall plotted in the same way, though there is less of a convention as to which value goes on which axis. We will call such plots PR curves.

Interpolated ROC and PR Curves

It is common in the information retrieval literature to see PR curves calculated by interpolation. Interpolation for both kinds of curves means adjusting the value for a given point on the x axis to be the maximum value for any point at that position or higher.[23]

By the same reasoning, we typically include operating points for 0% recall and 100% precision and for 100% recall and 0% precision. Interpolation may modify these operating points, so they may not appear as is in interpolated results.

Area under PR and ROC Curves

In order to reduce performance to a single statistic, it is common to see the area under the curve (AUC) reported.

The conventional approach to calculating these areas differs between PR and ROC curves. For PR curves, the usual approach is to use step-functions based on interpolation rather than connecting the operating points. In this case, area under the PR curve corresponds exactly to the average of the precision values at operating points corresponding to true positives. These will be evenly spaced in the interval between 0 and 1, and hence their average will correspond to the area under the curve.

For ROC curves, the usual approach is to report the area under the curve defined by connecting the points in the uninterpolated ROC curve.[24] The area-under methods in LingPipe compute actual area by using the resulting parallelograms. Areas may be computed under interpolated and uninterpolated curves.

The area under the ROC curve has a probabilistic interpretation as the estimate of the probability that a classifier will rank a random positive intstance

[23]As can be seen in the examples earlier, this does not quite correspond to computing the convex hull, because the results may still not be convex. The convex hull is an even more agressive interpolation that makes sense in some conditions where a system may probabilistically interpolate between results.

[24]The following article contains an excellent introduction to ROC curves and estimating the area under them:

> Lasko, Thomas A., Jui G. Bhagwat, Kelly H. Zou, and Lucila Ohno-Machado. 2005. The use of receiver operating characteristic curves in biomedical informatics. *Journal of Biomedical Informatics* **38**:404—415.

ahead of a randomly chosen negative instance. This is because the area under the curve will be the average specificity at true positive points, which is the average specificity at each true positive, and hence an estimate of the probability that a random negative item is ranked below a random positive item. In the continuous curve setting, this average becomes an integral, but the idea remains the same.

Averages on ROC and P-R Curves

The average value of precision is often reported for PR curves. This is conventionally done over all operating points, not just the ones on the convex hull, though it may be calculated either way.

When we have a whole set of evaluations, resulting in multiple PR curves, it's common to see mean average precision (MAP) reported, which simply averages the average precision precision values across the different PR curves.

Special Points on the P-R and ROC Curves

Several points on these curves are often called out for special attention. For PR curves, the precision-recall breakeven point (BEP) is the point on the curve for which precision equals recall. As we noted above, this value is equal to precision-at-R, where R is the number of reference positive results.

The second point that is often reported is the point on the PR curve with maximum F measure. In fact, we can compute any of the statistics applicable to sensitivity-specificity matrices and find the operating point that maximizes (or minimizes) it.

For information retrieval applications, it is common to see precision-at-N numbers reported, where we see what the precision is for the top N results. This statistic is relevant because we often present only N results to users.

Reciprocal Rank of First True Positive

The reciprocal rank statistic for ranked results is just the inverse $1/n$ of the rank n at which the first true positive appears int he list. Because we start counting from 1, $1/n$ will always fall between 0 and 1, with 1 being best.

As for average precision, it is common to report the average value of reciprocal rank across a number of evaluations.

9.9 Bias Correction

Suppose that when we evaluate a classifier on held-out data, it has a bias toward one category or the other. For instance, suppose we have a binary classifier with a sensitivity (accuracy on reference positive cases) of $\theta_1 = 0.8$ and a specificity (accuracy on negative reference cases) of $\theta_0 = 0.9$. If there are $I = 200$ positive reference cases and $J = 100$ negative cases, the classifier is expected to return an estimate of the number M of positive cases of

$$M = \theta_1 I + (1 - \theta_0)J = 0.8 \times 200 + (1 - 0.9) \times 100 = 170, \tag{9.51}$$

and an estimate of the number N of negative cases of

$$N = (1 - \theta_1)I + \theta_0 J = (1 - 0.8) \times 200 + 0.9 \times 100 = 130. \tag{9.52}$$

We will assume that we've evaluated our classifier so that we have an estimate of its sensitivity θ_1 and specificity θ_0.[25] In practice, we don't know the values of I or J, but we will know M and N. We are left with a pair of equations,

$$\theta_1 I + (1 - \theta_0)J = M, \text{ and} \tag{9.53}$$

$$(1 - \theta_1)I + \theta_0 J = N. \tag{9.54}$$

Solving for I and J in terms of M, N, θ_0, and θ_1 yields

$$J = \frac{\theta_1 N + \theta_1 M - M}{\theta_1 + \theta_2 - 1}, \text{ and} \tag{9.55}$$

$$I = M + N - J. \tag{9.56}$$

Thus This allows us to estimate the prevalence of positive responses, even though we have a biased classifier.

9.10 Post-Stratification

If the data are not in the same proportion in held out data as in test data, but we know the test data proportions, we could use post-stratification to predict overall accuracy on new test data.

This situation most commonly arises when we have artificially "balanced" training data. For instance, suppose we have 1000 positive and 1000 negative training examples for a binary classifier. Suppose we know our system has a sensitivity (accuracy on reference positive items) of 0.8 and a specificity (accuracy on reference negative items) of 0.9. On the test data, the overall accuracyw will be $(0.8 \times 1000 + 0.9 \times 1000)/2000$, which is 0.85.

But what if we know our test set is balanced completely differently, with 50 positive test items and 500 negative test items. If we don't know the proportion of each item in the test set, we could sample and annotate until we do. For instance, we know that about 15% of the 5000 or so articles released each day in MEDLINE are about genomics. But we might have training data with 1000 articles on genomics and 1000 items not on genomics.

Knowing the sensitivity θ_1, specificity θ_0, and prevalence π, which is the percentage of positive test items, we can predict our system's accuracy on the test data as $\pi\theta_1 + (1 - \pi)\theta_0$.

[25] In practice, there is uncertainty associated with these estimates which translates into downstream uncertainty in our adjusted estimates, but we'll take our Bayesian hats off for the time being and pretend our point estimates are correct.

Chapter 10

Naive Bayes Classifiers

So-called naive Bayes classifiers are neither naive nor, under their usual realization, Bayesian. In this chapter, we will focus on LingPipe's implementation of the traditional naive Bayes classifier, only returning the other naive Bayes-like classifier after we have covered language models and the general language-model classifiers.

After covering the basic modeling assumptions of naive Bayes, we provide a simple example to help you get started. The rest of the chapter considers the details of the API, how to tune and evaluate a naive Bayes model, and how to use it in a semi-supervised setting with small amounts of training

10.1 Introduction to Naive Bayes

The theory of naive Bayes classifiers is based on several fairly restrictive assumptions about the classification problem. This section lays out the basics.

10.1.1 Texts as Bags of Words

Although naive Bayes may be applied to arbitrary multivariate count data, LingPipe implements text classifiers, where the objects being classified are implementations of Java's `CharSequence` interface (e.g., `String` or `StringBuilder`).

For LingPipe's naive Bayes implementations, texts are represented as so-called bags of words.[1] LingPipe uses a tokenizer factory to convert a character sequence into a sequence of tokens. The order of these tokens doesn't matter for naive Bayes or other classifiers that operate over bags of words. As we'll see later, the way in which a sequence of characters is tokenized into sequences of strings plays a large role in tuning a naive Bayes classifier effectively.

A bag of words is like a set of words in that the order doesn't matter, but is unlike a set in that the count does matter. For instance, using a whitespace-based tokenizer, the strings *hee hee haw* and *hee haw hee* produce the same bag

[1]In general, these "words" will be arbitrary tokens, but "bag of words" is the usual terminology. In statistical parlance, a bag corresponds to a multinomial outcome.

of words, namely *hee* appears twice and *haw* once. These strings produce a different bag of words than *hee haw*, which only has a count of one for *hee*.

10.1.2 Exhaustivity and Exclusivity and Text Genre

Naive Bayes classifiers require two or more categories into which input texts are categorized. These categories must be both exhaustive and mutually exclusive.

For example, a news site such as Bing's or Google's, might classify a news story as to whether it belongs in the category U.S., World, Entertainment, Sci/Tech, Business, Politics, Sports, or Health. As another example, a research consortium might want to classify MEDLINE citations mentioning mice as to whether they mention the effects of any specific gene or not (the former class being useful for those researching the genetic landscape of mice). As a third example, a marketing firm interested in a particular brand might classify a blog post into four categories: positive toward brand, negative toward brand, neutral toward brand, doesn't mention brand. As a fourth example, we could classify a patients discharge summaries (long texts written by care givers) as to whether it indicates the patient is a smoker or not.

Note that in each case, the texts under consideration were a particular kind, such as newswire stories, MEDLINE citations about mice, general blog posts, or patient discharge summaries. We can refer to the set of documents of this type as a genre or the domain of the classifier.

The requirement of exhaustiveness is relative to texts that are drawn from the genre under consideration. We don't try to classify sports stories as to whether they are about genomics or not or have a positive sentiment toward the food quality.

Often, this genre boundary can be moved by reconceptualizing the classifier and training it on broader or narrower data types. For instance, the second example was restricted to MEDLINE citations about mice, and doesn't consider full-length research articles or scientific news stories, or even MEDLINE citations not about mice. The third example, in contrast, classifies all blog entries, but has a category "doesn't mention brand" to deal with posts not about the brand in question.

In practice, classifiers may be applied to texts drawn from a different genre from which they were trained. For instance, we could take blog sentiment classifiers and try to apply them to hotel reviews. Or we could apply MEDLINE citation classifiers to the full texts of research articles. In these cases, accuracy is almost always worse on out-of-domain texts than in-domain texts. For instance, we could apply our blog sentiment classifier to product reviews in magazines, but we would not expect it to work as well in that context as for the kinds of blogs over which it was trained. Similarly, we could apply the mouse genetics classifier to full-length journal articles, but it would likely not perform as well as for the citations over which it was trained.

10.1.3 Training Data: Natural versus Handmade

To train naive Bayes classifiers and other supervised classifiers, we require training data in the form of labeled instances. In the case of text classifiers, these consist of a sequence of texts paired with their unique categories.

For naive Bayes, and indeed for most statistical classifiers, the training data should be drawn at random from the same distribution as the test data will be drawn from. Naive Bayes uses information about which categories are most prevalent as well as what words are likely to show up in which category.

For instance, we would create training data for news story section headings by gathering news articles of the kind we'd like to classify and assigning them to categories by hand.

Sometimes we can find data already labeled for us. For instance, we could scrape a news aggregation or set of newspaper sites, recording the category under which the article was listed (and perhaps converting it back to the set of categories we care about). These might have originally arisen by hand labeling in the newspaper site, but are most likely automatically generated by a news aggregator. Similarly, we could examine the Medical Subject Heading (MeSH) tags applied to MEDLINE citations by its curators to see if they were marked as being about genomics and about mice.

Sometimes we can gather data from less directly labeled sources. For instance, we can find positive and negative restaurant reviews by examining how many stars a user assigned to them. Or find blog posts about food by creating a set of tags and searching for blogs with those tags. Or even by doing a web search with some terms chosen for each category.

Our training data is almost always noisy, even if labeled by task specialists by hand. Naive Bayes is particularly robust to noisy training data. This produces a quality-quantity tradeoff when creating the data. High-quality data labeling is very labor intensive. Sometimes it is more effective to collect more lower-quality data.

10.1.4 Generative Story

Naive Bayes classifiers are based on a probabilistic model of a corpus of texts with category-specific content. Models like naive Bayes are called "generative" in the machine learning literature because they are richly specified enough to generate whole corpora. In contrast, classifiers like logistic regression are not generative in the sense of being able to generate a corpus from the model.

The way in which naive Bayes represents a corpus, each document is provided with a single category among a set of possible categories. We suppose there is a fixed set of $K > 1$ categories and that each document belongs to exactly one category. To generate a document, we first generate its category based on a probability distribution telling us the prevalence of documents of each category in the collection.

Then, given the category of a document, we generate the words in the document according to a category-specific distribution over words. The words are generated independently from one another given the document's category. The conditional independence of words given the document category is almost always violated by natural language texts. that is why the naive Bayes model is often erroneously called “naive” (the error is in labeling the model itself naive rather than its application in a given setting).

Note that we do not model the selection of the number of topics K, the number of documents N, or the number of words M_n in the n-th document; these are given as constants.

10.1.5 Priors and Posteriors

Given a top-level distribution over categories along with a distribution over words for each category, it is straightforward to generate a corpus of documents. In a Bayesian setting we can go one step further and also generate the category prevalence distribution and the category-specific distributions over words. In order to do so, we need to go up a level of abstraction and consider a probability distribution over the parameters of a probabiltiy distribution. Then, we start by generating the prevalence distribution and the topic distributions, then continue to generate the documents.

Such a distribution over distributions is called a *prior* in Bayesian statistics. The reason such distributions are called priors has to do with how they apply in inference. The idea is that any point in time, we have a distribution over distributions. Then we observe data and update our prior distribution with the observed data to construct a posterior distribution. The posterior distribution combines the information in the prior with the information in the data. This process can be iterated, with the posterior aftering seeing a first batch of data used as the prior for subsequent data. This is, in fact, exactly what the naive Bayes implementation allows. The implementation starts with a prior distribution over parameters, then as it sees data, it updates the parameters of this prior distribution to define the posterior distribution.

In practice, we use a very simple kind of so-called *additive prior* that is both easy to understand and easy to calculate. The prior is characterized by prior counts for data. That is, we start from a condition that is the same as if we’ve observed some data. For instance, Laplace suggested adding one to all counts.

Our counts for naive Bayes are of two forms. The first is the number of documents that have been seen of each category. These counts are used to estimate the prevalence of categories. Our prior might do something like add 1 to all of the counts.

The second form of data is the number of times a token has been observed in a document of a given category. These counts are used to estimate the probability of tokens being generated in a document of a given category.

10.1.6 Maximum Likelihood Estimation

Suppose we choose to add zero prior counts to all of our data. The result is that the parameters will all be set to their empirical ratios. Adding zero prior counts is, in this case, equivalent to saying any distribution over categories or any distribution over the words in a category is equally likely. This leads to what is called maximum likelihood estimates, as we will see later.

10.2 Getting Started with Naive Bayes

LingPipe has two naive Bayes implementations. In this section, we focus on the traditional implementation of naive Bayes, which is found in the LingPipe class `TradNaiveBayes` in the package `com.aliasi.classify`.

10.2.1 Laugh Classification: His or Hers?

Our example involves classifying laughs based on whether they were produced by a man (his) or his wife (hers). The manly laugh consists of more "haw" and less "hee." The training data we will use has three examples of laughs from him and her. His laughs are *haw*, *haw hee haw*, and *haw haw*. Her laughs in the training data are *haw hee*, *hee hee hee haw*, and *haw*. Note that the single word *haw* shows up as a laugh for her and for him. Naive Bayes and all of our other classifiers can handle this kind of "inconsistent" training data, which is not actually inconsistent under a probabilistic model. It's a matter of who's most likely to utter what, not that they can't utter the same laughs.

10.2.2 Setting up the Classifier

The basic functionality of the naive Bayes classifier class can be gleaned from a simple demo program which shows how the model is trained and how it is run. We provide such an example in the class `TradNbDemo`, which consists of a single `main()` method. The method starts by assigning the input arguments, in this case a single argument representing the text to be classified.

```
String text = args[0];
```

The next step is to set up the classifier itself, the minimal constructor for which requires a tokenizer factory and set of categories represented as strings.

```
TokenizerFactory tf
    = new RegExTokenizerFactory("\\P{Z}+");

Set<String> cats
    = CollectionUtils.asSet("his","hers");

TradNaiveBayesClassifier classifier
    = new TradNaiveBayesClassifier(cats,tf);
```

The regular expression `\P{Z}+` (see the section on Unicode classes in the regular expression chapter of the companion volume, *Text Processing in Java*) produces a tokenizer factory that defines tokens to be maximal sequences of characters which Unicode considers not whitespace. We have used the `asSet()` method in LingPipe's `CollectionUtils` class to define a set of strings consisting of the categories his and hers. The classifier is constructed using the categories and tokenizer factory.

10.2.3 Providing Training Data

At this point, we are ready to train the classifier using training data. For the demo, we just hard code the training data. Each training datum consists of an instance of the class `Classified<CharSequence>`. These training data are passed to the classifier one at a time using its `handle()` method (see Section 2.1 for a general overview of LingPipe handlers). Because it implements `handle(Classified<CharSequence>)`, the naive Bayes classifier class is able to implement the interface `ObjectHandler<Classified<CharSequence>>`, which is convenient if we want to supply the classifier as a callback to either a parser or a corpus.

The order in which training data is provided to a naive Bayes classifier is irrelevant, so we will provide all the training data from the lady's laugh before the gentleman's.

```
Classification hersCl = new Classification("hers");

List<String> herTexts
     = Arrays.asList("haw hee", "hee hee hee haw", "haw");

for (String t : herTexts)
     classifier.handle(new Classified<CharSequence>(t,hersCl));
```

We start by creating a classification, `hersCl`, using the category name literal `"hers"`. A base LingPipe classification extends the `Classification` class in the `com.aliasi.classify` package. These are used as the results of classification. Classifications are immutable, so they may be reused for training, and thus we only have one.

After creating the classification, we create a list of training texts using Java's built-in utility `asList()` from the `Arrays` utility class. Note that we used a list rather than a set because we can train on the same item more than once. For instance, the woman in question may have hundreds of laughs in a training set with lots of duplication. Training on the same text again adds new information about the proportion of laughs of different kinds.

The final statement is a for-each loop, which iterates over the texts, wraps them in an instance of `Classified<CharSequence>`, and sends them to the classifier via its `handle(Classified<CharSequence>)` method. This is where the actual learning takes place. When the handle method is called, the classifier tokenizes the text and keeps track of the counts of each token it sees for each category, as well as the number of instances of each category.

We do the same thing for training the classifier for his laughs, so there's no need to show that code.

10.2.4 Performing Classification

Once we've trained our classifier with examples of his laughs and her laughs, we are ready to classify a new instance. This is done with a single call to the classifiers `classify(CharSequence)` method.

```
JointClassification jc = classifier.classify(text);
for (int rank = 0; rank < jc.size(); ++rank) {
     String cat = jc.category(rank);
     double condProb = jc.conditionalProbability(rank);
     double jointProb = jc.jointLog2Probability(rank);
```

Note that the result is an instance of `JointClassification`, which is the richest classification result defined in LingPipe. The joint classification provides a ranking of results, and for each provides the conditional probability of the category given the text as well as the log (base 2) of the joint probability of the category and the text. These are pulled off by iterating over the ranks and then pulling out the rank-specific values. After that, we print them in code that is not shown.

10.2.5 Running the Demo

The code is set up to be run from Ant using the target `nb-demo`, reading the value of property `text` for the text to be classified. for instance, if we want to classify the laugh *hee hee*, we would call it as follows.

```
> ant -Dtext="hee hee" nb-demo

Input=|hee hee|
Rank= 0  cat=hers  p(c|txt)=0.87  log2 p(c,txt)= -2.66
Rank= 1  cat= his  p(c|txt)=0.13  log2 p(c,txt)= -5.44
```

Our classifier estimates an 87% chance that *hee hee* was her laugh and not his, and thus it is the top-ranked answer (note that we count from 0, as usual). We will not worry about the joint probabilities for now.

10.2.6 Unknown Tokens

The naive Bayes model as set up in LingPipe's `TradNaiveBayesClassifier` class follows the standard practice of ignoring all tokens that were not seen in the training data. So if we set the text to *hee hee foo*, we get exactly the same output, because the token *foo* is simply ignored.

This may be viewed as a defect of the generative model, because it doesn't generate the entire text. We have the same problem if the tokenizer reduces, say by case normalizing or stemming or stoplisting — we lose the connection to the original text. Another way of thinking of naive Bayes is as classifying bags of tokens drawn from a known set.

10.3 Independence, Overdispersion and Probability Attenuation

Although naive Bayes returns probability estimates for categories given texts, these probability estimates are typically very poorly calibrated for all but the shortest texts. The underlying problem is the independence assumption underlying the naive Bayes model, which is not satisfied with natural language text.

A simple example should help. Suppose we have newswire articles about sports. For instance, as I write this, the lead story on the *New York Times* sports page is about the player Derek Jeter's contract negotiations with the New York Yankees baseball team. This article mentions the token *Jeter* almost 20 times, or about once every 100 words or so. If the independence assumptions underlying naive Bayes were correct, the odds against this happening would be astronomical. Yet this article is no different than many other articles about sports, or about any other topic for that matter, that focus on a small group of individuals.

In the naive Bayes model, the probability of seeing the word *Jeter* in a document is conditionally independent of the other words in the document. The words in a document are not strictly independent of each other, but they are independent of each other given the category of the document. In other words, the naive Bayes model assumes that in documents about sports, the word *Jeter* occurs at a constant rate. In reality, the term *Jeter* occurs much more frequently about baseball, particularly ones about the Yankees.

The failure of the independence assumption for naive Bayes manifests itself in the form of inaccurate probability assignments. Luckily, naive Bayes classifiers work better for first-best classification than one might expect given the violation of the assumptions on which the model is based. What happens is that the failed independence assumption mainly disturbs the probabiilty assignments to different categories given the texts, not the rankings of these categories.

Using our example from the previous section of the his and hers laughter classifier, we can easily demonstrate the effect. The easiest way to see this is duplicating the input. For instance, consider classifying *hee hee haw*, using

```
> ant -Dtext="hee hee haw" nb-demo

Rank= 0  cat=hers  p(c|txt)=0.79
Rank= 1  cat= his  p(c|txt)=0.21
```

If we simply double the input to *hee hee haw hee hee haw*, note how the probability estimates become more extreme.

```
> ant -Dtext="hee hee haw hee hee haw" nb-demo

Rank= 0  cat=hers  p(c|txt)=0.94
Rank= 1  cat= his  p(c|txt)=0.06
```

Just by duplicating the text, our estimate of the laugh being hers jumps from 0.79 to 0.94. The same thing happens when Derek Jeter's called out twenty times in a news story.

Because naive Bayes is using statistical inference, it is reasonable for the category probability estimates to become more certain when more data is observed. The problem is just that certainty grows exponentially with more data in naive Bayes, which is a bit too fast. As a result, naive Bayes typically grossly overestimates or underestimates probabilities of categories for documents. And the effect is almost always greater for longer documents.

10.4 Tokens, Counts and Sufficient Statistics

Document length per se is not itself a factor in naive Bayes models. It only comes into play indirectly by adding more tokens. In general, there are only two pieces of information that the naive Bayes classifier uses for training:

1. The bag of tokens for each category derived from combining all training examples, and
2. The number of training examples per category.

As long as we hold the number of examples per category constant, we can rearrange the positions of tokens in documents. For instance, we could replace his examples

```
List<String> hisTexts
     = Arrays.asList("haw", "haw hee haw", "haw haw");
```

with

```
     Arrays.asList("hee", "haw haw haw haw", "haw");
```

of even

```
     Arrays.asList("haw haw haw haw haw hee", "", "");
```

with absolutely no effect the resulting classifier's behavior. Neither the order of tokens or their arrangement into documents is considered.

The latter example shows that the empty string is a perfectly good training example; although it has no tokens (under most sensible tokenizers anyway), it does provide an example of him laughing and ups the overall probability of the laugher being him rather than her. That is, the last sequence of three training example is not equivalent to using a single example with the same tokens,

```
     Arrays.asList("haw haw haw haw haw hee");
```

10.5 Unbalanced Category Probabilities

In the simple example of laugh classification, we used the same number of training examples for his laughs and her laughs. The quantity under consideration is the number of times `handle()` was called, that is the total number of texts used to train each category, not the number of tokens.

The naive Bayes classifier uses the information gained from the number of training instances for each category. If we know that she is more likely to laugh than him, we can use that information to make better predictions.

In the example above, with balanced training sizes, if we provide no input, the classifier is left with only prevalence to go by, and returns a 50% chance for him or her, because that was the balance of laughs in the training set. The following invocation uses a text consisting of a single space (no tokens).[2]

[2]The program accepts the empty string, but Ant's notion of command-line arguments doesn't; empty string values for the `arg` element for the `java` task are just ignored, so that adding the line `<arg value=""/>` as an argument in the `java` element has no effect.

```
> ant -Dtext=" " nb-demo

Rank= 0  cat=hers  p(c|txt)=0.50
Rank= 1  cat= his  p(c|txt)=0.50
```

Now let's consider what happens when we modify our earlier demo to provide two training examples for her and three for him. The resulting code is in class CatSkew, which is identical to our earlier example except for the data,

```
List<String> herTexts
     = Arrays.asList("haw hee", "hee hee hee haw haw");

List<String> hisTexts
     = Arrays.asList("haw", "haw hee haw", "haw haw");
```

Note that we have used exactly the same tokens as the first time to train each category.

Now if we input a text with no tokens, we get a different estimate.

```
> ant -Dtext=" " cat-skew

Rank= 0  cat= his  p(c|txt)=0.58
Rank= 1  cat=hers  p(c|txt)=0.42
```

You might be wondering why the resulting estimate is only 58% likely to be his laugh when 60% of the training examples were his. The reason has to do with smoothing, to which we turn in the next section.

10.6 Maximum Likelihood Estimation and Smoothing

At root, a naive Bayes classifier estimates two kinds of things. First, it estimates the probability of each category independently of any tokens. As we saw in the last section, this is carried out based on the number of training examples presented for each category.

The second kind of estimation is carried out on a per-category basis. For each category, the naive Bayes model provides an estimate of the probability of seeing each token (in the training set) in that category.

10.6.1 Maximum Likelihood by Frequency

These estimates are carried out by counting. In the simplest case, for category prevalence, if there are two categories, A and B, with three training instances for category A and seven for category B, then a simple estimate, called the maximum likelihood estimate, can be derived based on relative frequency in the training data. In this case, the estimated probability of category A is 0.3 and that of category B is 0.7.

The frequency-based estimate is called "maximum likelihood" because it assigns the probability that provides the highest probability estimate for the data that's seen. If we have three instance of category A and seven of category B, the maximum likelihood estimate is 0.3 for A and 0.7 for category B. The same thing holds, namely that maximum likelihood probability estimates are proportional to frequency, in the situation where there are more than two categories.

The problem with simple frequency-based estimates is that they are not very robust for large numbers of words with limited training data, which is just what we find in language. For instance, if we had training for him and her that looked as follows,

```
List<String> hisTexts
    = Arrays.asList("haw", "har har", "hee haw haw");
List<String> herTexts
    = Arrays.asList("tee hee", "hee hee", "hee hee haw");
```

then the probability assigned to her uttering *har* or him uttering *tee* would be zero. This leads to a particularly troubling situation when classifying an utterance such as *har tee har*, which contains both strings. Such a string would be impossible in the model, because he is assigned zero probability of uttering *tee* and she's assigned zero probability of uttering *har*. This becomes a huge problem when we're dealing with vocabularies of thousands or millions of possible tokens, many of which are seen only a handful of times in a few categories.

10.6.2 Prior Counts for Smoothed Estimates

To get around the zero-probability estimates arising from maximum likelihood, the usual approach is to smooth these estimates. The simplest way to do this is to start all of the counts at a (typically small) positive initial value. This leads to what is known as *additive smoothing* estimates and the amount added is called the *prior count.*[3]

There is a second constructor for `TradNaiveBayesClassifier` that allows the prior counts to be specified, as well as a length normalization, which we currently set to `Double.NaN` in order to turn off that feature (see Section 10.8 for more information on length normalization).

An example of the use of this constructor is provided in the class `AdditiveSmooth` in this chapter's package. It consists of a static main method, which begins as follows.

```
Set<String> cats = CollectionUtils.asSet("hot","cold");
TokenizerFactory tf = new RegExTokenizerFactory("\\S+");
double catPrior = 1.0;
double tokenPrior = 0.5;
double lengthNorm = Double.NaN;
TradNaiveBayesClassifier classifier
    = new TradNaiveBayesClassifier(cats,tf,catPrior,
                                   tokenPrior,lengthNorm);
```

[3]In statistical terms, additive smoothing produces the maximum a posteriori (MAP) parameter estimate given a symmetric Dirichlet prior with parameter value α equal to the prior count plus 1.

Here we have defined the set of categories to be *hot* and *cold*, used a whitespace-breaking tokenizer factory, and set the category prior count to 1.0 and the token prior count to 0.5.[4] This will have the effect of adding 0.5 to the count of all tokens and 1.0 to the count of all categories.

Next, consider training three hot examples and two cold examples.

```
Classification hot = new Classification("hot");
for (String s : Arrays.asList("super steamy out",
                              "boiling",
                              "steamy today"))
     classifier.handle(new Classified<CharSequence>(s,hot));
Classification cold = new Classification("cold");
for (String s : Arrays.asList("freezing out",
                              "icy"))
     classifier.handle(new Classified<CharSequence>(s,cold));
```

There is a total of 7 different tokens in these five training items, *super*, *steamy*, *out*, *boiling*, *today*, *freezing*, and *icy*. Only the token *out* appears in both a hot training example and a cold one. All other tokens appear once, except *steamy*, which appears twice in the hot category.

After building the model, the code iterates over the categories and tokens printing probabilities. First, the category probabilities are computed; for generality, the code uses the method `categorySet()` on the classifier to retrieve the categories.

```
for (String cat : classifier.categorySet()) {
     double probCat = classifier.probCat(cat);
```

The method `probCat()` retrieves the probability of a category.

The next code block just iterates over the tokens using the traditional naive Bayes classifier method `knownTokenSet()`, which contains all the tokens in the training data. This is an unmodifiable view of the actual token set. Testing whether a token is known may be carried out directly using the method `isKnownToken(String)`.

Within the loop, it again iterates over the categories. This time, the traditional naive Bayes classifier method `probToken()`, with arguments for a token and category, calculates the estimated probability that any given token in a document of the specified category is the specified token.

```
for (String tok : classifier.knownTokenSet()) {
     for (String cat : classifier.categorySet()) {
         double pTok = classifier.probToken(tok,cat);
```

Running the Demo

There is an ant target `additive-smooth` which calls the command.

[4]It may appear odd that we are allowing “counts” to take on fractional values. This is unproblematic because all of our estimates are derived from ratios. On the other hands, counts will never be allowed to be negative.

```
> ant additive-smooth

p(cold)=0.429    p(hot)=0.571

p(   super|cold)=0.077    p(   super| hot)=0.158
p(     icy|cold)=0.231    p(     icy| hot)=0.053
p( boiling|cold)=0.077    p( boiling| hot)=0.158
p(  steamy|cold)=0.077    p(  steamy| hot)=0.263
p(   today|cold)=0.077    p(   today| hot)=0.158
p(freezing|cold)=0.231    p(freezing| hot)=0.053
p(     out|cold)=0.231    p(     out| hot)=0.158
```

At the top, we see the probabilities of the categories *cold* and *hot* before seeing any data. Note that these two values sums to 1. Below that, columns display the estimated probability of each token given the category, hot or cold. For each category, *hot* and *cold*, the sum of the probabilities of all tokens is also 1.

Each of these lines corresponds to a parameter in the model. Thus this model has 16 parameters. We only need to specify 13 of them, though. Given $p(cold)$, we know that $p(hot) = 1 - p(cold)$. Furthermore, we know that $p(out|cold)$ is one minus the sum of the probabilities of the other tokens, and similarly for $p(out|hot)$.

Let's look at some particular estimates, focusing on the hot category. Given this training data, a token in a statement expressing hot weather is 5% or so likely to be *icy* and 26% likely to be *steamy*. The token *icy* was not in the training data for the hot category, whereas the token *steamy* appeared twice.

With additively smoothed estimates, the probability assigned to a given token is proportional to that token's count in the training data plus its prior count. Looking at our examples, these are 2.5 for *steamy* (count of 2 plus 0.5 prior count), 0.5 for *icy* and *freezing*, and 1.5 for the other four tokens. Thus the total effective count is $1 \times 2.5 + 4 \times 1.5 + 2 \times 0.5 = 9.5$, and the probability for *steamy* is $2.5/9.5 \approx 0.263$. Given this estimation method, the sum of the token probabilities is guaranteed to be 1.

The probabilities for categories are calculated in the same way. That is, the overall count of hot instances was 3, whereas the count of cold training instances was 2. The prior count for categories was 1.0, so the effective count for hot is 4 and for cold is 3, so the estimated probability of a hot category is $4/7 \approx 0.571$. As with the tokens, the category probabilities will sum to 1, which is as it should be with the interpretation of the categories as exclusive and exhaustive.

10.7 Item-Weighted Training

Having seen in the last section how prior counts and training examples determine estimates for two kinds of probabilities. First, the probability of a category, and second, the probability of a token in a message of a given category. In each case, we added one to the count of categories or tokens for each relevant item in the training set.

There are two situations where it is helpful to weight the training examples non-uniformly. The first situation arises when some examples are more important than others. These can be given higher weights. The second situation arises when there is uncertainty in the category assignments. For instance, we might have a string and be only 90% sure it's about hot weather. In this case, we can train the hot catgory with a count of 0.9 and the cold category with a count of 0.1 for the same training example. This latter form of training is particularly useful for semi-supervised learning, which we consider below. For now, we will just describe the mechanisms for carrying it out and see what actually happens.

We provide an example in the class `ItemWeighting` in this chapter's package. For the most part, it is the same as the `AdditiveSmooth` class in the way it sets up the classifier and prints out the probability estimates. The difference is that it weights training examples rather than using the default weight of 1.0.

```
Classification hot = new Classification("hot");
Classification cold = new Classification("cold");
classifier.train("dress warmly",cold,0.8);
classifier.train("mild out",cold,0.5);
classifier.train("boiling out",hot,0.99);
```

We've used a smaller example set to focus on the effect of weighting.

There is an ant target `item-weight` that prints out the resulting estimates, which are as follows.

```
> ant item-weight

p(cold)=0.536    p(hot)=0.464

p( boiling|cold)=0.098    p( boiling| hot)=0.333
p(  warmly|cold)=0.255    p(  warmly| hot)=0.112
p(   dress|cold)=0.255    p(   dress| hot)=0.112
p(    mild|cold)=0.196    p(    mild| hot)=0.112
p(     out|cold)=0.196    p(     out| hot)=0.333
```

There are two cold training instances trained with weights of 0.8 and 0.5. Now rather than adding 1 to the category count for each instance, we add its weight, getting a total observed weight of 1.3. With a prior category count of 1.0, the efective count for the cold category is 2.3. There is one training instance for the hot category, trained with a weight of 0.99, so the effective hot category count is 1.99, and the resulting probability of the cold category is $2.3/(2.3+1.99) \approx 0.536$.

The tokens get weighted in the same way. For instance, dress is weighted 0.8, and there is a prior count of 0.5 for tokens, so the effective count of *dress* in the cold category is 1.3; its effective count in the hot category is just the prior count, 0.5. The token *out* is weighted 1.0 in the cold category and 1.49 in the hot category. The total count for cold tokens corresponds to all counts and priors, with effective counts of 1.3 for *dress*, 1.3 for *warmly*, 1.0 for *mild* and 1.0 for *cold*. There is also a count of 0.5 for *boiling*, even though it was never used in a cold example. Thus the probability estimate for *warmly* is $1.3/(1.3 + 1.3 + 1.0 + 1.0 + 0.5) \approx 0.255$.

10.7.1 Training with Conditional Classifications

Because it can train with weighted examples, the traditional naive Bayes classifier supports a method to train using a conditional classification. The method `trainConditional()` takes four arguments, a `CharSequence` for the input text, a `ConditionalClassification` for the result, as well as a count multiplier and minimum category virtual count, the effect of which we describe below. This result is the same as if weighted training had been carried out for every category with weight equal to the conditional probability of the category in the classification times the count multiplier. If the virtual count consisting of the probability times the count multiplier is less than the minimum virtual count, the category is not trained.

10.8 Document Length Normalization

It is common in applications of naive Bayes classification to mitigate the attenuating effect of document length and correlation among the tokens by treating each document as if it were the same length. This can result in better probabilistic predictions by reducing the tendency of conditional probability estimates for categories to be zero or one. Length normalization does not affect the ranking of categories for a single document, and thus returns the same first-best guess as to category for a document. Even though it doesn't change rankings, length normalization does rescale the probabilty estimates of categories given text. This can lead to better calibration of probabilities across documents, by making the probability estimates less dependent on document length and token correlation.

Document length normalization works by multiplying the weight of each token in the document by the actual document length divided by the length norm. Thus if the document is ten tokens long and the length normalization is four tokens, each token of the ten in the input counts as if it were only 0.4 tokens, for a total length of four tokens. If the length norm is four tokens and the input is two tokens long, each token counts as if it were two tokens. If the numbers are round, the effects will be the same as removing that many tokens. For instance, if the input is *hee hee haw haw haw haw* and the length normalization is 3, the result is the same as presenting *hee haw haw* to an un-length-normalized naive Bayes instance.

As we will see below in an example, length normalization applies before unknown tokens are discarded. Thus the behavior before and after length normalization is different if there are unknown tokens present. The decision to implement things this way is rather arbitrary; the model would be coherent under the decision to throw away unknown tokens before rather than after computing length. As implemented in LingPipe, unknown tokens increase uncertainty, but do not change the relative ranking of documents.

Because of the multiplicative nature of probability, overall scores are geometric averages, so that the reweighting takes place in the exponent, not as a simple multiplicative factor.

10.8.1 Demo: Length Normalization

Setting Length Normalization

The length normalization value, which is the effective length used for all documents, may be set in the constructor or using the method `setLengthNorm(double)`. The current value of the length norm is oreturned by `lengthNorm()`. Length norms cannot be negative, zero, or infinite. Setting the length norm to `Double.NaN` turns off length normalization.

The class `LengthNorm` in this chapter's package trains exactly the same model using exactly the same data as in our first example in Section 10.2. To repeat, his laughs are trained on three examples, *haw*, *haw hee haw*, and *haw haw*, and hers are trained on three instances, *haw hee*, *hee hee hee haw*, and *haw*.

The length normalization is read in as the second command-line argument after the text to classify, and parsed into the `double` variable `lengthNorm`. The length norm for the classifier is then set with the `setLengthNorm()` method.

```
classifier.setLengthNorm(lengthNorm);
JointClassification jc = classifier.classify(text);
```

Note that we set the value before running classification. It doesn't matter when the length norm is set before classification. We could have also set it in the constructor.

Running Examples

The Ant target `length-norm` runs the demo, with the text being supplied through property `text` and the length norm through property `length.norm`. For example, we can evaluate a length norm of 3.0 tokens as follows (with the rank 1 results elided):

```
> ant -Dtext="hee hee haw" -Dlength.norm=3.0 length-norm

Input=|hee hee haw|   Length Norm=   3.00
Rank= 0  cat=hers  p(c|txt)=0.79  log2 p(c,txt)= -3.85
```

Unlike the situation without length normalization, the number of times the input is repeated no longer matters. If we repeat it twice, we get the same result.

```
>           ant -Dtext="hee hee haw hee hee haw" -Dlength.norm=3.0
length-norm

Input=|hee hee haw hee hee haw|   Length Norm=   3.00
Rank= 0  cat=hers  p(c|txt)=0.79  log2 p(c,txt)= -3.85
```

We can also see that with length normalization, unknown tokens are included in the length normalization. For example, consider the following, which adds two unknown tokens to the input.

```
> ant -Dtext="hee hee haw foo bar" -Dlength.norm=3.0 length-norm

Input=|hee hee haw foo bar|   Length Norm=   3.00
Rank= 0  cat=hers  p(c|txt)=0.69  log2 p(c,txt)= -2.71
```

With length normalization, the presence of unknown tokens drive the probability estimates closer to 0.5 because they participate in the length calculation but are not discriminative.

Providing a length-norm value of `Double.NaN` turns off length normalization. For the input of *hee hee haw*, this provides the same result as setting length normalization to 3.0.

```
> ant -Dtext="hee hee haw" -Dlength.norm=NaN length-norm
```

```
Input=|hee hee haw|   Length Norm=    NaN
Rank= 0  cat=hers  p(c|txt)=0.79  log2 p(c,txt)= -3.85
```

But now if we duplicate the document, we see the attenuating effect of length again.

```
>          ant -Dtext="hee hee haw hee hee haw" -Dlength.norm=NaN
length-norm
```

```
Input=|hee hee haw hee hee haw|   Length Norm=    NaN
Rank= 0  cat=hers  p(c|txt)=0.94  log2 p(c,txt)= -6.71
```

10.9 Serialization and Compilation

The traditional naive Bayes classifier class implements both Java's `Serializable` interface and LingPipe's `Compilable` interface. See the sections on Java serialization and LingPipe compilation in the chapter on I/O in the companion volume, *Text Processing in Java*.

Serializing a traditional naive Bayes classifier and reading it back in results in an instance of `TradNaiveBayesClassifier` in exactly the same state as the serialized classifier. In particular, a deserialized traditional naive Bayes classifier may be trained with further examples.

For more efficiency, a traditional naive Bayes classifier may be compiled. Among other optimizations, all the logarithms required to prevent underflows and convert multiplications to additions are precomputed.

Compiling a traditional naive Bayes classifier and reading it back in results in an implementation of `JointClassifier<CharSequence>` if there are more than two categories. If there are only two categories, the class read back in will only implement the interface `ConditionalClassifier<CharSequence>`. The reason for this is that the binary (two-category) case is heavily optimized for computing the conditional probabilities and adding in the data to compute the joint probabilities would double the size and halve the speed of binary compiled classifiers.

In order to serialize a traditional naive Bayes classifier, its tokenizer factory must be serializable. For compilation, the tokenizer factory must be either compilable or serializable. If the tokenizer factory implements LingPipe's `Compilable` interface it'll be compiled, otherwise it'll be serialized.[5]

[5]This is carried out with LingPipe's static utility method `compileOrSerialize()`, which may be found in the `AbstractExternalizable` class in the `com.aliasi.util` package.

10.9.1 Serialization and Compilation Demo

The demo class SerializeCompile in this chapter's package illustrates both serialization and compilation. Because deserialization in java may throw class-not-found and I/O exceptions, the main method starts off declaring that.

```
public static void main(String[] args)
     throws IOException, ClassNotFoundException {
```

The first two command-line arguments are for the text to be classified and a (temporary) file name into which the classifier is serialized (or compiled).

After constructing and training with the same code as in the first example in Section 10.2, compiling a naive Bayes classifier to a file and reading it back in may be accomplished as follows.

```
AbstractExternalizable.compileTo(classifier,file);

@SuppressWarnings("unchecked")
ConditionalClassifier<CharSequence> compiledClassifier
     = (ConditionalClassifier<CharSequence>)
     AbstractExternalizable.readObject(file);

file.delete();
```

The static utility method compileTo() from Java's class AbstractExternalizable class (in the util package) is used to do the writing. This could also be done through LingPipe's Compilable interface directly using the traditional naive Bayes class's method compileTo(ObjectOut). We deserialized using another utility method, readObject(), which reads and returns serialized objects from files (there are also utility methods to read compiled or serialized models from the class path as resources).

Usually, one program would write the compiled file and another program, perhaps on another machine or at a different site, would read it. Here, we have put the two operations together for reference. Note that the unchecked cast warning on reading back in is suppressed; an error may still result at run-time from the cast if the file supplied to read back in has an object of a different type or isn't a serialized object at all. Note that when read back in, it is assigned to a JointClassifier<CharSequence>; attempting to cast to a TradNaiveBayesClassifier would fail, as the compiled version is not an instance of that class.

We do the same thing for serialization, using a different utility method to serialize, but the same readObject() method to deserialize.

```
AbstractExternalizable.serializeTo(classifier,file);

@SuppressWarnings("unchecked")
TradNaiveBayesClassifier deserializedClassifier
     = (TradNaiveBayesClassifier)
     AbstractExternalizable.readObject(file);
file.delete();
```

Here, we are able to cast the deserialized object back to the original class, `TradNaiveBayesClassifier`.

Because deserialization results in a traditional naive Bayes classifier, we may provide more training data. Repeating the serialization and deserialization with a different variable, we can go on to train.

```
String s = "hardy har har";
Classified<CharSequence> trainInstance
     = new Classified<CharSequence>(s,hisCl);
deserializedClassifierTrain.handle(trainInstance);
```

Running the Demo

The Ant target `serialize-compile` runs the class, classifying a text supplied as property `text` and using a file derived from the value of the property `file`. Running it shows the result of classifying with the original, compiled, deserialized, and then deserialized and further trained classifier.

```
> ant -Dtext="hee haw hee hee" serialize-compile
text=hee haw hee hee

Results for: Original
Rank= 0  cat=hers  p(c|txt)=0.91

Results for: Compiled
Rank= 0  cat=hers  p(c|txt)=0.91

Results for: Serialized
Rank= 0  cat=hers  p(c|txt)=0.91

Results for: Serialized, Additional Training
Rank= 0  cat=hers  p(c|txt)=0.97
```

Note that the conditional probability assigned to her laughing are the same for the original, compiled, and serialized model. The estimate is different for the model with additional training data, as we would expect. We have not reported log joint probabilities because they are not produced by the compiled model.

10.10 Training and Testing with a Corpus

In this section, we show how to use a `Corpus` implementation to train and test a classifier. In our demo of LingPipe's `Corpus` abstract base class in Section 2.3.2, we showed how to create a corpus from the 20 Newsgroups data set (the data and download location are described in Section D.2).

We will re-use our corpus code in this demo, though we have copied the implementation so as to not make the imports in the demos too confusing. Thus you'll find the implementation in `TwentyNewsgroupsCorpus` in this chapter's demo package (`com.lingpipe.book.naivebayes`).

10.10.1 Training Naive Bayes with a Corpus

Training a classifier like naive Bayes with a corpus is particularly straightforward. As usual, the demo code lives in the `main()` method and starts by marshalling the command-line arguments, here just the name of the tar gzipped file in which the corpus was saved as the `File` variable `corpusTgzFile`. We then just use the constructor to create the corpus.

```
Corpus<ObjectHandler<Classified<CharSequence>>> corpus
     = new TwentyNewsgroupsCorpus(corpusTgzFile);

Set<String> catSet = getCatSet(corpus);
String[] cats = catSet.toArray(Strings.EMPTY_STRING_ARRAY);
Arrays.sort(cats);
```

Given the corpus, we apply the static method `getCatSet()` to extract the categories from the corpus. The set is then used to generate an array of categories, which is then sorted.

The static utility category extraction method is defined using an anonymous class used as a callback, as follows.

```
static <T> Set<String>
     getCatSet(Corpus<ObjectHandler<Classified<T>>> corpus)
     throws IOException {

     final Set<String> cSet = new HashSet<String>();
     corpus.visitCorpus(new ObjectHandler<Classified<T>>() {
             public void handle(Classified<T> c) {
                 cSet.add(c.getClassification().bestCategory());
             }
         });
     return cSet;
}
```

The set is declared to be final so that it can be used within the anonymous handler's method. The corpus method `visitCorpus()` is then applied to a newly defined object handler instance; in general, the `visitCorpus()` method sends all of the classified objects in the corpus to the specified handler. In this case, our handler is anonymous and handles a classified object by adding its classification's first best category assignment to the category set. The generic <T> is being used for the type of object being classified, which doesn't matter here, but must be kept consistent across the corpus and the inner class handling the callbacks.

The next block of code creates a fully parameterized traditional naive Bayes classifier.

```
TokenizerFactory tokFact
     = IndoEuropeanTokenizerFactory.INSTANCE;

double catPrior = 1.0;
double tokPrior = 0.1;
double lengthNorm = 10.0;
```

```
TradNaiveBayesClassifier classifier
     = new TradNaiveBayesClassifier(catSet,tokFact,catPrior,
                                    tokPrior,lengthNorm);
corpus.visitTrain(classifier);
```

LingPipe's built-in Indo-Euroepan tokenizer factory singleton is used for tokenization (see Chapter 3 for more on tokenization in general and Section 3.2.1 for more information on LingPipe's Indo-European tokenizer factory). Further note that the category prior count is set to 1.0 and the token count prior to 0.1, and the length normalization is set to 10.0; these were explained earlier in this chapter.

After setting up the classifier, the corpus's `visitTrain()` method is used to send all of the training instances to the classifier instance.

After visiting the training data, the classifier's ready to use. Instead of using it directly, we compile it to a more efficient form. This is done with the utility method `compile()` in LingPipe's `AbstractExternalizable` class (see the section on compilation in the I/O chapter of the companion volume, *Text Processing in Java*). The required cast may throw I/O exceptions and the implicit deserialization may throw a class-not-found exception, so those are declared on the top-level `main()` method. Note that we have assigned the compiled object to a more general type, `ConditionalClassifier`. That's because the compiled form of the traditional naive Bayes classifiers are not instances of `TradNaiveBayesClassifier`.

10.10.2 Evaluation with a Corpus

Because our corpus was built with a training and test section according to the way the 20 Newsgroup corpus is distributed, we can also use the corpus for testing. The first step is to set up the evaluator.

```
boolean storeInputs = false;
ConditionalClassifierEvaluator<CharSequence> evaluator
     = new ConditionalClassifierEvaluator<CharSequence>(cc,
                                          cats, storeInputs);
corpus.visitTest(evaluator);
```

We use an evaluator for conditional classifications, which we construct with the compiled classifier, the array of categories, and a flag indicating whether or not to store inputs, which we set to `false` here. If you want to inspect the true positives, false positives, etc., you need to set this flag to `true`.

After setting up the classifier evaluator, we call the corpus method `visitTest()` to supply the test cases to the evaluator's `handle()` method.

Rather than just printing the evaluator, which provides a very verbose dump, we have focused on the most common evaluation metrics for all-versus-all and one-versus-all evaluations. See Chapter 9 for a complete description of available classifier evaluations.

All-Versus-All Results

The first thing we do is extract the results for accuracy, and macro-averaged precision, recall, F-measure, and kappa for the overall system

```
ConfusionMatrix cm = evaluator.confusionMatrix();
int totalCount = cm.totalCount();
int totalCorrect = cm.totalCorrect();
double accuracy = cm.totalAccuracy();
double macroAvgPrec = cm.macroAvgPrecision();
double macroAvgRec = cm.macroAvgRecall();
double macroAvgF = cm.macroAvgFMeasure();
double kappa = cm.kappa();
```

These are then printed using code not displayed here. See Section 9.6 for more information on interpreting macro averaged results; they're essentially averages of the one-versus-all results, to which we turn next.

One-Versus-All Results

In one versus all evaluations, we treat the twenty newsgroup categories as being decomposed into twenty one-versus-all evaluations. In these cases, we treat each problem as a binary classification problem. See Section 9.4.8 for more information on one-versus-all evaluations. In code, we just loop over the categories, extracting the one-versus-all evaluations as precision-recall evaluation objects.

```
int[][] matrix = cm.matrix();
for (int k = 0; k < cats.length; ++k) {
     PrecisionRecallEvaluation pr = cm.oneVsAll(k);

     long tp = pr.truePositive();
     long tn = pr.trueNegative();
     long fp = pr.falsePositive();
     long fn = pr.falseNegative();

     double acc = pr.accuracy();

     double prec = pr.precision();
     double recall = pr.recall();
     double specificity = pr.rejectionRecall();
     double f = pr.fMeasure();
```

From these, we pull out the true and false positives and negative counts, overall accuracy, along with precision, recall (aka sensitivity), specificity (aka rejection recall), and F measure. These are then printed in code not shown.

Further, for each category, we print out the non-zero cells of the confusion matrix itself. These indicate counts for the number of times a document of a given category was classified as something else, and vice-versa, though we only show the first case.

```
     for (int k2 = 0; k2 < cats.length; ++k2)
         if (matrix[k][k2] > 0)
```

```
            System.out.println(" * => " + cats[k2]
                                 + " : " + matrix[k][k2]);
```

The confusion matrix as an integer array was created outside of the loop as part of the last code block using the `matrix()` method from the confusion matrix class. The entries `matrix[k1][k2]` indicate the number of times a document with true category `k1` was misclassified as being of category `k2`.

10.10.3 Running the Demo

There's an Ant target `20-news-corpus` that runs the demo. It reads the name of the tar gzipped data file from the property `20-news-file`, which we specify here on the command line.

```
> ant -D20-news-file=../../data-dist/20news-bydate.tar.gz
20-news-corpus
```

The first thing it prints out after it finishes is the confusion matrix accuracies for all-versus-all:

```
All Versus All
  correct/total = 5638 / 7532
  accuracy=0.749
  Macro Avg prec=0.785  rec=0.728   F=0.716
  Kappa=0.735
```

After this, it prints the one-versus-all results. For instance, consider the fourth category, *talk.politics.mideast*, for which we see the following results.

```
Category[17]=talk.politics.mideast versus All
  TP=313 TN=7137 FP=19 FN=63
  Accuracy=0.989
  Prec=0.943  Rec(Sens)=0.832  Spec=0.997  F=0.884
    * => alt.atheism : 6
    * => comp.graphics : 1
    * => comp.windows.x : 5
    * => misc.forsale : 1
    * => rec.autos : 1
    * => rec.motorcycles : 2
    * => rec.sport.baseball : 5
    * => rec.sport.hockey : 2
    * => sci.electronics : 3
    * => sci.space : 1
    * => soc.religion.christian : 34
    * => talk.politics.guns : 2
    * => talk.politics.mideast : 313
    alt.atheism => * : 5
    comp.graphics => * : 1
    sci.med => * : 2
    talk.politics.guns => * : 2
    talk.politics.misc => * : 5
    talk.religion.misc => * : 4
```

After dumping the number true and false positive and negatives, it prints the overall accuracy, which is 98.9% in this case. The other one-versus-all evaluations have similarly high accuracies in the high 90% range. But overall accuracy was only around 75%. The reason for this is that any negative is counted as correct, with overall accuracy being the number of true positives plus true negatives divided by the total cases. The true negatives dominate these numbers, and aren't sensitive to getting the correct category as long as it's not the category being evaluated one versus all. This is reflected in relatively high precision and specificity values (94.3% and 99.7% here), but lower recall values (83.2% here).

The remaining lines indicate the actual confusions, with an asterisk replacing the category being evaluated. For instance, six instances of documents which were in the *talk.politics.mideast* category were mistakenly classified as being of category *alt.atheism*. Similarly, there were five cases of documents whose true category was *sci.med* that were mistakenly classified as *talk.politics.mideast*.

10.11 Cross-Validating a Classifier

Our next demonstration is of cross validation, which were introduced in Section 2.4.

10.11.1 Implementing the Cross-Validating Corpus

The implementation of the cross-validating corpus for the Twenty Newsgroups data is in the class `TwentyNewsXvalCorpus` in this chapter's package, `naivebayes`. For the implementation, we extend LingPipe's `XValidatingObjectCorpus` class.[6]

```
public class TwentyNewsXvalCorpus
     extends XValidatingObjectCorpus<Classified<CharSequence>> {

     public TwentyNewsXvalCorpus(File corpusFileTgz,
                                 int numFolds)
          throws IOException {

          super(numFolds);
```

Note that we have supplied the generic argument `Classified<CharSequence>`, which is the type of classifications making up the training and evaluation data. We have a simple constructor taking the name of the gzipped tar file in which the corpus may be found as well as the number of folds for cross validation. The number of folds is used in the explicit call to `super(int)`, which is required because the only constructor for `XValidatingObjectCorpus` requires an integer

[6]In general, cross-validating object corpora may be used directly without. An equally perspicuous approach for this example, in which the extension defines no new methods, would be to define a static factory method that returned an instance of `XValidatingObjectCorpus`.

argument specifying the number of folds. The constructor is declared to throw an I/O exception if it has trouble reading the corpus.[7]

The actual parser has not been abstracted, but rather is copied almost verbatim from the example in Section D.2. The main action is in the loop driving over the entries in the tar input stream.

```
        TarInputStream tarIn = new TarInputStream(gzipIn);
        while (true) {
            TarEntry entry = tarIn.getNextEntry();
```

Within the block, we repeat the parsing code from Section D.2 to pull out the data (not shown here), until we can retrieve the text and classification.

```
            Classified<CharSequence> classified
                = new Classified<CharSequence>(text,c);
            handle(classified);
        }
        tarIn.close();
    }
```

The difference from the previous usage is that we call the cross-validating corpus method `handle(E)`, where `E` is the generic type of the corpus, here `Classified<CharSequence>`. The implementation is inherited from the superclass. This call to the handle method adds the classified character sequence to the corpus. When we are done, we close the tar input stream, and are ready to go.

10.11.2 Implementing the Evaluation Driver

Now that we have a cross-validating corpus, we can see how to use it to do a cross-validating evaluation. The example is implemented in `main()` method of class `TwentyNewsXvalDemo` in this chapter's package, `naivebayes`. As usual, we start the main method by reading command-line arguments.

```
File corpusFileTgz = new File(args[0]);
double catPrior = Double.valueOf(args[1]);
double tokPrior = Double.valueOf(args[2]);
double lengthNorm = args[3].equalsIgnoreCase("NaN")
    ? Double.NaN ? Double.valueOf(args[3]);
long randomSeed = Long.valueOf(args[4]);
```

As is usual in commands to do cross-validation, we have an argument for the data file. In addition, we have command-line arguments for the parameters we wish to tune, here the the category prior, the token prior, the length normalization. Finally, we have a parameter that controls the randomization used for cross-validation itself.

[7]As tempting as it may be to capture I/O exceptions locally and ignore them, we believe it is better to flag them in static corpora like Twenty Newsgroups so that parsing errors in the corpus may be found. This also makes coding easier, as the exceptions are simply propagated. In a production system, the marshalling of data would need to be made more robust.

Because we will reuse it, we also declare the tokenizer factory (here, it's really just a local variable shortcut to save space).

```
TokenizerFactory tokFact
     = IndoEuropeanTokenizerFactory.INSTANCE;
```

Next, we create the corpus itself using its constructor.

```
int numFolds = 10;
XValidatingObjectCorpus<Classified<CharSequence>> corpus
     = new TwentyNewsXvalCorpus(corpusFileTgz,numFolds);
corpus.permuteCorpus(new Random(randomSeed));

Set<String> catSet = Nb20NewsCorpus.getCatSet(corpus);
String[] cats = catSet.toArray(Strings.EMPTY_STRING_ARRAY);
```

After it is created, we permute using a randomizer based on the seed number supplied as a command-line argument. Then we extract the set of categories from the corpus using the static utility method, as previously shown in Section 10.10.2.

We're finally ready to create the evaluator. For cross-validation, it's typical to have an overall evaluation, which is used for every fold. Here is how it's constructed.

```
boolean store = false;
BaseClassifierEvaluator<CharSequence> globalEvaluator
     = new BaseClassifierEvaluator<CharSequence>(null,cats,
                                                  store);
```

Note that we supplied a `null` classifier argument. We will set the classifier for the evaluation once we have trained it within a fold.

We then set up a loop over the folds.

```
for (int fold = 0; fold < numFolds; ++fold) {
     corpus.setFold(fold);

     TradNaiveBayesClassifier classifier
         = new TradNaiveBayesClassifier(catSet,tokFact,catPrior,
                                        tokPrior,lengthNorm);
     corpus.visitTrain(classifier);
```

It loops over the folds (counting from zero). Within the loop, it sets the fold on the corpus using the `setFold(int)` method.

Then, it constructs a new classifier instance using the parameters read from the command line. Note how this is done within the loop body. If we had constructed a classifier on the outside, it would've been reused within the loop, with the result being that we'd be testing on training data as we went along.

After setting up the classifier, we call the corpus's `visitTrain()` method, supplying the new classifier instance as an argument. This causes all the training items in the corpus to be supplied to the classifier's `handle(Classified<CharSequence>)` method.

We next compile the classifier and use it for evaluation.

```
    @SuppressWarnings("unchecked")
    BaseClassifier<CharSequence> cClas
        = (BaseClassifier<CharSequence>)
        AbstractExternalizable.compile(classifier);

    globalEvaluator.setClassifier(cClas);
    corpus.visitTest(globalEvaluator);

    BaseClassifierEvaluator<CharSequence> localEvaluator
        = new BaseClassifierEvaluator<CharSequence>(cClas,
                                                     cats,
                                                     store);
    corpus.visitTest(localEvaluator);
```

The compilation is only for efficiency, and is carried out using a static utility method in LingPipe's `AbstractExternalizable` class.

Next, we set the global classifier evaluator's classifier to be the result of compiling our trained naive Bayes classifier. We then call the corpus's `visitTest()` method to send all the test instances to the evaluator's `handle(Classified<CharSequence>)` method. As a result, the global evaluator will have the evaluation statistics for all of the folds in a single evaluation.

After updating our global evaluation, we create a local evaluator that we will use just for the current fold. It's set up just like the global evaluator and invoked in the same way using the corpus's `visitTest()` method. The code to print the local evaluations per fold and the global evaluation in the end are not shown.

10.11.3 Cross Validating

There are two purposes of cross validation. One is to use as much evaluation data as possible to get a tighter read on the system's performance. The second uses the first to tune the parameters of the system. Because the same data is continually being reused, cross-validated performance tends to be better than held out performance on new data.

Running the Demo

The Ant target `20-news-xval` is setup up to run the `main()` method of `TwentyNewsXvalDemo`. The Ant target passes the values of these properties to the command. Here's an example illustrating all of the arguments.

```
>          ant -Dcat-prior=2.0 -Dtok-prior=0.1 -Dlength-norm=10.0
-Drandom-seed=85 20-news-xval

  20-news-file=C:\lpb\data-dist\20news-bydate.tar.gz
  cat-prior=2.0
  tok-prior=0.1
  length-norm=10.0
  random-seed=85
  tok-fact=com.aliasi.tokenizer.IndoEuropeanTokenizerFactory
```

```
FOLD 0  totalCount=  1884   acc=0.825   conf95=0.017
  macro avg: prec=0.847  rec=0.809  F1=0.800

FOLD 1  totalCount=  1885   acc=0.816   conf95=0.017
  macro avg: prec=0.836  rec=0.801  F1=0.786

...

FOLD 9
  totalCount=  1885   acc=0.831   conf95=0.017
  macro avg: prec=0.854  rec=0.813  F1=0.805

Global Eval
  totalCount= 18846   acc=0.827   conf95=0.005
  macro avg: prec=0.848  rec=0.808  F1=0.802
```

After printing the command-line arguments, it prints the results per fold. In this case, we have only chosen to print the count of examples in the fold, the overall accuracy (and 95% +/- value), and the macro-averaged results; any of the other evaluation statistics could be treated in the same way. Further, by replacing the base classifier evaluator with a conditional classifier evaluator, even more statistics would be available (at the cost of speed and storage).

The main thing to take away from this display is the variation between the folds. Each represents training on a random 90% of the data and testing on the remaining 10%. The confidence intervals are computed in the usual way from a normal approximation to a binomial distribution. As discussed in Section 9.4.2, the way to read these is as a 95% interval is as 0.825 ± 0.017.

An immediate takeaway message is that using three decimal places of accuracy to report these results is highly misleading. The confidence interval tells us taht we're roughly ± 0.02, so the second decimal place isn't even likely to be accurate. The rough validity of these confidence intervals can be seen from looking at the range of values produced. The way to interpret the 95% intervals resulting from one fold can best be explained by example. For instance, in fold 0, the accuracy estimate is 0.825 with a confidence interval of ± 0.017. This is saying that (roughly because of the normal approximation) if we truly had an 0.825 accuracy nad run 1884 trials (the total count for the fold), 95% of the time we would expect the measured accuracy estimate to fall in the interval 0.825 ± 0.017.

After the final fold (fold 9, because we numbered from 0), we print the global evaluation. This is based on all the cases, so note the tenfold higher total count. The accuracy here is just the average of the per-fold accuracies (weighted by count, of course). Note that because we have ten times as many evaluation points, our confidence interval is much tighter. Though it's still not tight enough to report more than two decimal places.

We can conclude with the settings we have that our system is roughly 83% accurate.

10.11.4 Tuning a System

The goal of tuning is to find the settings that work the best for a given system. Usually this is done by optimizing some measure of performance, such as accuracy or macro-averaged F measure. The two standard approaches have been illustrated already for naive Bayes, namely setting aside an official train/test split of the data and the second is by cross validation.

The advantage of cross validation is that it uses more data, which all else being equal, is a good thing. The main drawback to cross-validation is well illustrated with the twenty newsgroups data, namely that the set of examples in the corpus is not stationary. In particular, the twenty newsgroups corpus is the result of a typewritten dialog taking place on a newsgroup over time. The later articles are often closely related to the earlier articles by inclusion of quotes and responses. Thus they are related exactly and also topically. This is wholly typical of natural language and many other domains in which forecasting is much harder than backcasting (or "aftcasting" if you prefer a more nautical terminology).

One reason for this in language data is easily understood by imagining ten years of newswire data. Over time, different political figures and sports figures come into and out of public focus and different events come into and out of focus. Often with a cyclic schedule; we would not be surprised to find the news mentioning the word *holiday* and the phrase *new year* more often in December and January than in other months. The terms *World Cup* and *Winter Olympics* tend to occur on four-year cycles. Basically, language is not neatly homogeneous over time (what a statistician would call "stationary").

For tuning, it's usually easiest to start with reasonable default parameters and then move a single parameter at a time until results look roughly optimal, then move onto the next parameter, continuing to cycle through all the parameters until you run out of time, energy or improvements.

10.11.5 Tuning Naive Bayes

The three quantitative tuning parameters for naive Bayes are reflected in the command-line arguments to our demo, namely the two priors and the length normalization.

Category Prior

The category prior indicates the prior count of examples to be added to each category's count. During prediction, naive Bayes adds the prior count and observed count in training to derive an estimate of the probability of seeing a given category of text.

Increasing the category prior will drive the category distribution closer to uniform, meaning that the categories will have similar probabilities. This is easy to see with an example with two categories, A and B. Suppose our corpus has 10 instances of A and 40 instances of B. With a prior count of 0, the probability estimate for A is 40/(10+50), or 0.8. But if we have a prior count of 20, the estimate is (40+20)/(10+20 + 50+20), or 0.6. The higher prior drives the probability closer to 0.5, representing equal odds of either category appearing.

The twenty newsgroups is a balanced corpus, meaning that there are roughly the same number of items in each category. This is not typical of the real data from which it was gathered. In the wild, different newsgroups have wildly different numbers of messages posted to them. If someone were to give us a posting, we would ideally be able to use our knowledge of the frequency of postings as well as the text.

For balanced corpora like twenty newsgroups, the prior will have no effect. If ther training data counts were 20 and 20 for categories A and B, then the estimated probabilities are 0.5 no matter what the prior count. As a result, changing this parameter will have no effect on the outcomes.

As with all count-based priors, the more data that is seen, the smaller the effect of the prior. This is also easily seen by example. If we have a prior count of 10, then we get very different estimates with counts of 4 and 1, 40 and 10, and 4000 and 1000. As the size of the data increases, the estimates converge to the maximum likelihood estimate that arises from a zero prior.

Token Prior

Like the category prior, the token prior smooths estimates, specifically the estimate of the probability of seeing a given token in a document drawn from a particular category. This parameter has a profound effect on naive Bayes classifiers. For instance, here's the overall accuracies based on different choices of token prior, leaving the category prior at 2.0 and the length normalization at 10.0.

Token Prior	*Accuracy*
0.0001	0.91
0.0010	0.91
0.0100	0.89
0.1000	0.83
1.0000	0.69
10.0000	0.44

In this case, performance goes up as we lower the token count prior. Typically, performance will begin to drop as the prior is reduced below its optimal value. While sometimes a zero prior will work (it's equivalent to a maximum likelihood estimator), in practice it can result in zero probability estimates, which can lead to numerical instability in the evaluations. For instance, setting the prior to 0.0 will crash naive Bayes, but setting it to a very small value still gives performance of roughly 0.905.

For robustness, it's usually safest to leave the parameters close to where the performance begins to get good rather than to extreme values. For instance, from the above measurements, we'd set the token prior to 0.001, not 0.0001. By more robust, we mean that the performance will likely be better for test examples which are not exactly like the training examples.

If we were to continue to raise the prior, the performance would degrade to chance accuracy, which is 0.05 given the balanced 20 categories.

Tuning Length Normalization

The length normalization treats the entire document as if it were the length norm number of tokens long. Setting the value to `Double.NaN` turns off length normalization (that's done with command-line argument *NaN* in this demo program).

The length normalization has a greater effect on probabilility estimates than on first-best classification. Its effect on first-best results depends on the length of the document. If the document is shorter than the length norm, the document's model of tokens gets upweighted relative to the overall category distribution. If the document is longer than the length norm, the reverse happens, with more weight being given to the distribution over the categories than would occur in the unnormalized case.

Tuning Tokenization

Because naive Bayes runs purely off of the observed tokens, the tokenizer factory's behavior has a very large effect on how the classifier behaves. In these examples, we have used LingPipe's default tokenizer for Indo-European languages (like English).

It is common to apply a number of filters to a basic tokenizer. Specifically, it is common to case normalize, remove stop words, and reduce to a stemmed (or lemmatized form). See Chapter 3 for a thorough description of LingPipe's built-in tokenizers.

Although it is possible to use n-gram based tokenization for naive Bayes, either at the character or token level, these tokenizers further violate the assumption of independence underlying naive Bayes. For instance, if we have the word *fantastic* and are looking at 4-grams, then the first two tokens, *fant* and *anta* are highly related; in fact, we know that the character 4-gram following *fant* must begin with *ant*.

10.12 Formalizing Naive Bayes

The naive Bayes model has a particularly simple structural form. Setting aside document-length normalization, and assuming tokenization has already happened, the model is nothing more than a multinomial mixture model.

10.12.1 The Model

The easiest way to write it is using sampling notation.

Observed Data

The data we observe is structured as follows.

Data	*Description*
$K \in \mathbb{N}$	number of categories
$N \in \mathbb{N}$	number of documents
$V \in \mathbb{N}$	number of distinct tokens
$M_n \in \mathbb{N}$	number of words in document n
$x_{n,m} \in 1{:}V$	word m in document n
$z_n \in 1{:}K$	category of document n

Model Parameters

The parameters of the model are as follows.

Parameter	*Description*
$\alpha \in \mathbb{R}_+$	prior count for categories plus 1
$\beta \in \mathbb{R}_+$	prior count for tokens in categories plus 1
$\pi_k \in [0,1]$	probability of category k
$\phi_{k,v} \in [0,1]$	probability of token v in document of category k

Note that the Dirichlet prior parameters α and β are set to the prior counts (as defined in our interface) plus 1. In general, prior count parameters for a Dirichlet may be effectively negative; they are not allowed in LingPipe because they can easily lead to situations where there is no maximum a posteriori estimate (see Section 10.12.3).

Probability Model

The generative model for the entire corpus (all observed data and parameteters) is defined in sampling notation as follows.

$$\pi \sim \mathsf{Dirichlet}(\alpha) \tag{10.1}$$

$$\phi_k \sim \mathsf{Dirichlet}(\beta) \qquad \text{for } k \in 1{:}K \tag{10.2}$$

$$z_n \sim \mathsf{Discrete}(\pi) \qquad \text{for } n \in 1{:}N \tag{10.3}$$

$$x_{n,m} \sim \mathsf{Discrete}(\phi_{z_n}) \qquad \text{for } n \in 1{:}N, m \in 1{:}M_n \tag{10.4}$$

For us, we fix the prior parameters α and β to constants. Specifically, these values are one plus the prior counts as described in the API.[8]

To start, we generate the category discrete distribution parameter π using a Dirichlet prior. This parameter determines the distribution of categories in the (super)population.[9] Next, for each category k, we generate the discrete distribution parameter ϕ_k which models the probability of each token appearing at any point in a document of category k.

[8]This is because, in general, the prior count can be conceptually negative for Dirichlet priors; LingPipe doesn't accomodate this case, because the MAP estimates often diverge with prior counts below zero.

[9]The finite population sampled may have a different distribution of categories. This is easy to see. If I roll a six-sided die twelve times, it's unlikely that I'll get each number exactly twice even though each has a 1/6 probability in each throw. The parameter represents the distribution over a theoretical superpopulation. The law of large numbers states that as the corpus size (N) grows, the sampled proportions approach the superpopulation proportions.

Now, we are ready to generate the observed data. For each document n, we generate its category z_n based on the category distribution π. Then, for each word m in document n, we generate its token $x_{n,m}$ using the distributioin ϕ_{z_n} over words for category z_n. That's the entire model.

The sampling notation determines the entire probability of all of the parameters (excluding the constant prior parameters) and the observed data.

$$\begin{aligned} & p(\pi,\phi,z,x) && (10.5) \\ & = \mathsf{Dir}(\pi|\alpha) \times \prod_{k=1}^{K} \mathsf{Dir}(\phi_k|\beta) \times \prod_{n=1}^{N} \mathsf{Disc}(z_n|\pi) \times \prod_{n=1}^{N} \prod_{m=1}^{M_n} \mathsf{Disc}(w_{n,m}|\phi_{z_n}). \end{aligned}$$

10.12.2 Predictive Inference

For now, we will assume we know the parameters for a model; in the next section we show how to estimate them. Our prediction problem for classification is to model the probability distribution over categories given an observed document, which has the form

$$p(z_n|w_n,\pi,\phi). \tag{10.6}$$

Applying Bayes's rule and a little algebra reveals that

$$\begin{aligned} p(z_n|w_n,\pi,\phi) &= \frac{p(z_n,w_n|\pi,\phi)}{p(w_n|\pi,\phi)} && (10.7) \\ &\propto p(z_n,w_n|\pi,\phi) \\ &= p(z_n|\pi) \times \prod_{m=1}^{M_n} p(w_{n,m}|z_n,\phi). \end{aligned}$$

To compute the actual probabilities, we just renormalize, computing

$$p(z_n|\pi,\phi) = \frac{p(z_n|\pi) \times \prod_{m=1}^{M_n} p(w_{n,m}|z_n,\phi)}{\sum_{k=1}^{K} p(k|\pi) \times \prod_{m=1}^{M_n} p(w_{n,m}|k,\phi)}. \tag{10.8}$$

10.12.3 Parameter Estimation

Given training data, our job is to estimate the parameters for a model.[10]

What we do for naive Bayes is known as maximum a posteriori (MAP) estimate, because we are finding the parameters that maximize the Bayesian posterior $p(\pi,\phi|z,w)$, which represents the probability distribution over the parameters π and ϕ given the observed data z and w; here we have suppressed the implicit conditioning on the constants α, β, N, M, and K. The MAP estimates for parameters are written as $\hat{pi}$ and $\hat{\phi}$ and defined by

$$(\hat{\pi},\hat{\phi}) = \arg\max_{\pi,\phi} \; p(\pi,\phi|z,w). \tag{10.9}$$

[10]This is an instance of so-called "point estimation", which may be contrasted with full Bayesian inference. The latter averages over the uncertainty of the inferred parameters, whereas the former just assumes they are correct (or good enough). Bayesian inference tends to be more robust, but inferences based on point estimates are much easier and faster, and tend to provide similar results.

Because of the convenient form we have chosen for the priors, namely Dirichlet, this equation can be solved analytically (as long as the priors α and β are greater than or equal to 1). We just formalize what we said earlier, namely that we add the prior counts to the observed counts. For instance, for the category prevalence parameter π, we have

$$\hat{\pi}_k \propto \alpha + \sum_{n=1}^{N} \mathrm{I}(z_n = k), \tag{10.10}$$

where the indicator function $\mathrm{I}(z_n = k)$ has value 1 if the condition is true and value 0 otherwise. In words, its value is the count of training examples for which the category z_n was equal to k. This expression normalizes to

$$\hat{\pi}_k = \frac{\alpha + \sum_{n=1}^{N} \mathrm{I}(z_n = k)}{\sum_{k=1}^{K}(\alpha + \sum_{n=1}^{N} \mathrm{I}(z_n = k))} = \frac{\alpha + \sum_{n=1}^{N} \mathrm{I}(z_n = k)}{K \times \alpha + N}. \tag{10.11}$$

The denominator is derived as the total of counts contributed by training instances (N) and by the prior, which adds a prior count of α for each category ($K \times \alpha$).

10.12.4 Length Normalization

Length normalization in naive Bayes models normalizes documents so that they are effectively all of a specified length. For instance, if the length normalization is 10, every document is treated as if it were of length 10. This means that if the document is shorter than 10 tokens, each token counts more than once and if the document is longer that 10 tokens, the effect of each token is as if it showed up less than once.

Formally, length normalization is easiest to define by proportion. Recall from Equation 10.7 that the probabilty of a category z_n, given a sequence of tokens $w_{n,1}, \ldots, w_{n,M_n}$ given model parameters π and ϕ_z, is proportional to the probabilty of the category times the probabilty of the tokens in that category.

$$p(z_n|w_n, \pi, \phi) \propto p(x_n|\pi) \times \prod_{m=1}^{M_n} p(w_{n,m}|\phi_{z_n}). \tag{10.12}$$

Note that the probabilty of category z_n is proportional to the category probability $p(x_n|\pi)$ given category prevalence π, and the product of the probabilities of the individual words, $p(w_{n,m}|\phi_{z_n})$ in category z_n. The simple product of word probabilities arises because of the conditional independence assumed for words $w_{n,m}$ given category z_n. Note that in this form, each word counts once as a term in the product.

Suppose our length normalization parameter is $\lambda > 0$, indicating that we wish to treat every document as if it were really length λ rather than its actual length M_n. To rescale the product to be effectively λ terms long instead of the M_n terms long as it's defined, we raise it to the λ/M_n power,

$$p(z_n|w_n, \pi, \phi) \propto p(x_n|\pi) \times \left(\prod_{m=1} M_n p(w_{n,m}|\phi_{z_n})\right)^{\lambda/M_n} \tag{10.13}$$

First note that the normalization only applies to the token terms, not the category distribution terms. Second note that if the length norm and document length are the same, $\lambda = M_n$, there is no effect. Now if $\lambda > M_n$, we inflate the contribution of each term by a power of λ/M_n. If $\lambda < M_n$, raising to the power λ/M_n downweights each term.

so that the length norm λ is larger than the number of tokens M_n in the document, then we raise the product to a power less than 1.

Chapter 11

Tagging

Consider the problem of trying to assign a grammatical category such as noun or verb to every word in a sentence. This is what is known as a tagging problem and the labels assigned to the words are called part-of-speech tags.

One productive way of viewing a tagging problem is as a structured classification problem. A tagging for a given sequence of tokens consists of a sequence of tags, one for each token. These tags are like the categories assigned by a classifier, only now there is a whole sequence of such categories that must be assigned.

LingPipe provides two standard statistical taggers, hidden Markov models (HMM) and conditional random fields (CRF); we cover these in detail in Chapter 12 and Chapter 13, respectively. HMMs provide a generative model of tagging that is analogous to naive Bayes classifiers. CRFs provide a discriminative model of tagging that is closely related to logistic regression.

Like their analogous classifiers, LingPipe's taggers implement not only first-best tagging, but also n-best tagging at the sequence level based on sequence probabilities and n-best tagging at the token level based on marginal probabilities.

Like their analogous classifiers, taggers are defined through a suite of tagging-specific interfaces and evaluated with interface-specific evaluators. Before covering HMMs and CRFs, we introduce the tagging interfaces and evaluators in this chapter.

11.1 Taggings

LingPipe provides a base class for representing a complete tagging of a sequence as well as some specialized subclasses which we use heavily for n-best output and natural-language inputs.

LingPipe locates its general tagging interfaces and evaluators in the package `com.aliasi.tag`. Within that package, the generic class `Tagging<E>` represents taggings of sequences of type `E`. For example, `Tagging<String>` provides taggings over sequences of strings, such as tokens.

11.1.1 Base Tagging Class

A tagging is nothing more than a lightweight wrapper for the sequence of tags assigned to a sequence of symbols. Unless you write your own tagger, you will likely only need tagging objects that are produced by existing taggers.

Constructing a Tagging

The constructor `Tagging(List<E> syms, List<String> tags)` creates a tagging consisting of a list of symbols of type `E` and a parallel list of tags of type `String`. These lists must be the same length, because the tag at position i is taken to correspond to the symbol at position i; in Java terms, `tags.get(i)` is the tag for symbol `syms.get(i)`. If the tags are not the same length, the constructor will throw an illegal argument exception.

Accessing Tags and Symbols

The entire list of tags and list of tokens are available through the methods `tags()` and `tokens()`. Both of these methods return immutable views of the underlying tags and tokens as produced by the static utility method `unmodifiableList(List<T>)` in Java's utility class `Collections` in package `java.util`. The return result is a list, but it throws unsupported operation exceptions if an attempt is made to modify it using methods such as `add()`.

The tags and tokens may also be accessed on a position-by-position basis. The method `size()` returns the lengths of the lists of tags and symbols. For each integer between 0 (inclusive) and the size of the lists (exclusive), the methods `token(int)` and `tag(int)` return the symbol and tag at a specified position. If an integer out of this range is suppolied, the methods throw index out of bounds exceptions.

The object method `toString()` may be used to render the entire tagging in a human-readable format (assuming the symbols and tags are human readable, of course).

Like most of LingPipe's objects, taggings are immutable in the sense that once they are constructed, their content is constant in the sense can't be modified. One advantage of this design is that all of the tagging classes are entirely thread safe without any synchronization.

11.1.2 Scored Taggings

The class `ScoredTagging<E>` extends `Tagging<E>`, and thus serves as a tagging itself. Scored taggings also implement the general LingPipe interface `Scored` in `com.aliasi.util`.

Constructing Scored Taggings

As with regular taggings, these are most commonly created by taggers rather than by users. A scored tagging is constructured using

ScoredTagging(List<E>,List<String>,double), with the first two arguments passed to the supertype constructor Tagging(List<E>,List<String>) and the double being used for the score.

Like base taggings, scored taggings are immutable.

Accessing Scores

The Scored interface defines a single method score() returning a double-precision floating-point value. The advantage of encapsulating having a score in an interface is that it makes it easy to sort Scored instances by their scores. For example, there are predefined comparators for scored objects, accessible through static methods ScoredObject.comparator() and ScoredObject.reverseComparator() for sorting in increasing or decreasing order of score respectively. The class ScoredObject<E> provides a general implementation of Scored that also holds an immutable reference to an object of type E.

11.1.3 String Taggings

The class StringTagging also extends the base Tagging<E>, but with the generic type E fixed to String. In addition to the tokens and tags, a string tagging stores an underlying character sequence from which the tokens were derived along with the start and end position of each token in the sequence.

Constructing String Taggings

A string tagging is construccted using StringTagging(List<String> toks, List<String> tags, CharSequence cs, int[] tokenStarts, int[] tokenEnds). The tokens and tags are passed to the superclass constructor Tagging(List<String>,List<String>), so must be the same length. The arrays of token starts and token ends must also be the same length as the lists of tokens and tags; if they aren't, an illegal argument exception will be thrown. The character sequence is copied into a local string object.

Like the other tagging classes, string taggings are immutable.

Underlying String and Tokens

The method characters() returns a String corresponding to the sequence of charcters undelrying this tagging.

The methods tokenStart(int) and tokenEnd(int) provide the position of a token, with the start position being inclusive and the end position being exclusive. The method rawToken(int) is a convenience method to return the slice of the underlying string spanning the token at the specified psoition. As with the tag and token access methods, these methods throw index out of bounds execptions if the indices are out of range.

The string tagging class overrides toString() to provide human-readable output.

Equality and Hash Codes

The basic `Tagging` and extended `ScoredTagging` classes define equality by reference (by not overriding the definition of `equals(Object)` inherited from `java.util.Object`). That is, a tagging and scored tagging are only equal if they are reference equal. String taggings have an equality method defined in terms of their contents, with a corresponding hash code method.

11.2 Tag Lattices

The `Tagging` class provides an output representation for first-best taggers that return their best estimate of a tagging. The `ScoredTagging` class is sufficient for n-best output, which consists of a sequence of taggings with scores attached to each complete tagging. For marginal taggers, output takes the form of a tag lattice. A tag lattice encapsulates a rich computation of probabilities for individual tags given the rest of the output.

In this section, we describe the abstract class `TagLattice<E>`. Like for regular taggings, the generic type parameter `E` represents the type of symbols being tagged. Additional methods provide the means to estimate the probabilities of specific tags.

11.2.1 Constructing Tag Lattices

Tag lattices are constructed by marginal taggers and returned to clients. If you are not defining your own marginal tagger, you should not need to construct a tag lattice. There is a private implementation in the `tag` package if you'd like to examine a simple implementation.

There are concrete implementations for both HMMs and CRFs. The HMM-specific subclass is not public, so is not accessible through the API. If you're curious, the code is availabe in the class `TagWordLattice` in the package `com.aliasi.hmm`. There is a public subclass `ForwardBackwardTagLattice` in the `crf` package which we describe along with conditional random fields in Chapter 13; it provides some extra functionality beyond that provided by the abstract base class described in this section.

11.2.2 Accessing Tag Probabilities

The primary service provided by tag lattices is encapsulating probability estimates for tags for individual symbols. For instance, consider the sentence *The dog barked.*. With a standard English tokenization, such as LingPipe's built in `IndoEuropeanTokenizerFactory`), the sentence consists of four tokens *The*, *dog*, *barked*, and *.*

Given this sentence, with LingPipe's HMM and CRF taggers, it's possible to estimate the probabilty that the token *barked* is a past tense verb as well as the probability that it is a past particple verb. Note that because *barked* may be used for simple past tense or the past participle, it's not possible to determine the category by looking at just the word. Instead, we want to know what the chance *barked* is a past participle in the context of the sentence *The dog barked.* (the answer is that it should be pretty low; the existence of a preceding noun and the absence of an auxiliary verb form of *have* are both evidence that the word in question is a simple past tense usage.

Underlying Tokens

The underlying set of tokens may be retrieved using methods similar to the simple taggings. The number of tokens in the input is provided by the method `numTokens()`. The token at a given position is retrieved using `token(int)`, which returns an instance of `E`, the type of symbols in the tagging. An immutable view of the entire list of tokens is given by `tokenList()`.

Per-Token Marginal Probability Estimates

The simplest way to access the probability estimates for each tag for a given token is through the method `tokenClassification(int)`. Given an index, this method returns an instance of `ConditionalClassification`, which is able to provide the probability of each category for the token given the context of the entire input string (see Section 9.2.4 for more conditional classifications).

Tag Set as Symbol Table

Because a tag lattice provides a probability estimate for each tag for each token position, it is helpful to be able to access the entire set of possible tags. The tags are represented in a symbol table, a copy or immutable view of which is returned by `tagSymbolTable()`; see Chapter 5 for an overview of LingPipe's symbol tables. The basic idea of the symbol table is that it lets you retrieve an integer identifier for each tag and vice-versa.

It's also possible to retrieve the sequence of tags in symbol-table order using the method `tagList()`, which returns a list of strings.

Simple Tag Probabilities

The method `logProbability(int token, int tag)` returns the (natural) log of the probability of the token with the specified index being assigned to the tag with the specified identifier.

Tag Sequence Probabilities

Because the lattice stores such a rich set of probability information, it is possible to extend the calculation of the probability of a single category for a single token to the calculation of a sequence of categories for a sequence of tokens, all conditional on the entire input sequence. The method `logProbability(int n, int[] tags)` returns the (natural) log probability of the sequence of tags starting at position `n`. The tag positions must all be within the sequence, so that $0 \leq$ `n` and `n` $+$ `tags.length` $\leq$ `numTokens`.

11.3 Taggers

The `com.aliasi.tag` package defines three tagging interfaces, corresponding to the three kinds of tag outputs. The two statistical tagging implementations in LingPipe, hidden Markov models (HMM) and conditional random fields (CRF), implement all three interfaces.

The interfaces are broken out to allow simpler run-time interfaces. For instance, this lets us define separate evaluation interfaces for each mode of operation, as well as leaving open the option for taggers that do not implement all three interfaces.

11.3.1 First-Best Taggers

The first-best tagging interface is `Tagger<E>`, with the type parameter `E` determining the kind of tokens that are tagged. The interface defines a single method,

```
Tagging<E> tag(List<E> tokens);
```

That is, given a list of tokens of type `E`, return a first-best tagging over that type. The return type `Tagging` is described in Section 11.1.1.

11.3.2 *N*-Best Taggers

The n-best tagging interface is `NBestTagger<E>`, and like the first-best interface, the type parameter `E` determines the type of tokens that are tagged. The n-best tagger interface extends the simple `Tagger<E>` interface, so that any n-best tagger may be used as a simple tagger.

There are two methods specified in this interface, differing in terms of their score normalizations.

The first method is

```
Iterator<ScoredTagging<E>> tagNBest(List<E>,int);
```

where the input list is the list of tokens to parse and the integer argument specifies the maximum number of results to generate. Setting the maximum number of results lower than `Integer.MAX_VALUE` can save space during tagging by allowing taggings that are known to be beyond the result set to be pruned.

The return result is an iterator over scored taggings, which are described in Section 11.1.2. Essentially, a scored tagging combines a tagging with a score. The interface specifies that the scored taggings are returned in decreasing order of score. Thus the best tagging is returned first, the second-best next, and so on. This is what allows any n-best tagger to implement first-best tagging; it just returns the first-result from `tagNBest()` as the result for the superinterface's `tag()` method.[1]

The second method specifiedin the interface has a different name, but with identical argument and result types,

[1]This presupposes that there is always at least one tagging result, which is the case for all of the taggers of which we are aware. While the `tag()` method could be defined to return `null`, that would not play nicely with other interfaces which expect non-null taggings.

```
Iterator<ScoredTagging<E>> tagNBestConditional(List<E>,int);
```

The difference between this method and the simple `tagNBest()` method is that this method requires the scores to be conditional probabilities of the tagging given the input tokens.[2] The difference between n-best and conditionally normalized n-best taggers is the same as that between scored classifiers and conditional classifiers (see Section 9.2.4).

The `tagNBestConditional()` method is specified in the interface to throw an `UnsupportedOperationException` if the method is not supported. Thus it's possible to write an unnormalized n-best tagger.

11.3.3 Marginal Taggers

The third tagging interface is `MarginalTagger<E>`. It produces an entire tag lattice as output in oen method call, specified by the single interface method

```
TagLattice<E> tagMarginal(List<E>);
```

As with the other interfaces, the type parameter `E` picks out the type of the tokens that are being tagged.

The marginal tagger interface does not extend the basic tagging interface. That's because it's not in general possible to extract a coherent first-best result from a tag lattice, because the first-best tags on a per-token basis may not be consistent as an entire tag sequence; we'll see examples when we consider using taggers for chunking.

11.4 Tagger Evaluators

Like classifier evaluators, which we discussed in Chapter 9, there is a separate tagger evaluator for each of the tagging interfaces. Each of the the three interfaces is named for the tagger interface it evaluates.

11.4.1 First-Best Tagger Evaluation

The simplest evaluator is the `TaggerEvaluator<E>`, which evaluates taggers over type `E` tokens.

As with the classifier evaluators, a tagger evaluator is constructed from a tagger. The constructor,

```
TaggerEvaluator(Tagger<E>,boolean);
```

takes an argument for the tagger that will be evaluated. This tagger may be null at construction time and reset later using the method `setTagger(Tagger<E>)`. The tagger must be either supplied in the constructor or set before cases are evaluated. The second argument indicates whether the tokens in the test cases are saved along with the tags.

[2]Both the conditional probability normalization and the order are informal requirements, because they are beyond the expressive power of Java's type system.

Adding Test Cases

Like the classifier evaluators, the tagger evaluators implement object handlers, in this case implementing the interface `ObjectHandler<Tagging<E>>` (see Section 2.1), which specifies the single method

```
void handle(Tagging<E>);
```

As explained in Section 2.1 and the rest of Chapter 2, this allows the handlers to be plugged together with corpora and parsers.

The second way to add test cases is directly through the `addCase()` method, which has signature

```
void addCase(Tagging<E> ref, Tagging<E> resp);
```

The two tagging arguments represent the gold-standard reference tagging and the system's response, respectively.

Basic Input Profile

A tagger evaluator has various utility methods that provide information about the data that has been seen so far. The method `numCases()` returns the number of examples seen so far, which is just the number of calls to `handle()` or `addCase()`). The method `numTokens()` represents the total number of tokens seen so far. The method `tags()` returns a list of the tags seen so far, each of which is a string.

The method `inputSizeHistogram()` returns a histogram, represented as an object to counter map of integers,

```
ObjectToCounterMap<Integer> inputSizeHistogram();
```

The result maps integers representing the number of tokens in an input to the number of cases seen so far with that many input tokens.

The underlying tagger is returned by the method `tagger()` and the flag indicating whether or not to store tokens is returned by `storeTokens()`.

Evaluations

There are several methods that return the evaluation results. The simplest is `caseAccuracy()`, which returns a `double` between 0 and 1 representing the fraction of cases that were tagged completely correctly. A response tagging is case correct if every tag matches matches the corresponding reference tagging.

The method `tokenEval()` returns an evaluation on a token-by-token basis. It treats the tagging of each token as a classification problem, and hence is able to return an instance of `BaseClassifierEvaluator<E>`, where `E` is the type of the tokens. This provides all the evaluations that are available for base classifiers.

The token-by-token evaluation can be restricted to a set of tokens not in a given set. The main use for this is to evaluate performance on tokens that were not seen in the training data. The method

```
BaseClassifierEvaluator<E> unknownTokenEval(Set<E>);
```

takes a set of tokens as an argument, and returns an evaluation restricted to tokens not in the argument set. If the argument set is the set of tokens in the training data, the resulting evaluation is on tokens that are not in the training set.

Chapter 12

Tagging with Hidden Markov Models

Hidden Markov models (HMMs) provide a generative statistical model of tagging (see Chapter 11 for a general discussion of taggers). HMMs may be thought of as a sequentially structured generalization of naive Bayes classifiers (see Chapter 10). Like language models (see Chapter 6), their sequential structure is Markovian. What this means is that we only use a finite window of past history in order to predict the tag for a given token. Unlike language models, the state depends on previous tags, not on previous tokens. These tags are not directly observable, which is the etymology of the term “hidden” in their name.

As with classifiers, many tasks that do not superficially appear to involve tagging, such as named-entity chunking, can be reduced to tagging problems.

Chapter 13

Conditional Random Fields

Chapter 14

Latent Dirichlet Allocation

Latent Dirichlet allocation (LDA) provides a statistical approach to document clustering based on the words (or tokens) that appear in a document. Given a set of documents, a tokenization scheme to convert them to bags of words, and a number of topics, LDA is able to infer the set of topics making up the collection of documents and the mixture of topics found in each document. These assignments are soft in that they are expressed as probability distributions, not absolute decisions.

At the basis of the LDA model is the notion that each document is generated from its own mixture of topics. For instance, a blog post about a dinner party might 70% about cooking, 20% about sustainable farming, and 10% about entertaining.

Applications of LDA include document similarity measurement, document clustering, topic discovery, word association for search, feature extraction for discriminative models such as logistic regression or conditional random fields, classification of new documents, language modeling, and many more.

14.1 Corpora, Documents, and Tokens

For LDA, a corpus is nothing more than a collection of documents. Each document is modeled as a sequence of tokens, which are often words or normalized forms of words.

Applications of LDA typically start with a real document collection consisting of character data. LingPipe uses tokenizer factories to convert character sequences to sequences of tokens (see Chapter 3). Utilities in the LDA implementation class help convert these to appropriate representations with the help of symbol tables (see Chapter 5).

For LDA, and many other models such as traditional naive Bayes classification and k-means clustering, only the count of the words is significant, not their order. A bag is a data structure like a set, only with (non-negative) counts for each member. Mathematically, we may model a bag as a mapping from its members to their counts. Like other models that don't distinguish word order, LDA is called a bag-of-words model.

14.2 LDA Parameter Estimation

In this section, we will cover the steps necessary to set up the constant parameters for an LDA model and then estimate the unknown parameters from a corpus of text data.

14.2.1 Synthetic Data Example

We begin with a synthetic example due to Steyvers and Griffiths.[1] Synthetic or "fake data" models play an important role in statistics, where they are used like unit tests for statistical inference mechanisms.

In this section, we will create an LDA model by fixing all of its parameters, then use it to generate synthetic data according to the model. Then, we will provide the generated data to our LDA inference algorithm and make sure that it can recover the known structure of the data.

The implementation of our synthetic example, which illustrates the basic inference mechanism for LDA, is in the demo class `SyntheticLdaExample`.

Tokens

Steyvers and Griffiths' synthetic example involved only two topics and five words. The words are *river*, *stream*, *bank*, *money*, and *loan*. The first two words typically occur in articles about rivers, and the last two words in articles about money. The word *bank*, the canonical example of an ambiguous word in English, occurs in both types of articles.

What's a Topic?

A topic in LDA is nothing more than a discrete probability distribution over words. That is, given a topic, each word has a probability of occurring, and the sum of all word probabilities in a topic must be one.

Steyvers and Griffiths' example involves two topics, one about banking and one about rivers, with the following word probabilities.

	river	*stream*	*bank*	*money*	*loan*
Topic 1	1/3	1/3	1/3	0	0
Topic 2	0	0	1/3	1/3	1/3

Topic 1 is about water and topic 2 about money. If a word is drawn from topic 1, there is a 1/3 chance it is *river*, a 1/3 chance it is *stream* and a 1/3 chance it is *bank*.

What's a Document?

For the purposes of LDA, a document is modeled as a sequence of tokens. We use a tokenizer factory to convert the documents into counts. The identity of

[1]From Steyvers, Mark and Tom Griffiths. 2007. Probabilistic topic models. In Thomas K. Landauer, Danielle S. McNamara, Simon Dennis and Walter Kintsch (eds.), *Handbook of Latent Semantic Analysis*. Laurence Erlbaum.

the tokens doesn't matter. It turns out the order of the tokens doesn't matter for fitting the model, though we preserve order and identity so that we can label tokens with topics in our final output.

The code in SyntheticLdaExample starts out defining an array of sixteen character sequences making up the document collection that will be clustered. These documents were generated randomly from an LDA model by Steyvers and Griffiths. The texts making up the documents are defined starting with

```
static final CharSequence[] TEXTS = new String[] {
    "bank loan money loan loan money bank loan bank loan"
    + " loan money bank money money money",

    "loan money money loan bank bank money bank money loan"
    + " money money bank loan bank money",
```

and ending with

```
    "river stream stream stream river stream stream bank"
    + " bank bank bank river river stream bank stream",

    "stream river river bank stream stream stream stream"
    + " bank river river stream bank river stream bank"
};
```

The only reason we used concatenation is so that the lines would fit on the pages of the book. The complete set of documents is most easily described in a table.

Doc ID	*river*	*stream*	*bank*	*money*	*loan*
0	0	0	4	6	6
1	0	0	5	7	4
2	0	0	7	5	4
3	0	0	7	6	3
4	0	0	7	2	7
5	0	0	9	3	4
6	1	0	4	6	5
7	1	2	6	4	3
8	1	3	6	4	2
9	2	3	6	1	4
10	2	3	7	3	1
11	3	6	6	1	0
12	6	3	6	0	1
13	2	8	6	0	0
14	4	7	5	0	0
15	5	7	4	0	0

The first document (identifier/position 0) has four instances of *bank*, six of *money* and six of *loan*. Each document has sixteen tokens, but equal sized documents is just an artifact of Steyvers and Griffiths' example. In general, document sizes will differ.

Tokenizing Documents

LDA deals with token identifiers in the form of a matrix, not with string tokens. But there's a handy utility method to produce such a matrix from a corpus of texts and a tokenizer factory. The first few lines of the `main()` method in the `SyntheticLdaExample` class provide the conversion given our static array `TEXTS` of character sequences:

```
TokenizerFactory tokFact
    = new RegExTokenizerFactory("\\p{L}+");
SymbolTable symTab = new MapSymbolTable();
int minCount = 1;
int[][] docWords
    = tokenizeDocuments(TEXTS,tokFact,symTab,minCount);
```

The regex-based tokenizer factory is used with a regex defining tokens as maximal sequences of characters in the Unicode letter class (see Section 3.2.3 for regex tokenizer factories and for Unicode regex classes, the section of the chapter on regular expressions in the companion volume, *Text Processing in Java*). This will pull out each of the words in our documents.

Next, we create a symbol table (see Section 5.2) which we will use to provide integer indexes to each of the tokens.

The variable `minCount` will specify how many instances of a token must be seen in the entire set of documents for it to be used in the model. Here we set the threshold to 1, so no tokens will be pruned out. For most applications, it makes sense to use a slightly larger value.

Then, we create a two-dimensional array of document-word identifiers. The value of `docWords[n]` is an array of the symbol identifiers from the symbol table for the words in document `n` (which will range from 0 (inclusive) to 15 (inclusive) for our 16-document corpus).

The program prints the symbol table and document-word array so you may see what is actually produced,

```
symbTab={0=stream, 1=money, 2=bank, 3=loan, 4=river}

docWords[ 0] = { 2, 3, 1, 3, 3, 1, 2, 3, 2, 3, 3, 1, 2, 1, 1, 1 }
docWords[ 1] = { 3, 1, 1, 3, 2, 2, 1, 2, 1, 3, 1, 1, 2, 3, 2, 1 }
...
docWords[14] = { 4, 0, 0, 0, 4, 0, 0, 2, 2, 2, 2, 4, 4, 0, 2, 0 }
docWords[15] = { 0, 4, 4, 2, 0, 0, 0, 0, 2, 4, 4, 0, 2, 4, 0, 2 }
```

Given the symbol table, the first document (index 0) has tokens 2 (*bank*), 3 (*loan*), 1 (*money*), and so on, matching the input.

It is this document-word matrix that is input to LDA.

14.2.2 LDA Parameters

Before running LDA, we need to set up a number of parameters, which we will explain after seeing the example run. The first of these are the model parameters,

```
short numTopics = 2;
double docTopicPrior = 0.1;
double topicWordPrior = 0.01;
```

First, we set up the number of topics, which must be fixed in advance. Here we cheat and set it to 2, which we happen to know in this case is the actual number of topics used to generate the documents. In general, we won't know the best value for the number of topics for a particular application.

Sampler Parameters

Next, we set up the variables that control the sampler underlying the LDA inference mechanism.

```
int burnin = 0;
int sampleLag = 1;
int numSamples = 256;
Random random = new Random(43L);
```

These are the number of samples we take, the number of samples thrown away during the burnin phase, and the period between samples after burnin (we explain these parameters in full detail below).

The sampler also requires a pseudo-random number generator. We use Java's built-in pseudo-random number generator in the class `Random` in `java.util` (see Appendix C.1 for more information bout `Random` and using it in experimental settings). By providing the constructor an explicit seed, `43L`, the program will have the same behavior each time it is run. Remove the argument or provide a different seed to get different behavior.

Reporting Callback Handler Parameters

After that, we set up a handler that'll be used for reporting on the progress of the sampler. Specifically, we define a callback class `ReportingLdaSampleHandler` (the code of which we describe below), which will be passed to the inference method.

```
int reportPeriod = 4;
ReportingLdaSampleHandler handler
     = new ReportingLdaSampleHandler(symTab,reportPeriod);
```

The callback handler will receive information about each sample produced and provides the flexibility to perform online Bayesian inference (which we explain below). It's constructed with the symbol table and takes an argument specifying the period between reports (measured in samples).

Invoking the Sampler

At last, we are in a position to actually call the LDA inference method, which is named after the algorithmic approach, Gibbs sampling, which we'll also define below,

```
GibbsSample sample
     = gibbsSampler(docWords,
                    numTopics, docTopicPrior, topicWordPrior,
                    burnin, sampleLag, numSamples,
                    random, handler);
```

Basically, we pass it all of our parameters, including the document-token matrix.

After the method returns, we have a sample in hand. We use the reporting methods in the handler to print out information about the final sample. It requires the final sample and parameters controlling verbosity (printing at most 5 words per topic and 2 topics per document, which here, are not actually constraints).

```
int wordsPerTopic = 5;
int topicsPerDoc = 2;
boolean reportTokens = true;
handler.reportParameters(sample);
handler.reportTopics(sample,wordsPerTopic);
handler.reportDocs(sample,topicsPerDoc,reportTokens);
```

14.3 Interpreting LDA Output

We run the example using the Ant target synthetic-lda to run the main() method in SyntheticLdaExample. a

```
> ant synthetic-lda
```

We've already seen the output of the symbol table and the document-token matrix above.

14.3.1 Convergence Monitoring

The next thing that's output is monitoring information from the sampler.

```
n=     0 t=     :00 x-entropy-rate=  2.1553
n=     4 t=     :00 x-entropy-rate=  2.1528
...
n=    20 t=     :00 x-entropy-rate=  1.7837
n=    24 t=     :00 x-entropy-rate=  1.7787
...
n=   248 t=     :00 x-entropy-rate=  1.7811
n=   252 t=     :00 x-entropy-rate=  1.7946
```

The results here are typical. The report for the sample number. Note that we only get a report for every four samples because of how we parameterized the handler. We can report every epoch, but it's relatively expensive. The second column is the elapsed time. This example is so slow, it takes less than a second total. The third column is the cross-entropy rate. This is roughly a measure of the expected number of bits required to code a word if we were to use the

LDA model for compression.[2] Note that the cross-entropy starts high, at 2.1 bits, but quickly drops to 1.8 bits. Further note that at the converged state, LDA is providing a better model than randomly generating words, which would have a cross-entropy of $-\log_2 1/5 \approx 2.3$ bits.

The goal in running Gibbs samplers is to reach convergence, where numbers like cross-entropy rates stabilize. Note that Gibbs sampling is not an optimization algorithm, so while the cross-entropy rates start high and get lower, once they've reached about 1.78 in this example, they start bouncing around.

The number of samples used for burnin should be large enough that it gets you to the bouncing around stage. After that, results should be relatively stable (see below for more discussion of LDA stability).

14.3.2 Inferred Topics

Next, we get a report of the parameters we've fit for the topic models themselves.

```
TOPIC 0  (total count=104)
SYMBOL             WORD    COUNT   PROB          Z
--------------------------------------------------
     0           stream       42   0.404       4.2
     2             bank       35   0.336      -0.5
     4            river       27   0.260       3.3

TOPIC 1  (total count=152)
SYMBOL             WORD    COUNT   PROB          Z
--------------------------------------------------
     2             bank       60   0.395       0.5
     1            money       48   0.316       3.1
     3             loan       44   0.289       3.0
```

We fixed the number of topics at 2. LDA operates by assigning each token to a topic (as we will see in a second). These reports show the most frequent words associated with each topic along with the symbol table ID and the number of times each word was assigned to a topic. At the top of the report is the total number of words assigned to that topic (just a sum of the word counts).

The probability column displays the estimated probabiltiy of a word in a topic. Thus if we sample a word from topic 0, it is 40.4% likely to be *stream*, 33.6% likely to be *bank* and 26.6% likely to be *river*. Note that these probabilities add up to 1. Also note that these are not probabilities of topics given words. The probability of *bank* in topic 1 is only 39.5% and thus it should be evident that these can't be probabilities of topics given words, becuase they don't add up to 1 (33.6% + 39.5% = 78.1% < 1).

These probabilities estimated from the small set of randomly generated data are as close as could be expected to the actual probability distributions. With larger data sets, results will be tighter.

[2]This figure ignores the generation of the multinomial for the document. For use in compression, we would also need to encode a discrete approximation of the draw of the document's multinomial from the Dirichlet prior.

The order of the topics is not determined. With a different random seed or stopping after different numbers of samples, we will get slightly different results. One difference may be that the identifiers on the topics will be switched. This lack of identifiability of the topic identifier is typical of clustering models. We return to its effect on Bayesian inference below.

The final column provides a z score for the word in the topic. This value measures how much more prevalent a word is in a topic than it is in the corpus as a whole. The units are standard deviations. For instance, the word *bank* appears 95 times in the corpus of 256 words, accounting for 95/256, or approximately 37% of all tokens. In topic 0, it appears 35 times in 104 tokens, accounting for approximately 34% of all tokens; in topic 1, it appears 60 times in 152 words, accounting for roughly 39% of the total number of words. Because the prevalence of *bank* in topic 0 is less than in the corpus (34% versus 37%), its z score is negative; because it's higher in topic 1 than in the corpus (39% versus 37%), the z score is positive.

14.3.3 Inferred Token and Document Categorizations

The final form of output is on a per-document basis (controlled by a flag to the final output method). This provides a report on a document-by-document basis of how each word is assigned.

```
DOC 0
TOPIC    COUNT    PROB
----------------------
    1       16   0.994

bank(1) loan(1) money(1) loan(1) loan(1) money(1) bank(1)
loan(1) bank(1) loan(1) loan(1) money(1) bank(1) money(1)
money(1) money(1)
...
DOC 8
TOPIC    COUNT    PROB
----------------------
    1       11   0.685
    0        5   0.315

money(1) bank(1) money(1) loan(1) stream(0) bank(1) loan(1)
bank(1) bank(1) stream(0) money(1) bank(0) bank(1) stream(0)
river(0) money(1)
...
DOC 15
TOPIC    COUNT    PROB
----------------------
    0       16   0.994

stream(0) river(0) river(0) bank(0) stream(0) stream(0)
stream(0) stream(0) bank(0) river(0) river(0) stream(0) bank(0)
river(0) stream(0) bank(0)
```

We've only shown the output from three representative documents. For each document, we show the topic assignment to each token in parens after the token. All the words in document 0 are assigned to topic 1 whereas all the word in document 15 are assigned to topic 0. In document 8, we see that 11 words are assigned topic 1 and 5 words topic 0. The word *bank* is assigned to both topic 0 and topic 1 in the document. This shows how LDA accounts for documents that are about more than one topic.

The aggregate counts show how many words are assigned to each topic, and the probability that a word from a document is chosen from the probability.

Even though all words were assigned to topic 1 in document 0, the probability is still only 0.994. This is because of the priors, which induce smoothing, setting aside a small count for unseen topics in each document. This kind of smoothing is immensely helpful for robustness in the face of noisy data, and almost all natural language data is immensely noisy.

14.4 LDA's Gibbs Samples

The return value of the LDA estimation method is a Gibbs sample.

14.4.1 Constant, Hyperparameter and Data Access

The samples store all the argments supplied to the LDA method other than number of samples information. These values will be constant for the chain of samples produced by the iterator.

The method `numTopics()` returns the constant number of topics specified in the original invocation. The methods `topicWordPrior()` and `documentTopicPrior()` return the two hyperparameters (i.e., constant prior parameters).

The sample object also provides access to the underlying document-word matrix matrix through the methods `numDocuments()`, `documentLength(int)`, and `word(int,int)`.

14.4.2 Word Topic Assignments and Probability Estimates

What we're really after is the actual content of the sample. Each step of sampling contains an assignment of each token in each document to one of the topics. The method `topicSample(int,int)` returns the `short`-valued topic identifier for the specified document and token position.

These assignments of topics to words, combined with the priors' hyperparameters, determine the probability estimates for topics and documents we showed in the output in the previous section. The method `topicWordProb(int,int)` returns the `double`-valued probability estimate of a word in a topic. These probability distributions fix the behavior of the topics themselves.

The method `documentTopicProb(int,int)` returns the `double`-valued proability estimate for a topic in a document. These probabilities summarize the topics assigned to the document.

These probability estimates, coupled with the underlying document-word data matrix, allows us to compute `corpusLog2Probability()`, which is the probability of the topics and words given the models. Note that it does not include the probabilities of the document multinomials or the topic multinomials given their Dirichlet priors. The main reason they're excluded is that the multinomials are not sampled in the collapsed sampler.

The static utility method `dirichletLog2Prob(double[],double[])` in the class `Statistics` in LingPipe's `stats` package may be used to compute Dirichlet probabilities. We may plug the maximum a posteriori (MAP) estimates provided by the methods `topicWordProb()` and `documentTopicProb()` into the `dirichletLog2Prob()` method along with the relevant prior, `topicWordPrior()` and `documentTopicPrior()`, respectively.

14.4.3 Retrieving an LDA Model

Each Gibbs sample contains a factory method `lda()` to construct an instance of `LatentDirichletAllocation` based on the topic-word distributions implied by the sample and the prior hyperparameters.

Although the Gibbs samples themselves are not serializable, the LDA models they return are. We turn to the use of an LDA instance in the next section. The method we used in the last section to invoke the sampler returned a single Gibbs sample. There is a related static estimation method in `LatentDirichletAllocation` that returns an iterator over all of the Gibbs samples. The iterator and one-shot sampler return instances of `GibbsSample`, which is a static class nested in `LatentDirichletAllocation`.

Samples are produced in order by the sampler. The poorly named method `epoch()` returns the sample number, with numbering beginning from 0.

14.5 Handling Gibbs Samples

In our running synthetic data example, we provided an instance of the demo class `ReporingLdaSampleHandler` to the LDA estimation method `gibbsSampler()`. The use of the handler class follows LingPipe's generic callback handler pattern. For every Gibbs sample produced by the sampler, the handler object receives a reference to it through a callback.

14.5.1 Samples are Reused

Unlike many of our other callbacks, the Gibbs sampler callbacks were implemented for efficiency. Rather than getting a new Gibbs sample object for each sample, the same instance of `GibbsSample` is reused. Thus it does not make any sense to create a collection of the Gibbs samples for later use. All statistics must be computed online or accumulated outside the Gibbs sample object for later use.

14.5.2 Reporting Handler Demo

The reporting handler demo is overloaded to serve two purposes. First, it handles Gibbs samples as they arrive as required by its interface. Second, it provides a final report in a class-specific method.

Online Sample Handler

The class declaration, member variables and constructor for the handler are as follows.

```
public class ReportingLdaSampleHandler
     implements ObjectHandler<GibbsSample> {

     private final SymbolTable mSymbolTable;
     private final long mStartTime;
     private final int mReportPeriod;

     public ReportingLdaSampleHandler(SymbolTable symbolTable,
                                      int reportPeriod) {
         mSymbolTable = symbolTable;
         mStartTime = System.currentTimeMillis();
         mReportPeriod = reportPeriod;
     }
```

We maintain a reference to the symbol table so that our reports can reassociate the integer identifiers with which LDA works with the tokens from which they arose. We see how the report period is used in the implementation of the handler method.

The class was defined to implement the interface `ObjectHandler<GibbsSample>`. It thus requires an implementation of the `void` return method `handle(GibbsSample)`. Up to the print statement, this method is implemented as follows.

```
public void handle(GibbsSample sample) {
     int epoch = sample.epoch();
     if ((epoch % mReportPeriod) != 0) return;
     long t = System.currentTimeMillis() - mStartTime;
     double xEntropyRate
         = -sample.corpusLog2Probability() / sample.numTokens();
```

This is the callback method that will be called for every sample. It first grabs the epoch, or sample number. If this is not an even multiple of the report period, the method returns without performing any action. Otherwise, we fall through and calcualte the elapsed time and cross-entropy rate. Note that the cross-entropy rate as reported here is the negative log (base 2) probability of the entire corpus of documents divided by the number of tokens in the entire corpus. This cross-entropy rate is what we saw printed and what we monitor for convergence.

General Reporting

There are additional reporting methods in the handler above the required one that provide more detailed reporting for a sample. They produce the output shown in the previous section after the online reporting of cross-entropy rates finishes and the final sample is produced by LDA's `gibbsSampler()` method.

The general parameter reporting method, `reportParameters()`, just prints general corpus and estimator parameters from the sample. There are two more interesting reporters, one for topics and one for documents. The topic reporting method, `reportTopics()`, begins as follows.

```
public void reportTopics(GibbsSample sample, int maxWords) {
     for (int topic = 0; topic < sample.numTopics(); ++topic) {
          int topicCount = sample.topicCount(topic);
          ObjectToCounterMap<Integer> counter
              = new ObjectToCounterMap<Integer>();
          for (int word = 0; word < sample.numWords(); ++word)
              counter.set(word,
                           sample.topicWordCount(topic,word));
          List<Integer> topWords
              = counter.keysOrderedByCountList();
```

The loop is over the topic identifiers. Within the loop, the method begins by assigning the number of words assigned to a topic. It then allocates a general LingPipe object counter to use for the word counts. Each word identifier is considered int he inner loop, and the counter's value for that word is set to the number of times the word was assigned to the current topic. After collecting the counts in our map, we create a list of the keys, here word indexes, ordered in descending order of their counts.

We continue by looping over the words by rank.

```
           for (int rank = 0;
                rank < maxWords && rank < topWords.size();
                ++rank) {

               int wordId = topWords.get(rank);
               String word = mSymbolTable.idToSymbol(wordId);
               int wordCount = sample.wordCount(wordId);
               int topicWordCount
                   = sample.topicWordCount(topic,wordId);
               double topicWordProb
                   = sample.topicWordProb(topic,wordId);
               double z = binomialZ(topicWordCount,
                                    topicCount,
                                    wordCount,
                                    sample.numTokens());
```

We bound the number of words by the maximum given and make sure not to overflow the model's size bounds. Inside the loop, we just pull out identifiers and counts, and calculate the z-score. This is where we need the symbol table in order to print out the actual words rather than just their numbers. The rest of the method is just printing.

The method used to calculate the z-score is

```
static double binomialZ(double wordCountInDoc,
                        double wordsInDoc,
                        double wordCountinCorpus,
                        double wordsInCorpus) {
     double pCorpus = wordCountinCorpus / wordsInCorpus;
     double var = wordsInCorpus * pCorpus * (1 - pCorpus);
     double dev = Math.sqrt(var);
     double expected = wordsInDoc * pCorpus;
     return (wordCountInDoc - expected) / dev;
}
```

It first calculates the maximum likelihood estimate of the word's probabilty in the corpus, which is just the overall count of the word divided by the number of words in the corpus. The maximum likelihood variance estimate is calculated as usual for a binomial outcome over n samples (here wordsInCorpus) with known probabiltiy θ (here pCorpus), namely $n \times \theta \times (1 - \theta)$. The expected number of occurrences if it had the corpus distribution is given by the number of words in the document times the probability in the corpus. The z-score is then the actual value minus the expected value scaled by the expected deviation.

Returning to the general reporting, we have a method reportDocs(), which provides a report on the documents, optionally including a word-by-word report of topic assignments.

```
public void reportDocs(GibbsSample sample, int maxTopics,
                       boolean reportTokens) {
     for (int doc = 0; doc < sample.numDocuments(); ++doc) {
         ObjectToCounterMap<Integer> counter
             = new ObjectToCounterMap<Integer>();
         for (int topic = 0;
              topic < sample.numTopics();
              ++topic)
             counter.set(topic,
                         sample.documentTopicCount(doc,topic));
         List<Integer> topTopics
             = counter.keysOrderedByCountList();
```

This method enumerates all the documents in the corpus, reporting on each one in turn. For each topic, we follow the same pattern as we did for the topics beofre, establishing a LingPipe object counter mapping topic identifiers to the number of times a word in the current document was assigned to that topic. We then sort as before, then loop over the ranks up to the maximum number of topics or total number of topics in the model.

```
         for (int rank = 0;
              rank < topTopics.size() && rank < maxTopics;
```

```
             ++rank) {
            int topic = topTopics.get(rank);
            int docTopicCount
                = sample.documentTopicCount(doc,topic);
            double docTopicProb
                = sample.documentTopicProb(doc,topic);
```

For each rank, we calculate the counts and probabilities and display them.

Finally, we report on a token-by-token basis if the flag is set.

```
        if (!reportTokens) continue;
        int numDocTokens = sample.documentLength(doc);
        for (int tok = 0; tok < numDocTokens; ++tok) {
            int symbol = sample.word(doc,tok);
            short topic = sample.topicSample(doc,tok);
            String word = mSymbolTable.idToSymbol(symbol);
```

Here we just enumerate over the tokens in a document printing the topic to which they were assigned.

14.5.3 General Handlers

In general, handlers can do anything. For instance, they can be used to accumulate results such as running averages, thus supporting full Bayesian inference with a small memory footprint (only the current sample need stay in memory).

For instance, the following handler would calculate average corpus log 2 probability over all samples.

```
public class CorpusLog2ProbAvgHandler
     implements ObjectHandler<GibbsSample> {

     OnlineNormalEstimator mAvg = new OnlineNormalEstimator();

     public void handle(GibbsSample sample) {
         mAvg.handle(sample.corpusLog2Probability());
     }

     public OnlineNormalEstimator avg() {
         return mAvg;
     }
}
```

We use an instance of LingPipe's `OnlineNormalEstimator` for the calculation. It calculates averages and sample variances and standard deviations online without accumulating the data points. The method `mAvg.mean()` returns the sample mean and `mAvg.varianceUnbiased()` the unbiased sample variance.

Along these sample lines, more complex statistics such as word or document similarities or held out data entropy estimates may be computed online.

14.6 Scalability of LDA

LDA is a highly scalable model. It takes an amount of time per token that is proportional to the number of topics. Thus with a fixed number of topics, it scales linearly in topic size. Tens of thousands of documents are no problem on desktop machines.

14.6.1 Wormbase Data

In this section, we consider a larger scale LDA estimation problem, in analyzing a collection of biomedical research paper citations related to the model organism *Caenorhabditis elegans* (C. elegans), the nematode worm. The data was downloaded from the WormBase site (see Section D.4 for mroe information). Similar data from a slightly different source on a smaller scale was analyzed by Blei et al. in an attempt to relate the topics inferred using LDA with those added for genetics by the database curators.[3] An example of a citation in raw form is

```
%0 Journal Article
%T Determination of life-span in Caenorhabditis elegans
   by four clock genes.
%A Lakowski, B
%A Hekimi, S
%D 1996
%V 272
%P 1010-1013
%J Science
%M WBPaper00002456
%X The nematode worm Caenorhabditis elegans is a model system
   for the study of the genetic basis of aging. Maternal-effect
   mutations in four genes--clk-1, clk-2, clk-3, and gro-1
   --interact genetically to determine both the duration of
   development and life-span. Analysis of the phenotypes of
   physiological clock in the worm. Mutations in certain genes
   involved in dauer formation (an alternative larval stage
   induced by adverse conditions in which development is
   arrested) can also extend life-span, but the life extension
   of Clock mutants appears to be independent of these genes.
   The daf-2(e1370) clk-1(e2519) worms, which carry life-span-
   extending mutations from two different pathways, live nearly
   five times as long as wild-type worms.
```

There are not actually line breaks in the %T and %X elements.

[3]Blei et al. analyzed a smaller set of worm citations from the Caenorhabditis Genetic Center (CGC) bibliography. This data is available from the WormBook site at `http://www.wormbook.org/wli/cgcbib`.

Blei, D. M., K. Franks, M. I. Jordan and I. S. Mian. 2006. Statistical modeling of biomedical corpora: mining the Caenorhabditis Genetic Center Bibliography for genes related to life span. *BMC Bioinformatics* **7**:250.

Blei et al. considered 50 topics generated from 5225 documents using a total of 28,971 distinct words.

14.6.2 Running the Demo

First, you'll need to download the gzipped data from wormbase; see Section D.4. It doesn't matter where you put it; the program takes the location as a parameter.

We created an Ant target `lda-worm` to run the demo. It is configured to provide seven command-line arguments as properties, `wormbase.corpus.gz` for the location of the gzipped corpus, `min.token.count` for the minimum token count in the corpus to preserve the word in the model, `num.topics` for the number of topics to generate, `topic.prior` and `word.prior` for the priors, `random.seed` for the random seed, and `num.samples` for the total number of samples to draw.

```
> ant -Dwormbase.corpus.gz="../../data-dist/2007-12-01-wormbase-literature.endr
-Dmin.token.count=5 -Dnum.topics=50 -Dtopic.prior=0.1
-Dword.prior=0.001 -Drandom.seed=43 -Dnum.samples=500
-Dmodel.file=wormbase.LatentDirichletAllocation
-Dsymbol.table.file=wormbase.SymbolTable lda-worm > temp.txt
```

It takes about half an hour to generate the 500 samples on my workstation, and quite a bit more time to generate the roughly 50MB of output reporting on the topics and the assignment of tokens in documents. Because there's so much output, we redirect the results to a file; you can use the command `tail -f` to follow along in real time.

The output begins by reporting on the parameters and the corpus itself, the interesting part of which reports corpus statistics rather than what we entered.

```
...
#articles=26888 #chars=43343525
Number of unique words above count threshold=28499
Tokenized.  #Tokens After Pruning=3731693
```

We see that there are 26,888 total citations, made up of around 43 million characters. After tokenization, there are roughly 28 thousand distinct tokens that each appeared at least 5 times (the minimum token count), and a total token count of around 3.7 million instances. That means each Gibbs sampling step is going to reassign all 3.7 million tokens.

After the corpus statistics, it reports on the iterator's progress, as before.

```
n=     0 t=     :03 x-entropy-rate= 11.6898
n=     5 t=     :21 x-entropy-rate= 11.3218
n=    10 t=     :39 x-entropy-rate= 10.9798
n=    15 t=     :57 x-entropy-rate= 10.8136
n=    20 t=    1:15 x-entropy-rate= 10.7245
...
n=   485 t=   28:50 x-entropy-rate= 10.3250
n=   490 t=   29:08 x-entropy-rate= 10.3246
n=   495 t=   29:26 x-entropy-rate= 10.3237
...
```

We only ran for 500 samples, and at that point, we still haven't converged to the lowest cross-entropy rate. We ran the sampler overnight, and it hadn't converged even after 10,000 samples. Because LDA's an approximate statistical process anyway, stopping before convergence to retrieve a single sample isn't going to make much of a difference in terms of the clustering output.[4]

The program then reports the topic distributions and document distributions and token topic assignments, all based on the 500th sample. These look different than the ones we saw in the synthetic example, because there are many more longer documents (though stil short, being based on abstracts) and many more tokens.

The first few words in the first few topics are.

```
     [java] TOPIC 0  (total count=70201)
     [java] SYMBOL             WORD    COUNT   PROB          Z
     [java] --------------------------------------------------
     [java]  18616          disease     1136   0.016      30.6
     [java]  10640        infection      686   0.010      25.7
     [java]  26059         pathogen      658   0.009      25.2
     [java]   4513            fatty      576   0.008      23.6
     [java]   1863         toxicity      638   0.009      23.4
     [java]  20474             drug      732   0.010      21.2
     [java]  26758           immune      394   0.006      19.0
...
     [java]  23394        alzheimer      163   0.002      10.6
     [java]   5756            agent      216   0.003      10.6
     [java]  20684         clinical      128   0.002      10.4

     [java] TOPIC 1  (total count=88370)
     [java] SYMBOL             WORD    COUNT   PROB          Z
     [java] --------------------------------------------------
     [java]  17182    transcription     3813   0.043      51.3
     [java]   6619           factor     4383   0.050      49.7
     [java]    717          binding     1913   0.022      25.0
     [java]  21676  transcriptional     1215   0.014      24.6
     [java]  17200       regulatory     1347   0.015      24.6
     [java]  22813          element     1433   0.016      22.7
     [java]  12394             zinc      594   0.007      20.6
     [java]  13680           target     1423   0.016      19.7
     [java]  11864           finger      597   0.007      19.2
...
     [java]  11043  tissue-specific      188   0.002       8.2
     [java]   9171          hypoxia      111   0.001       7.9
     [java]  22550              dna      829   0.009       7.6
```

Overall, the topics seem coherenent and well separated.

Blei et al.'s paper analyzing the same data was concerned with seeing that topics about genes grouped genes that participate in the same pathways to show up in the same topic. This is easy to see in the data, with genes that operate

[4]On the other hand, if we wanted to estimate posterior variance, or get a very tight estimate on the posterior means, we would have to run to convergence.

together being highly clustered in the output. For instance, topic 10 is about sensory-motor genes,

```
[java] TOPIC 10  (total count=37931)
[java] SYMBOL              WORD    COUNT   PROB          Z
[java] --------------------------------------------------
[java]  12604             touch     1785   0.047      41.3
[java]  23411             mec-4      926   0.024      30.1
[java]  15424            unc-86      620   0.016      24.6
[java]  23416             mec-3      577   0.015      23.8
[java]  16331           channel     1547   0.041      23.3
[java]  20518               ion      771   0.020      22.5
[java]  22644               deg      327   0.009      17.9
[java]  23421             mec-7      309   0.008      17.4
```

whereas topic 21 is about genes that degrade RNA, including top tokens such as *rnai*, *interference*, *dsrna*, *silencing* and the gene anmes *ego-1* and *rde-1* and *rrf-1*, all of which are involved in RNA silencing through interference.

The documents are longer and more diverse in topics than in the synthetic example. For instance, here's a document that contains many words assigned to topic 10, which was shwon above.

```
DOC 1301

TOPIC    COUNT    PROB     TOPIC    COUNT    PROB
----------------------     ----------------------
   10       36   0.354         8        2   0.021
   44       33   0.325        23        2   0.021
    2       14   0.138        11        1   0.011
   31        4   0.040        17        1   0.011
   20        3   0.030        26        1   0.011

mec-4(10) gene(44) member(10) family(2) gen(10) mutate(10) induce(44)
neuronal(10) degeneration(10) dominant(10) mutation(10) mec-4(10)
gene(11) needed(10) mechanosensation(10) cause(10) touch-receptor(10)
neuron(10) degenerate(10) deg-1(10) another(31) gene(2) mutate(10)
induce(10) neuronal(10) degeneration(10) similar(2) sequence(20)
mec-4(10) defin(2) new(2) gene(2) family(2) cross-hybridizing(8)
sequenc(8) detectable(17) spec(20) raising(44) possibility(2)
degenerative(10) condition(23) organism(20) caused(10) mutation(31)
similar(2) gen(10) dominant(10) mec-4(10) mutation(10) affect(10)
amino(2) acid(2) effect(23) amino-acid(2) substitution(2) position(2)
suggest(10) steric(10) hindrance(10) induce(44) degenerative(10)
state(26) ad(44) department(44) biological(44) scienc(44)
columbia(44) university(44) new(31) york(44) new(31) york(44) fau(44)
driscoll(10) mau(44) driscoll(10) mfau(44) chalfie(10) mau(44)
chalfie(10) mla(44) engsi(44) genbank(44) genbank(44) genbank(44)
genbank(44) genbank(44) genbank(44) genbank(44) genbank(44)
genbank(44) genbank(44) journal(44) articlecy(44) englandta(44)
naturejid(44) imsb(44)
```

14.6.3 Code Walkthrough

The code is all in the class `LdaWorm`. The main differences from the synthetic demo include the parser for the compressed data and a custom tokenizer for the Wormbase data.

The method to read the corpus into an array of character sequences is based on LingPipe's `FileLineReader` utility (see the section on LingPipe's line reader in the I/O chapter of the companion volume, *Text Processing in Java*). It just scans the lines and accumulates texts and adds them to an accumulating list.

The tokenization scheme is more interesting. We create it with the following method.

```
static final TokenizerFactory wormbaseTokenizerFactory() {
     String regex = "[\\x2D\\p{L}\\p{N}]{2,}";
     TokenizerFactory factory
         = new RegExTokenizerFactory(regex);
     Pattern alpha = Pattern.compile(".*\\p{L}.*");
     factory = new RegExFilteredTokenizerFactory(factory,alpha);
     factory = new LowerCaseTokenizerFactory(factory);
     factory = new EnglishStopTokenizerFactory(factory);
     factory = new StopTokenizerFactory(factory,STOPWORD_SET);
     factory = new StemTokenizerFactory(factory);
     // repeated to remove stemmed and unstemmed stopwords
     factory = new StopTokenizerFactory(factory,STOPWORD_SET);
     return factory;
}
```

It creates a regex-based tokenizer that includes hyphens, letters and numbers as parts of tokens. We include hyphens and allow alphanumeric mixtures to include tokens for gene or protein names such as *p53* and *daf-19*, which would otherwise be separated. It also requires at least two characters.

It then immediately filters out tokens that are not at least two characters long and don't contain at least one letter. This filter could've been rolled into the base regex, but these kinds of exclusions quickly get clunky when mixed with alternations like in our base tokenizer factory.

We then convert the tokens to lower case and remove standard English stop words. We also remove custom stop words we discovered from looking at the distribution of tokens in the corpus. This list contains domain-specific words like *elegan* (remember the "plural" stripping) and non-topical words from the biomedical research domain such as *however* and *although*. We also remove the names of numbers, such as *eleven* and Roman numerals, such as `viii`, which are prevalent in this data. Note that we have already lowercased before stoplisting, so the stop list consists only of lowercase entries.

The last filter is our own custom stemmer, which was designed to follow the approach to removing English plural markers outlined in Blei et al.'s paper. Here's the code.

```
static class StemTokenizerFactory
     extends ModifyTokenTokenizerFactory {
```

```
    public StemTokenizerFactory(TokenizerFactory factory) {
        super(factory);
    }

    public String modifyToken(String token) {
        Matcher matcher = SUFFIX_PATTERN.matcher(token);
        return matcher.matches() ? matcher.group(1) : token;
    }

    static final Pattern SUFFIX_PATTERN
        = Pattern.compile("(.+?[aeiouy].*?|.*?[aeiouy].+?)"
                          + "(ss|ies|es|s)");
```

This filter fairly crudely strips off suffixes consisting of simple English plural markers, ensuring that what's left behind is at least two characters long. It uses relucatant quantifiers on the contexts ensuring the match of the plural suffix is as long as possible. Thus we map *fixes* to *fix* not *fixe*, and map *theories* to *theori*, not *theorie*. In contrast, we reduce *lies* to *lie*, because we can't match the first group with a vowel if we pull of *-ies*, but we map *entries* to *entr*.

This crude stemmer will overstrip in the situation where a word ends with one of these suffixes but is not plural. It just checks for suffixes and strips them off. If the result is too short or no longer contains a vowel, we return the original stem. (The restriction to ASCII is OK here because the documents were all normalized to ASCII by the bibliography curators.)

14.7 Understanding the LDA Model Parameters

Given a tokenizer and corpus, there are three key parameters that control the behavior the LDA model: the number of topics and the two priors.

14.7.1 Number of Topics

The number of topics parameter controls the number of topics into which the model factors the data. The main effect of increasing the number of topics is that the topics may become more fine grained. If there are only 10 topics to categorize all the worlds' news, we will not discover small topics like American baseball or Australian politics.

As the number of topics increases, the amount of data available to estimate each topic goes down. If the corpus is very large, it may be possible to fit a large number of topics effectively.

Which number of topics works best will depend on the application and the size and the shape of data. In the end, like any clustering or other exploratory data analysis technique, the best solution is to use trial and error. Explore a range of different topic sizes, such as 10, 50, 100, and 500.

The time to process each token is proportional to the total number of topics. It takes ten times as long to generate a sample for a 500-topic model than a 50-topic model.

Document-Topic and Topic-Word Priors

The two priors control how diffuse the models are. Technically, as we explain in Section 14.11, the priors are additive smoothing terms for the model of topics in a document and the model of words in a topic.

In LDA, each document is modeled as a mixture of different topics. When the document-topic prior is low, the model is encouraged to model a document using fewer topics than when the document-topic prior is high. What is going on probabilistically is that the prior adds to the probability of all topics, bringing the distribution over topics in a document closer to uniform (where all topics have equal probability).

Similarly, each topic is modeled as a distribution over words. When the topic-word prior is high, each topic is encouraged to be more balanced in the probabilities it assigns to each word. As with the document-topic prior, it moves the distribution of words in topics closer to the uniform distribution.

As with the number of topics, it will likely take some experimentation to find reasonable priors for a problem. A reasonable starting point for both priors is the inverse number of outcomes. For instance, with 50 topics, 1/50 is a good starting prior for the document-topic prior, and with 30,000 words, 1/30,000 is a reasonable starting point for the topic-word prior. It's unlikely you'll want much smaller priors than these, but raising the priors above these levels will lead to smoother topic and document models, and may be more reasonable for some applications.

14.8 LDA Instances for Multi-Topic Classification

We can use instances of the `LatentDirichletAllocation` class to perform classification on documents that were not in the training corpus. These new documents are classified with multiple topics and each word is assigned to a topic just as when LDA is applied to a corpus of documents.

14.8.1 Constructing LDA Instances

An instance of LDA represents a particular parameterization of LDA's multi-topic document model. Specifically, it stores the document-topic prior (the "Dirichlet" part of LDA) and a set of topic models in the form of distributions over words.

Direct Construction

The LDA class has a single constructor, `LatentDirichletAllocation(double,double[])`, which takes a document-topic prior and array of topic-word distributions. You'll notice that this does not include a symbol table or tokenizer. LDA runs purely as a multinomial model, and all symbol processing and tokenization happen externally.

With an external means of generating distributions over words (say, with a traditional naive Bayes classifier or K-means clusterer), we can create an LDA instance from the word distributions (and document-topic prior).

Construction from a Sample

More commonly, we will construct an LDA instance using a Gibbs sample generated from a corpus using the static method `gibbsSample()`. We saw an example of this in the stability evaluate example in Section 14.10.

At the end of the Wormbase example in Section 14.6.3, after we'd generated the Gibbs sample and assigned it to variable `sample`, we snuck the following lines of code.

```
static final TokenizerFactory wormbaseTokenizerFactory() {
     String regex = "[\\x2D\\p{L}\\p{N}]{2,}";
     TokenizerFactory factory
         = new RegExTokenizerFactory(regex);
     Pattern alpha = Pattern.compile(".*\\p{L}.*");
     factory = new RegExFilteredTokenizerFactory(factory,alpha);
     factory = new LowerCaseTokenizerFactory(factory);
     factory = new EnglishStopTokenizerFactory(factory);
     factory = new StopTokenizerFactory(factory,STOPWORD_SET);
     factory = new StemTokenizerFactory(factory);
     // repeated to remove stemmed and unstemmed stopwords
     factory = new StopTokenizerFactory(factory,STOPWORD_SET);
     return factory;
}
```

This shows the production of an LDA instance from a sample using the `lda()` method on the sample. Then we use LingPipe's abstract extenalizable utility (see the section on `AbstractExternalizable` in the chapter on I/O in the companion volume, *Text Processing in Java*) to write its serialized form to a file. We also serialize the map symbol table to file.

We will be able to deserialize this model from disk and use it to classify new documents that were not in the corpus used to generate the chain of samples.

14.8.2 Demo: Classifying with LDA

In the sample class `LdaClassifier`, we show how a serialized LDA model and symbol table We provide a sample program that illustrates the procedure for using an LDA model and symbol table for classification.

Code Walkthrough

The code is all in the `main()` method, and starts by assigning a model file `modelFile`, symbol table file `symbolTableFile`, text `text`, and random seed `randomSeed` from the command-line parameters. We then deserialize the two models and recreate the tokenizer factory.

```
@SuppressWarnings("unchecked")
LatentDirichletAllocation lda
     = (LatentDirichletAllocation)
     AbstractExternalizable.readObject(modelFile);
```

```
@SuppressWarnings("unchecked")
SymbolTable symbolTable
     = (SymbolTable)
     AbstractExternalizable.readObject(symbolTableFile);

TokenizerFactory tokFact = LdaWorm.wormbaseTokenizerFactory();
```

We have to cast the deserialized object and suppress the warnings the unchecked cast would otherwise generate. Deserialize also throws a `ClassNotFoundException`, which the `main()` method is also declared to throw in addition to an `IOException`.

We next marshal the parameters needed for classification and run the clasisfier.

```
int[] tokenIds
     = LatentDirichletAllocation
     .tokenizeDocument(text,tokFact,symbolTable);

int numSamples = 100;
int burnin = numSamples / 2;
int sampleLag = 1;
Random random = new Random(randomSeed);

double[] topicDist
     = lda.bayesTopicEstimate(tokenIds,numSamples,burnin,
                              sampleLag,random);
```

We use the tokenizer factory and symbol table to convert the text to a sequence of token identifiers in the symbol table, using the static utility method `tokenizeDocument()` built into LDA. We then set parameters of the topic estimation method, including the number of samples, burnin, sample lag and random number generator, all of which have the same interpretation as for the main LDA Gibbs sampling method. Finally, we call LDA's `bayesTopicEstimate()` method with the tokens and Gibbs sampling parameters.

What happens under the hood is that we run a chain of Gibbs samples just like we did for LDA itself. But rather than returning an iterator over the samples or a single sample, we return an average over all the samples. Because the topics are fixed by the LDA model, this produces (an approximation of) the Bayesian estimate of the topic distribution for the text. Usually we don't need many samples because we don't need multiple decimal places of accuracy.

What we get back is an array of doubles `topicDist`, indexed by topic. Topic `k` has probability `topicDist[k]`. The rest of the code just prints out the top 10 of these in order of probability.

Running the Demo

The Ant target `lda-classify` runs the classification demo, using properties `model.file`, `symbol.table.file`, `text`, and `random.seed` as command-line arguments.

For an example, we chose a MEDLINE citation about C. elegans that was more recent than the corpus used to estimate the LDA model.[5]

> Evolution of early embryogenesis in rhabditid nematodes. The cell-biological events that guide early-embryonic development occur with great precision within species but can be quite diverse across species. How these cellular processes evolve and which molecular components underlie evolutionary changes is poorly understood. To begin to address these questions, we systematically investigated early embryogenesis, from the one- to the four-cell embryo, in 34 nematode species related to C. elegans. We found 40 cell-biological characters that captured the phenotypic differences between these species. By tracing the evolutionary changes on a molecular phylogeny, we found that these characters evolved multiple times and independently of one another. Strikingly, all these phenotypes are mimicked by single-gene RNAi experiments in C. elegans. We use these comparisons to hypothesize the molecular mechanisms underlying the evolutionary changes. For example, we predict that a cell polarity module was altered during the evolution of the Protorhabditis group and show that PAR-1, a kinase localized asymmetrically in C. elegans early embryos, is symmetrically localized in the one-cell stage of Protorhabditis group species. Our genome-wide approach identifies candidate molecules—and thereby modules—associated with evolutionary changes in cell-biological phenotypes.

We set the property `text` to this value in the Ant file, but do not show it on the command line. We run the command as follows.

```
> ant -Dmodel.file=wormbase.LatentDirichletAllocation
-Dsymbol.table.file=wormbase.SymbolTable -Drandom.seed=42
lda-classify
```

```
 Topic    Pr          Topic    Pr
------ -----         ------ -----
    20 0.310              3 0.017
     6 0.254             38 0.015
    25 0.229             35 0.014
    21 0.077              7 0.005
    26 0.024             32 0.004
```

Like LDA as a whole (see Section 14.10), we get similar results for the bigger topics with different random seeds. Even with the small number of samples, 100, we used here. For instance, using seed 195334, we get very similar values for the top 8 topics.

```
> ant lda-classify -Drandom.seed=195334 ...
```

```
 Topic    Pr          Topic    Pr
------ -----         ------ -----
    20 0.308             26 0.025
     6 0.252             38 0.015
```

[5]Brauchle M., K. Kiontke, P. MacMenamin, D. H. Fitch, and F. Piano. 2009. Evolution of early embryogenesis in rhabditid nematodes. *Dev Biol.* 335(1). `doi:10.1016/j.ydbio.2009.07.033`.

```
    25 0.237              35 0.014
    21 0.063               9 0.009
     3 0.027              13 0.007
```

Let's take a look at the top words in the top three topics, which account for 80% of the tokens in the document or so.

```
TOPIC 20  (total count=94511)
SYMBOL             WORD    COUNT   PROB         Z
--------------------------------------------------
 22828           intron     1527   0.016      35.8
 19980         briggsae     1438   0.015      32.7
 10114             spec     1911   0.020      29.3
  8465         splicing     1002   0.011      29.0
  8619           splice      812   0.009      27.5
 15121             exon     1129   0.012      26.3
  3883        evolution      921   0.010      23.4
 13935          sequenc     1599   0.017      23.4
 15569          spliced      566   0.006      22.2
 10613         sequence     2043   0.022      20.6

TOPIC 6  (total count=147967)
SYMBOL             WORD    COUNT   PROB         Z
--------------------------------------------------
 10528            model     2535   0.017      30.7
 13442         research      925   0.006      27.2
 27073           system     2621   0.018      27.1
  3378          network     1035   0.007      25.2
 22863         organism     1742   0.012      24.5
 26328          biology      910   0.006      24.2
 11563           review      537   0.004      22.0
  5439             data     1707   0.012      21.8
 18566             tool      654   0.004      21.7
  8185      information      938   0.006      21.4


TOPIC 25  (total count=89810)
SYMBOL             WORD    COUNT   PROB         Z
--------------------------------------------------
 27843          spindle     2629   0.029      50.1
 16152           embryo     4915   0.055      47.0
 17879        microtubul     971   0.011      30.4
  8200        cytokinesi     868   0.010      27.9
 20537         polarity     1272   0.014      26.0
 23475              par      680   0.008      25.5
  3098       microtubule     729   0.008      25.1
 18067        centrosome     655   0.007      25.0
 18114           cortex      640   0.007      24.7
```

```
  7615         cortical      592   0.007      23.3
```

These topics are reasonable. Topic 20 is largely about evolution as it relates to splicing and introns, topic 6 is about systems biology and this is definitely a systems biology paper, and topic 25 is about embryos and the family of *par* genes that control them (including the gene *par-1* mentioned in the article at rank 20 or so.

With different random seeds, the LDA models produced from a corpus will vary. This will also affect document similarlity. It's possible to use full Bayesian analysis to average over all this uncertainty (see Section 14.8.3).

14.8.3 Bayesian Inference with an LDA Model

In the last section, we showed how to provide the Bayesian point estimate for the topic distributions. This is a reasonable estimate in that it minimizes the expected difference between the estimates and the "true" values (given the model). We could then plug these estimates in to reason about the document.

The basic problem with point estimates is that they do not encode the variance of the estimate. Some of the numbers may be more or less certain than others, and this can affect downstream inference.

The LDA class also provides support for full Bayesian inference in classification.[6] For Bayesian inference, we collect a group of samples, reason about each sample, then average the results. This may seem like what we did in the last section, but there we averaged over the Gibbs samples before performing inference. With full Bayesian inference, we do the averaging of the result of reasoning with over each Gibbs sample.

To retrieve a set of Gibbs samples, the LDA method `sampleTopics()`, taking exactly the same arguments in the same sequence as `bayestopicEstimat()`, produces an array of samples, each assigning each token to a single topic. Each member of the two-dimensional array assigns each token in the document to a topic.

The point estimation method we described in the last section for classification could be reconstructed by averaging the result of sampling (with the priors used for estimation). Roughly, we just count up all the words assigned to a given topic and divide by the total number of words; in reality, we also apply the prior.

14.9 Comparing Documents with LDA

After running LDA on a corpus, or on fresh documents using the classifier, it's possible to use the output of LDA to compare documents to each other. Each document gets a distribution over topics. These may be compared to each other using KL-divergence in exactly the same way as we compared topics to each other (see Section 14.10).

[6]What we discuss uses a point estimate of the LDA model. If we used more than one LDA model, we could start producing Bayesian estimaes one level back. The problem with doing that is that the topics are not identified in LDA, so we couldn't use multiple Gibbs samples of LDA parameters for classification directly.

The benefit of comparing documents this way is that two documents that share few or even no words may be considered similar by being about the same topic. For instance, a document mentioning *p53* (a tumor suppressor gene) will be related to a document mentioning *tumor* and *suppressor*, even if the documents have no other words in common. This is because *p53* often mentioned in the same documents as *tumor* and *suppressor*, so they will belong to similar topics.

Comparing documents is a situation in which we can usefully apply full Bayesian inference (see Section 14.8.3), because it doesn't depend on the identity of topics across samples when comparing documents.

14.10 Stability of Samples

Because LDA uses a random initialization, the results will not be the same each time it is run with a different random seed. For instance, a roughly similar topic in two runs may be topic number 17 in one run and topic 25 in another run. The topics may even drift over time in very, very long chains of samples.

The issue is then one of whether the topics are stable independent of their numbering. It turns out that across runs, many topics tend to be relatively stable, but other topics will vary. This idea could even be used to decide how many topics to select, restricting the number to those that are stable.

Steyvers and Griffiths tested the stability of topics by running LDA over the same corpus from two different seeds, then comparing the set of topics that are produced. Their basic approach is to line up the topics in the two samples based on the similarity of their probability distributions. In this section, we replicate their approach with a demo. The code is in demo class `LdaTopicSimilarity`.

14.10.1 Comparing Topics with Symmetrized KL Divergence

Because topics are nothing more than probability distributions over tokens, to compare topics, we need a way to compare probabiilty distributions. Here, we follow Steyvers and Griffiths in using symmetrized Kullback-Leibler (KL) divergence (see Section B.5.6 for definitions and examples).

Steyvers and Griffiths produced two chains of Gibbs samples, both of which they ran for a large nubmer of iterations. They then compared the last samples from each chain to each other. Specifically, they compared every every topic in the final sample from chain 1 against every topic in the final sample from chain 2.

The basis of their comparison was a greedy alignment. Conceptually, if we put the pairwise scores in order from lowest divergence to highest, we walk down the list of pairs, retaining every pair for which neither topic has been consumed. That way, every topic only shows up once on each side of the pairing. The choices are greedy in that we always select the least divergent pair of topics to add next from among the topics that haven't yet been assigned.

14.10.2 Code Walkthrough

The code to run the example is in `LdaTopicSimilarity`. Most of the work is done in the `main()` method of the command. It's declared to throw not only an I/O exception, but also an interrupted exception, because we are going to run each chain in its own thread.

```
public static void main(String[] args)
     throws IOException, InterruptedException {
```

After that, the method parses the text, creates the tokenizer factory, and extracts the tokens the same way as in our earlier Wormbase demo, using the utility methods introduced in the demo class `LdaWorm`.

We run two chains concurrently in their own threads. We basically just create the runnable from the doc-token matrix and random seed, create threads for teh runnable, start the threads, join them so our `main()` method's thread waits for the newly spawned threads to complete. We then extract the LDA model from the result, where they are stored in the member variable `mLda`. Finally, we compute the scores, then print them out.

```
     long seed1 = 42L;
     long seed2 = 43L;
     LdaRunnable runnable1 = new LdaRunnable(docTokens,seed1);
     LdaRunnable runnable2 = new LdaRunnable(docTokens,seed2);
     Thread thread1 = new Thread(runnable1);
     Thread thread2 = new Thread(runnable2);
     thread1.start();
     thread2.start();
     thread1.join();
     thread2.join();
     LatentDirichletAllocation lda0 = runnable1.mLda;
     LatentDirichletAllocation lda1 = runnable2.mLda;

     double[] scores = similarity(lda0,lda1);
```

The real work's in the runnable and in the similarity method.

The runnable that executes in each thread is defined to simply generate 199 Gibbs samples, then set the 200th as the value of the member variable `mLda`.

```
static class LdaRunnable implements Runnable {

     LatentDirichletAllocation mLda;
     final int[][] mDocTokens;
     final Random mRandom;

     LdaRunnable(int[][] docTokens, long seed) {
         mDocTokens = docTokens;
         mRandom = new Random(seed);
     }

     public void run() {
         short numTopics = 50;
```

```
        double topicPrior = 0.1;
        double wordPrior = 0.001;
        int numSamples = 200;
        Iterator<GibbsSample> it
            = LatentDirichletAllocation
            .gibbsSample(mDocTokens,numTopics,topicPrior,
                         wordPrior,mRandom);
        for (int i = 1; i < numSamples; ++i)
            it.next();
        mLda = it.next().lda();
    }
}
```

We save the doc-token matrix and the random generated from the seed. The `run()` method uses the other Gibbs sampling method in LDA, which returns an iterator over samples. It's simpler to configure because there's nothing about number of epochs or reporting with callbacks. We then generate number of samples minus one samples in a for loop, then take another sample and assign it to `mLda` for later use.

The similarity method which computes the greedy alignment between topics in the two LDA instances is as follows.

```
static double[] similarity(LatentDirichletAllocation lda0,
                           LatentDirichletAllocation lda1) {
    int numTopics = lda0.numTopics();
    List<TopicSim> pairs = new ArrayList<TopicSim>();
    for (int i = 0; i < numTopics; ++i) {
        double[] pi = lda0.wordProbabilities(i);
        for (int j = 0; j < numTopics; ++j) {
            double[] pj = lda1.wordProbabilities(j);
            double divergence
                = Statistics.symmetrizedKlDivergence(pi,pj);
            TopicSim ts = new TopicSim(i,j,divergence);
            pairs.add(ts);
        }
    }
    Collections.sort(pairs,ScoredObject.comparator());
    boolean[] taken0 = new boolean[numTopics];
    boolean[] taken1 = new boolean[numTopics];
    double[] scores = new double[numTopics];
    int scorePos = 0;
    for (TopicSim ts : pairs) {
        if (!taken0[ts.mI] && !taken1[ts.mJ]) {
            taken0[ts.mI] = taken1[ts.mJ] = true;
            scores[scorePos++] = ts.score();
        }
    }
    return scores;
}
```

It starts by walking over the pairs of topics, extracting the distribution for a specified topic using the method `wordProbabilities(int)`, which is defined on the LDA object. We compute the divergence using the symmetrized KL divergence in LingPipe's `Statistics` utility class.

We keep track of the pairs of topics and their scores using a nested static class, `TopicSim`, defined in the same file.

```
static class TopicSim implements Scored {

    final int mI, mJ;
    final double mScore;

    public TopicSim(int i, int j, double score) {
        mI = i;
        mJ = j;
        mScore = score;
    }

    public double score() {
        return mScore;
    }
}
```

The constructor stores the indices of the topics and the divergence between their topics. The class is defined to implement LingPipe's `Scored` interface to allow easy sorting.

Once we have the set of pairs, we sort it using Java's static `sort()` utility method in the utility class `Collections`. This ensures the pairs are sorted in order of increasing divergence.

Next, we create boolean arrays to keep track of which topics have already been assigned. All values start with default `false` values, then they are set to `true` when the topic with the index is used. This makes sure we don't double-assign topics in our alignment across samples. We then just walk over the pairs in increasing order of divergence, and if both their topics are not already assigned, we assign the topics to each other, set the score, and set the fact that we've assigned. Then, we just return the result.

14.10.3 Running the Demo

The demo is hard coded for everything but the path to the compressed data, which is specified as in the simple LDA example. The Ant target `lda-topic-sim` invokes the demo, specifying the path to the compressed corpus as the value of the property `wormbase.corpus.gz`.

```
> ant -Dwormbase.corpus.gz="../../data-dist/2007-12-01-wormbase-literature.endn
lda-topic-sim

 0   3.0     14   5.1     29   7.2     44  11.9
 1   3.2     15   5.2     31   7.3     45  12.0
 2   3.5     16   5.3     32   7.5     46  12.6
 3   3.8     17   5.6     33   7.6     47  13.1
 4   4.0     18   5.7     34   7.9     48  14.5
 5   4.2     19   6.1     35   8.0     49  15.7
 6   4.4     20   6.2     36   8.1
 7   4.4     21   6.3     37   8.3
 8   4.5     22   6.4     38   8.9
 9   4.7     23   6.5     39   9.4
10   4.8     24   6.8     40   9.9
11   5.0     25   6.9     41   9.9
12   5.0     26   7.0     42  10.3
13   5.0     27   7.1     43  11.6
```

The most closely related topics from the two samples drawn from different chains have a KL-divergence of roughly 3 bits, down to almost 16 bits for the last pair of topics. You can also try running for more epochs or with different random seeds, though these are hard-coded into the demo. As the estimates converge to their stationary states, you'll see the top topics coming closer together and the bottom topics moving further apart in this greedy alignment.

If you really care about comparing topics, this program is easily modified to print out the identifiers of the topics linked this way. By printing the topics as we did before, you can compare them by eye.

14.11 The LDA Model

LDA is a kind of probabilistic model known as a generative model. Generative models provide step-by-step characterizations of how to generate a corpus of documents.[7] This setup seems strange to many people at first because we are in practice presenting LDA with a collection of documents and asking it to infer the topic model, not the other way around. Nevertheless, the probabilistic model is generative.

LDA generates each document independently based on the model parameters. To generate a document, LDA first generates a topic distribution for that document.[8] This happens before any of the words in a document are generated. At this stage, we might only know that a document is 70% about politics and 30% about the economy, not what words it contains.

After generating the topic distribution for a document, we generate the words for the document.

In the case of LDA, we suppose that the number of documents and the number of tokens in each document is given. This means they're not part of

That means we don't try to predict how long each document is or use document-length information for relating documents. Furthermore, LDA does

[7]LDA does not model the number of documents in a corpus or the number of tokens in each document. These values must be provided as constants. This is usually not a problem because the corpus is given as data.

[8]The topic distribution for a document is generated from a Dirichlet distribution, which is where the model gets its name.

not We also do not attempt to model the size of the corpus.

14.11.1 Generalized “Documents”

Although LDA is framed in terms of documents and words, it turns out it only uses the identities of the tokens. As such, it may be applied to collections of “documents” consisting of any kind of count data, not just bags of words. For instance, LDA may be applied to RNA expression data.

Chapter 15

Singular Value Decomposition

Chapter 16

Sentence Boundary Detection

Appendix A

Mathematics

This appendix provides a refresher in some of the mathematics underlying machine learning and natural language processing.

A.1 Basic Notation

A.1.1 Summation

We use summation notation throughout, writing $\sum_{n \in N} x_n$ for the sum of the x_n values for all $n \in N$. For instance, if $N = \{1, 5, 7\}$, then $\sum_{n \in N} x_n = x_1 + x_5 + x_7$.

We also write $\sum_{n=a}^{b} x_n$ for the sum $x_a + x_{a+1}, \ldots, x_{b-1}, x_b$. If $a = -\infty$ or $b = \infty$, the sums are open ended. For instance, we write $\sum_{n=a}^{\infty} x_n$ for the infinite sum $x_a + x_{a+1} + x_{a+2} + \cdots$.

The boundary condition of $\sum_{n \in N} x_n$ when N is the empty set is 0, because 0 is the additive unit (i.e., $0 + n = n + 0 = n$).

A.1.2 Multiplication

We use product notation just like summation notation, swapping the product symbol $\prod$ for the summation symbol $\sum$. For instance, we write $\prod_{n \in N} x_n$ for the product of all the x_n values for $n \in N$. We use bounded and infinite products in exactly the same way as for summation.

The boundary condition for $\prod_{n \in N} x_n$ in the case where N is empty is 1, because 1 is the multiplicative unit (i.e., $1 \times n = n \times 1 = n$).

A.2 Useful Functions

A.2.1 Exponents

The exponential function $\exp(x)$ is defined as the unique non-zero solution to the differential equation $f' = f$. Thus we have

$$\frac{d}{dx} \exp(x) = \exp(x) \quad \text{and} \quad \int \exp(x)\, dx = \exp(x). \tag{A.1}$$

It turns out the solution is $\exp(x) = e^x$, where e, known as Euler's number, is approximately 2.718.

The exponential function may be applied to any real number, but its value is always positive. The exponential function has some convenient properties, such as

$$\exp(a + b) = \exp(a) \times \exp(b), \text{ and} \tag{A.2}$$

$$\exp(a \times b) = \exp(a)^b. \tag{A.3}$$

A.2.2 Logarithms

The natural logarithm, $\log x$ is defined as the inverse of $\exp(x)$. That is, $\log x$ is defined to be the y such that $\exp(y) = x$. Thus for all $x \in (-\infty, \infty)$ and all $y \in (0, \infty)$ we have

$$\log \exp(x) = x \quad \text{and} \quad \exp(\log y) = y. \tag{A.4}$$

Logarithms convert multiplication into addition and exponentiation into multiplication, with

$$\log(x \times y) = \log x + \log y, \text{ and} \tag{A.5}$$

$$\log x^y = y \log x. \tag{A.6}$$

We can define logarithms in different bases, where $\log_b x$ is defined to be the y such that $b^y = x$. In terms of natural logs, we have

$$\log_b x = \log x / \log b. \tag{A.7}$$

A.2.3 The Factorial Function

The factorial function computes the number of different ordered permutations there are of a list of a given (non-negative) number of distinct objects. The factorial function is written $n!$ and defined for $n \in \mathbb{N}$ by

$$n! = 1 \times 2 \times \cdots \times n = \prod_{m=1}^{n} m. \tag{A.8}$$

The convention is to set $0! = 1$ so that the boundary conditions of definitions like the one we provided for the binomial coefficient work out properly.

A.2.4 Binomial Coefficients

The binomial coefficient has many uses. We're mainly interested in combinatorics, so we think of it as providing the number of ways to select a subset of a given size from a superset of a given size. The binomial coefficient is written $mchoosen$, prounced "m choose n," and defined for $m, n \in \mathbb{N}$ with $n \geq m$ by

$$\binom{n}{m} = \frac{n!}{m!\,(n-m)!}. \tag{A.9}$$

A.2.5 Multinomial Coefficients

The multinomial coefficient generalizes the binomial coefficient to choosing more than one item. It tells us how many ways there are to partition a set into subsets of fixed sizes. For a set of size n, the number of ways partition it into subsets of sizes $m_1, \ldots, m_K$, where $m_k \in \mathbb{N}$ and $n = \sum_{k=1}^{K} m_k$, is

$$\binom{n}{m_1, \ldots, m_K} = \frac{n!}{\prod_{k=1}^{K} m_k!}. \tag{A.10}$$

A.2.6 The Γ Function

The Γ function is the complex generalization of the factorial function (see Section A.2.3). It is written $\Gamma(x)$, and we're only concerned with real $x \in [0, \infty)$, where its value is defined by

$$\Gamma(x) = \int_0^\infty t^{x-1} \exp(-t)\, dt. \tag{A.11}$$

In general, we have

$$\Gamma(x+1) = x\, \Gamma(x). \tag{A.12}$$

Because we have $\Gamma(1) = 1$, we also have for all $n \in \mathbb{N}, n > 0$,

$$\Gamma(n) = (n-1)!. \tag{A.13}$$

Appendix B

Statistics

This appendix provides a refresher in some of the statistics underlying machine learning and natural language processing.

B.1 Discrete Probability Distributions

B.1.1 Bernoulli Distribution

The Bernoulli distribution has binary outcomes 0 and 1, with outcome 1 conventionally referred to as "success." The distribution has a single parameter $\theta \in [0, 1]$ for the chance of success. If a random variable X has a Bernoulli distribution with parameter θ we write $X \sim \mathsf{Bern}(\theta)$. The resulting probablity distribution for $x \sim \mathsf{Bern}(\theta)$ is defined for $x \in \{0, 1\}$ by

$$p_X(x) = \theta^x \times (1 - \theta)^{1-x}. \tag{B.1}$$

Working out the cases, if $x = 1$, we have $p_X(x) = \theta$ and if $x = 0$, we have $p_X(x) = 1 - \theta$.

The variance and standard deviation for $X \sim \mathsf{Bern}(\theta)$ are given by

$$\mathrm{var}[X] = \theta(1 - \theta) \quad \text{and} \quad \mathrm{sd}[X] = \sqrt{\theta(1 - \theta)}. \tag{B.2}$$

B.1.2 Binomial Distribution

The binomial distribution is parameterized by a chance-of-success parameter $\theta \in [0, 1]$ and a number-of-trials parameter $N \in \mathbb{N}$. If a random variable X has a binomial distribution with parameters θ and N, we write $X \sim \mathsf{Binom}(\theta, N)$. The resulting probability distribution is defined for $n \in 0{:}N$ by

$$p_X(n) = \binom{N}{n} \theta^n \; (1 - \theta)^{N-n}. \tag{B.3}$$

The variance and standard deviation of a binomially-distributed random variable $X \sim \mathsf{Binom}(\theta, N)$ are given by

$$\mathrm{var}[X] = N\theta(1 - \theta) \quad \text{and} \quad \mathrm{sd}[X] = \sqrt{N\theta(1 - \theta)}. \tag{B.4}$$

We are often more interested in the percentage of the N trials that resulted in success, which is given by the random variable X/N (note that N is a constant here). The variable X/N is scaled to be between 0 and 1. With large N, X/N is expected to be close to to θ. The variance and standard deviation of X/N are derived in the usual way, by dividing by the constant N,

$$\mathrm{var}[X/N] = \frac{1}{N^2}\mathrm{var}[X] = \frac{\theta(1-\theta)}{N}, \qquad \text{and} \tag{B.5}$$

$$\mathrm{sd}[X/N] = \sqrt{\frac{\theta(1-\theta)}{N}}. \tag{B.6}$$

B.1.3 Discrete Distribution

The discrete distribution generalizes the Bernoulli distribution to K outcomes. A discrete distribution is parameterized by a vector $\theta \in [0,1]^K$ such that

$$\sum_{k=1}^{K} \theta_k = 1. \tag{B.7}$$

If the variable X has a discrete distribution with parameter θ, we write $X \sim \mathsf{Discr}(\theta)$. The resulting probabilty distribution for $X \sim \mathsf{Discr}(\theta)$ is defined for $x \in 1{:}K$ by

$$p_X(x) = \theta_x. \tag{B.8}$$

If $K = 2$, a discrete distribution has the same distribution over outcomes 1 and 2 as a Bernoulli has over outcomes 0 and 1. In symbols, for $x \in \{0,1\}$, we have

$$\mathsf{Bern}(x|\theta) = \mathsf{Discr}(x+1|\langle 1-\theta, \theta\rangle) \tag{B.9}$$

Note that if we collapse categories together so that we are left with two outcomes (either one versus all, or one subset versus another subset), we are left with a Bernoulli distribution.

B.1.4 Multinomial Distribution

Multinomials generalize discrete distributions to counts in the same way that binomials generalize Bernoulli distributions. Given a K-dimensional discrete parameter θ and a number of trials N, the multinomial distribution assigns probabilities to K-dimesnional vectors of counts (non-negative integers in $\mathbb{N}$) that sum to N. If a vector random variable X is distributed as a multinomial with discrete parameter θ and number of trials N, we write $X \sim \mathsf{Multinom}(\theta, N,)$. For a variable $X \sim \mathsf{Multinom}(\theta, N,)$, the probability assigned to outcome $x \in \mathbb{N}^K$ where $\sum_{k=1}^{K} x_k = N$, is

$$p_X(x) = \binom{N}{x_1, \ldots, x_K} \prod_{k=1}^{K} \theta_k^{x_k}. \tag{B.10}$$

where we the normalizing term is a multinomial coefficient as defined in Section A.2.5.

B.2 Continuous Probability Distributions

B.2.1 Normal Distribution

B.3 Maximum Likelihood Estimation

B.4 Maximum a Posterior Estimation

B.4.1 χ^2 Distribution

The χ^2 distribution with n degrees of freedom arises as the sum of the squares of n independent unit normal distributions (see Section B.2.1). In symbols, if $X_n \sim \mathsf{Norm}((,0),1)$ for $n \in 1{:}N$, then

$$Y = \sum_{n=1}^{N} X_n^2 \tag{B.11}$$

has a χ^2 distribution with N degrees of freedom.

If a variable X has a χ^2 distribution with N degrees of freedom, we write $X \sim \mathsf{ChiSq}(N)$. The probability distribution for X is defined for $x \in [0, \infty)$ by

$$p_X(x) = \frac{1}{2^{N/2}\,\Gamma(n/2)}\; x^{N/2-1}\; \exp(-x/2). \tag{B.12}$$

If $X \sim \mathsf{ChiSq}(N)$, then the mean, variance and standard deviation of X are

$$\mathbb{E}[X] = N, \quad \operatorname{var}[X] = 2n, \text{ and } \quad \operatorname{sd}[X] = \sqrt{2n}. \tag{B.13}$$

B.5 Information Theory

Information theory was originally developed by Claude Shannon to model the transmission of messages over noisy channels such as telephones or telegraphs.[1]

B.5.1 Entropy

Entropy is an information-theoretic measure of the randomness of a distribution.

The entropy of a random variable X is most easily expressed as the expectation of the negative log probability of the variable,

$$\mathrm{H}[X] = \mathbb{E}[-\log_2 p_X(X)]. \tag{B.14}$$

Because we use base 2, the results are in units of binary digits, otherwise known as bits.

For a discrete random variable X, unfolding the expectation notation yields

$$\mathrm{H}[X] = -\sum_{n\in\mathbb{N}} p_X(n)\; \log_2 p_X(n). \tag{B.15}$$

[1]Claude Shannon. 1948. A mathematical theory of communication. *Bell System Technical Journal* **27**:379–423 and **27**:623–656.

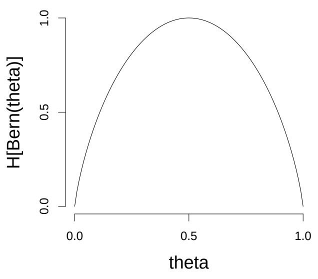

Fig. B.1: *Entropy of a random variable distributed as* $\text{Bern}(\theta)$.

For a continous random variable X, we replace the summations over natural numbers with integration over the real numbers,

$$\text{H}[X] = -\int_{-\infty}^{\infty} p_X(x)\ \log_2 p_X(x)\ dx \tag{B.16}$$

For example, if we have a Bernoulli-distributed random variable $Y \sim \text{Bern}(\theta)$, its entropy is

$$\begin{aligned}\text{H}[Y] &= -p_Y(1)\log_2 p_Y(1) - p_Y(0)\log_2 p_Y(0) && \text{(B.17)}\\ &= -\theta\log_2\theta - (1-\theta)\log_2(1-\theta). && \text{(B.18)}\end{aligned}$$

A plot of $\text{H}[Y]$ for $Y \sim \text{Bern}(\theta)$ is provided in Figure B.1. The graph makes it clear that when $\theta = 0$ or $\theta = 1$, the outcome is certain, and therefore the entropy is 0. The maximum entropy possible for a Bernoulli distribution is 1 bit, achieved at $\theta = 0.5$.

Entropy and Compression

Entropy has a natural interpretation in terms of compression. Suppose we want to send the a message X which might take on values in $\mathbb{N}$.[2] If the sender and receiver both know the distribution p_X of outcomes, the expected cost to send the message is $\text{H}[X]$ bits. For instance, if we need to send a bit with value 0 or 1, and each outcome is equally likely, it will require 1 bit to send the message. On the other hand, if one outcome is more likely than the other, we can save space (if we have repeated messages to send; otherwise, we must round up).

B.5.2 Joint Entropy

We can measure the entropy of two variables X and Y by measuring the entropy of their joint distribution $p_{X,Y}$, generalizing our definition to

$$\text{H}[X, Y] = \mathbb{E}[\log_2 p_{X,Y}(x, y)]. \tag{B.19}$$

[2]We can always code sequences of characters as numbers, as originally noted by Kurt Gödel, and as explained in any introductory computer science theory text. Just think of a string as representing a number in the base of the number characters.

For instance, with discrete X and Y, this works out to

$$\mathrm{H}[X, Y] = -\sum_x \sum_y p_{X,Y}(x, y)\ \log_2 p_{X,Y}(x, y). \tag{B.20}$$

Joint entropy is symmetric, so that

$$\mathrm{H}[X, Y] = \mathrm{H}[Y, X]. \tag{B.21}$$

B.5.3 Conditional Entropy

We can measure the entropy of a variable X conditional on knowing the value of another variable Y. The expectation-based definition is thus

$$\mathrm{H}[X|Y] = \mathbb{E}[-\log_2 p_{X|Y}(X|Y)]. \tag{B.22}$$

In the case of discrete X and Y, this works out to

$$\begin{aligned}\mathrm{H}[X|Y] &= -\sum_y \sum_x p_{X,Y}(x, y)\ \log_2 p_{X|Y}(x|y) & \text{(B.23)}\\ &= -\sum_y p_Y(y) \sum_x p_{X|Y}(x|y)\ \log_2 p_{X|Y}(x|y). & \text{(B.24)}\end{aligned}$$

Conditional entropy, joint entropy and entropy are related as for probabilty distributions, with

$$\mathrm{H}[X, Y] = \mathrm{H}[X|Y] + \mathrm{H}[Y]. \tag{B.25}$$

B.5.4 Mutual Information

The mutual information between a pair of variables X and Y measures how much more information there is in knowing their joint distribution than their individual distributions for prediction. Using expectations, mutual information between variables X and Y is defined to be

$$\begin{aligned}\mathrm{I}[X; Y] &= \mathbb{E}[-\log_2 \frac{p_{X,Y}(x, y)}{p_X(x)\ p_Y(y)}] & \text{(B.26)}\\ &= \mathbb{E}[-\log_2 p_{X,Y}(x, y)] + \mathbb{E}[\log_2 p_X(x)] + \mathbb{E}[\log_2 p_Y(y)] & \text{(B.27)}\end{aligned}$$

Another way of looking at mutual information is in terms of the ratio of a conditional and marginal distribution. If we unfold the joint distribution $p_{X,Y}(x, y)$ into the product of the marginal $p_X(x)$ and conditional $p_Y(y|x)$, we get

$$\mathrm{I}[X; Y] = \mathbb{E}[-\log_2 \frac{p_{X|Y}(x|y)}{p_X(x)}] = \mathbb{E}[-\log_2 \frac{p_{Y|X}(y|x))}{p_Y(y)}]. \tag{B.28}$$

In the discrete case, this unfolds to

$$\mathrm{I}[X; Y] = -\sum_{x,y} p_{X,Y}(x, y)\ \log_2 \frac{p_{X,Y}(x, y)}{p_X(x)\ p_Y(y)}. \tag{B.29}$$

Mutual information is symmetric, so that

$$\mathrm{I}[X; Y] = \mathrm{I}[X; Y]. \tag{B.30}$$

It's related to conditional and joint entropies by the unfolding in the second step of the definition through

$$\begin{aligned} \mathrm{I}[X;Y] &= \mathrm{H}[X] + \mathrm{H}[Y] - \mathrm{H}[X,Y] && \text{(B.31)} \\ &= \mathrm{H}[X] - \mathrm{H}[X|Y]. && \text{(B.32)} \end{aligned}$$

B.5.5 Cross Entropy

Cross-entropy measures the number of bits required, on average, to transmit a value of X using the distribution of Y for coding. Using expectation notation, the definition reduces to

$$\mathrm{CH}[X \parallel Y] = \mathbb{E}[-\log_2 p_Y(X)]. \tag{B.33}$$

In the case of discrete random variables X and Y, this works out to

$$\mathrm{CH}[X \parallel Y] = -\sum_n p_X(n)\ \log_2 p_Y(n). \tag{B.34}$$

B.5.6 Divergence

Kullback-Leibler (KL) divergence is the standard method to compare two distributions. Pairs of distributions that have similar probabilities will have low divergence from each other. Because KL divergence is not symmetric, we also consider two standard symmetric variants.

Following the standard convention, our previous infomration definitions were all in terms of random variables X and Y, even though the definitions only dependeded on their corresponding probability distributions p_X and p_Y. Divergence is conventionally defined directly on distributions, mainly to avoid the case of comparing two different random variables X and Y with the same distribution $p_X = p_Y$.

KL Divergence

Given two discrete distributions p_X and p_Y, the KL divergence of p_Y from p_X is given by

$$\mathrm{D}_{\mathrm{KL}}(p_X \| p_Y) = \sum_{n=1}^{N} p_X(n) \log_2 \frac{p_X(n)}{p_Y(n)}. \tag{B.35}$$

If there is an outcome n where $p_Y(n) = 0$ and $p_X(n) > 0$, the divergence is infinite. We may allow $p_X(n) = 0$ by interpreting $0 \log_2 \frac{0}{p_Y(n)} = 0$ by convention (even if $p_Y(n) > 0$).

If we assume p_X and p_Y arose from random variables X and Y, KL divergence may expressed using expectations as

$$\begin{aligned} \mathrm{D}_{\mathrm{KL}}(p_X \| p_Y) &= \mathbb{E}[\log_2 \frac{p_X(X)}{p_Y(X)}] && \text{(B.36)} \\ &= \mathbb{E}[\log_2 p_X(X)] - \mathbb{E}[\log_2 p_Y(X)] && \text{(B.37)} \\ &= \mathbb{E}[-\log_2 p_Y(X)] - \mathbb{E}[-\log_2 p_X(X)] && \text{(B.38)} \\ &= \mathrm{CH}[X \parallel Y] - \mathrm{H}[X]. && \text{(B.39)} \end{aligned}$$

This definition makes clear that KL divergence may be viewed as as the expected penalty in bits for using p_Y to transmit values drawn from X rather than p_X itself. In other words, KL divergence is just the cross-entropy minus the entropy.

Although we do not provide a proof, we note that $\mathrm{D}_{\mathrm{KL}}(p_X \| p_Y) \geq 0$ for all distributions p_X and p_Y. We further note without proof that $\mathrm{D}_{\mathrm{KL}}(p_X \| p_Y) = 0$ if and only if $p_X = p_Y$.

Symmetrized KL Divergence

KL divergence is not symmetric in the sense that there exist pairs of random variables X and Y such that $\mathrm{D}_{\mathrm{KL}}(p_X \| p_Y) \neq \mathrm{D}_{\mathrm{KL}}(p_Y \| p_X)$.

There are several divergence measures derived from KL divergence that are symmetric. The simplest approach is just to introduce symmetry by brute force. The symmetrized KL-divergence $\mathrm{D}_{\mathrm{SKL}}(p_X \| p_Y)$ between distributions p_X and p_Y is defined by averaging the divergence of p_X from p_Y and the divergence of p_Y from p_X, yielding gives us

$$\mathrm{D}_{\mathrm{SKL}}(p_X \| p_Y) = \frac{1}{2}(\mathrm{D}_{\mathrm{KL}}(p_X \| p_Y) + \mathrm{D}_{\mathrm{KL}}(p_Y \| p_X)). \tag{B.40}$$

Obviously, $\mathrm{D}_{\mathrm{SKL}}(p_X \| p_Y) = \mathrm{D}_{\mathrm{SKL}}(p_Y \| p_X)$ by definition, so the measure is symmetric. As with KL divergence, symmetrized KL divergence so that, in general, $\mathrm{D}_{\mathrm{SKL}}(p_X \| p_Y) \geq 0$ with $\mathrm{D}_{\mathrm{SKL}}(p_X \| p_Y) = 0$ if and only if $p_X = p_Y$.

Jensen-Shannon Divergence

Another widely used symmetric divergence measure derived from KL divergence is the Jensen-Shannon divergence. To compare X and Y using Jensen-Shannon divergence, we first define a new distribution p_Z by averaging the distributions p_X and p_Y, by

$$p_Z(n) = \frac{1}{2}(p_X(n) + p_Y(n)). \tag{B.41}$$

If X and Y are random variables and Z is defined to be $(X + Y)/2$, then p_X, p_Y and p_Z satisfy the above definition.

Given the average distribution p_Z, we can define Jensen-Shannon divergence by taking average divergence of p_X and p_Y from p_Z as defined in Equation B.41, setting

$$\mathrm{D}_{\mathrm{JS}}(p_X \| p_Y) = \frac{1}{2}(\mathrm{D}_{\mathrm{KL}}(p_X \| p_Z) + \mathrm{D}_{\mathrm{KL}}(p_Y \| p_Z)). \tag{B.42}$$

As with the symmetrized KL divergence, Jensen-Shannon divergence is symmetric by definition.

LingPipe KL-Divergence Utilities

KL-divergence is implemented as a static utility method in the `Statistics` utility class in package `com.aliasi.stats`. The method takes two double arrays representing probability distributions and measures how much the first is like the second.

In order to give you a sense of KL-divergence, we implement a simple utility in the demo class `KlDivergence` to let you try various values. The code just parses out double arrays from teh command line and sends them to the KL function.

```
public static void main(String[] args) {
     double[] pX = parseDoubles(args[0],"pX");
     double[] pY = parseDoubles(args[1],"pY");
     double kld = Statistics.klDivergence(pX,pY);
     double skld = Statistics.symmetrizedKlDivergence(pX,pY);
     double jsd = Statistics.jsDivergence(pX,pY);
```

The ant target `kl` runs the command with the first two arguments given by properties `p.x` and `p.y`. To calculate the example from above, we have

```
> ant -Dp.x="0.2 0.8" -Dp.y="0.4 0.6" kl
```

```
pX=(0.2, 0.8)     pY=(0.4, 0.6)
kld=0.13202999942307514
skld=0.1415037499278844
jsd=0.03485155455967723
```

The Jensen-Shannon divergence is less than the symmetrized KL divergence because the interpolated distribution p_Z is closer to both of the original distributions p_X and p_Y than they are to each other.

Appendix C

Java Basics

In this appendix, we go over some of the basics of Java that are particularly relevant for text and numerical processing. We also refer to the reader to the first chapter of the companion volume, *Text Processing with Java*. It contains introduction to Java primitive types and their bit-level representations, arrays, objects, and synchronization.

C.1 Generating Random Numbers

Java has a built-in pseudorandom number generator. It is implemented in the class `Random` in package `java.util`.

C.1.1 Generating Random Numbers

Given an instance of `Random` (see the next section for construction details), we are able to use it to generate random outcomes of several primitive types.

Discrete Random Numbers

The method `nextBoolean()` returns a random `booleann` value, `nextInt()` a random `int`, and `nextLong()` a random long. The algorithms generating these values are designed to assign equal probabilities to all outcomes.

The method `nextInt(int)` is particularly useful in that it generates a value between 0 (inclusive) and its argument (exclusive).

Byte are generated using the C-like method `nextBytes(byte[])`, which fills the specified byte array with random bytes.

Floating Point Numbers

As with all things floating point, random number generation is tricky. The method `nextFloat()` and `nextDouble()` return values beteween 0.0 (inclusive) and 1.0 (exclusive). This is, in part, to compensate for the uneven distribution of values across the bit-level representations. When values are large, there are larger gaps between possible floating point representations. The algorithms used are not quite generating random bit-level representations; they are documented in the class's Javadoc.

There is also a method `nextGaussian()` that generates a draw from a unit normal, $\mathsf{Norm}(0,1)$, distribution.

C.1.2 Seeds and Pseudorandomness

A random number generator is constructed using a base costructor `Random(long)`. The argument is called the seed of the random number generator.

The behavior of `Random` is fully determined by the seed. A generator with deterministic behavior like this is said to be *pseudorandom*. By setting the seed, we are able to generate the same "random" results whenever we run the algorithm. This is immensely helpful for debugging purposes.

If you want true randomization, just use the zero-argument constructor `Random()`. This constructor combines the time obtained from Java's `System.nanoTime()`) and a changing internal key in an attempt to generate unique seeds each time it is called.

We sometimes use the following pattern to enable us a fresh random seed for each run, but still enable reproducible results.

```
Random seeder = new Random();
long seed = seeder.nextLong();
System.out.printf("Using seed=" + seed);
Random random = new Random(seed);
```

By printing out the seed, we can reproduce any results from a run. This is especially useful for random algorithms (that is, algorithms that makes choices probabilistically, such as Gibbs samplers or K-means clusterers).

LingPipe is set up so that randomized algorithms take an explicit `Random` argument, so that results may be reproduced.

C.1.3 Alternative Randomization Implementations

Because the `Random` class is not final, it is possible to plug in different implementations. In fact, `Random` was designed for extension by implementing most of its work through the protected `next(int)` method. This method is used to define the other random number generators, sometimes being called more than once in a method.

C.1.4 Synchronization

Although not externally documented, there is a great deal of careful synchronization underlying the methods in `Random`. As of Java 1.6, the various `next`*X*`()` methods (other than `nextGaussian()`) use a scalable compare-and-set (CAS) strategy through Java's built-in `AtomicLong` class. The `nextGaussian()` method uses its own synchronization.

This synchronization happens in the `next(int)` method, so classes that extend `Random` need to be designed with thread safety in mind.

Even with its design for liveness, the Javadoc itself recommends using a separate random number generator in each thread if possible to reduce even the minimal lock contention imposed by the atomic long's compare-and-set operation.

C.1.5 The `Math.random()` Utility

The `Math` utility class in the built-in `java.lang` package provides a simple means of generating double values. The utility method `Math.random()` that returns the result of calling `nextDouble()` on a random number generator, thus returning a value between 0.0 (inclusive) and 1.0 (exclusive). The random number generator is lazily initialized thread safely so that calls to the method are thread safe.

Appendix D

Corpora

In this appendix, we list the corpora that we use for examples in the text.

D.1 Canterbury Corpus

The Canterbury corpus is a benchmark collection of mixed text and binary files for evaluating text compression.

Contents

The files contained in the archive are the following.

File	*Type*	*Description*	*Bytes*
`alice29.txt`	text	English text	152,089
`asyoulik.txt`	play	Shakespeare	125,179
`cp.html`	html	HTML source	24,603
`fields.c`	Csrc	C source	11,150
`grammar.lsp`	list	LISP source	3,721
`kennedy.xls`	Excl	Excel Spreadsheet	1,029,744
`lcet10.txt`	tech	Technical writing	426,754
`plrabn12.txt`	poem	Poetry	481,861
`ptt5`	fax	CCITT test set	513,216
`sum`	SPRC	SPARC Executable	38,240
`xargs.1`	man	GNU manual page	4,227

Authors

Ross Arnold and Tim Bell.

Licesning

The texts are all public domain.

Download

`http://corpus.canterbury.ac.nz/descriptions/`

Warning: you should create a directory in which to unpack the distribution because the tarball does not contain any directory structure, just a bunch of flat files.

D.2 20 Newsgroups

Contents

The corpus contains roughly 20,000 posts to 20 different newsgroups. These are

`comp.graphics`	`rec.autos`	`sci.crypt`
`comp.os.ms-windows.misc`	`rec.motorcycles`	`sci.electronics`
`comp.sys.ibm.pc.hardware`	`rec.sport.baseball`	`sci.med`
`comp.sys.mac.hardware`	`rec.sport.hockey`	`sci.space`
`comp.windows.x`		
`misc.forsale`	`talk.politics.misc`	`talk.religion.misc`
	`talk.politics.guns`	`alt.atheism`
	`talk.politics.mideast`	`soc.religion.christian`

Authors

It is currently being maintained by Jason Rennie, who speculates that Ken Lang may be the original curator.

Licensing

There are no licensing terms listed and the data may be downloaded directly.

Download

`http://people.csail.mit.edu/jrennie/20Newsgroups/`

D.3 MedTag

Contents

The MedTag corpus has two components, MedPost, a part-of-speech-tagged corpus, and GENETAG, a gene/protein mention tagged corpus. The data annotated consists of sentences, and sentence fragments from MEDLINE citation titles and abstracts.

Authors

The MedPost corpus was developed at the U. S. National Center for Biotechnology Information (NCBI), a part of the National Library of Medicine (NLM), which is itself one of the National Institutes of Health (NIH). Its contents are described in

Smith, L. H., L. Tanabe, T. Rindflesch, and W. J. Wilbur. 2005. MedTag: a collection of biomedical annotations. In *Proceedings of the ACL-ISMB Workshop on Linking Biological Literature, Ontologies and Databases: Mining Biological Semantics*. 32–37.

Licensing

The MedPost corpus is a public domain corpus released as a "United States Government Work."

Download

`ftp://ftp.ncbi.nlm.nih.gov/pub/lsmith/MedTag/medtag.tar.gz`

D.4 WormBase MEDLINE Citations

The WormBase web site, `http://wormbase.org`, curates and distributes resources related to the model organism *Caenorhabditis elegans*, the nematode worm. Part of that distribution is a set of citation to published research on *C. elegans*. These literature distributions consist of MEDLINE citations in a simple line-oriented format.

Authors

Wormbase is a collaboration among many cites with many supporters. It is described in

Harris, T. W. et al. 2010. WormBase: A comprehensive resource for nematode research. *Nucleic Acids Research* **38**. `doi:10.1093/nar/gkp952`

Download

`ftp://ftp.wormbase.org/pub/wormbase/misc/literature/`
`2007-12-01-wormbase-literature.endnote.gz` (15MB)

Appendix E

Further Reading

In order to fully appreciate contemporary approaches to natural language processing requires three fairly broad areas of expertise: linguistics, statistics, and algorithms. We don't pretend to introduce these areas in any depth in this book. Instead, we recommend the following textbooks, sorted by area, from among the many that are currently available.

See the companion volume, *Text Processing in Java 6*, for more background on Unicode, Java and its libraries, and general programming.

E.1 Algorithms

Because natural language processing requires comptuational implementations of its models, a good knowledge of algorithm analysis goes a long way in understanding the approaches found in a tool kit like LingPipe.

- Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduciton to Algorithms*, Third Edition. MIT Press.

  A thorough yet readable introduction to algorithms.

- Durbin, Richard, Sean R. Eddy, Anders Krogh, Grame Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids.* Cambridge University Press.

  Even though it focuses on biology, it has great descriptions of several key natural language processing algorithms such as hidden Markov models and probabilistic context-free grammars.

- Gusfield, Dan. 1997. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press.

  Incredibly detailed and thorough introduction to string algorithms, mainly aimed at computational biology, but useful for all string and tree processing.

E.2 Probability and Statistics

Most of the tools in LingPipe are statistical in nature. To fully understand modern natural language processing it's necessary to understand the statistical models which make up their foundation. To do this requires an understanding of the basics of statistics. Much of the work in natural language processing is Bayesian, so understanding Bayesian statistics is now a must for understanding the field. Much of it also relies on information theory, which is also introduced in the machine learning textbooks.

- Bulmer, M. G. 1979. *Principles of Statistics*. Dover.

  Packs an amazing amount of information into a readable little volume, though it assumes some mathematical sophistication.

- Cover, Thomas M. and Joy A. Thomas. 2006. *Elements of Information Theory*, Second Edition. Wiley.

  This is the classic textbook for information theory.

- DeGroot, Morris H. and Mark J. Schervish. 2002. *Probability and Statistics*, Third Edition. Addison-Wesley.

  Another introduction to classical frequentist statistical hypothesis testing and estimation but with some Bayesian concepts thrown into the mix.

- Gelman, Andrew and Jennifer Hill. 2006. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

  A great practical introduction to regression from a Bayesian perspective with an emphasis on multilevel models. Much less mathematically demanding than the other introductions, but still very rigorous.

- Gelman, Andrew, John B. Carlin, Hal S. Stern, and Donald B. Rubin. 2003. *Bayesian Data Analysis*, Second Edition. Chapman and Hall.

  The bible for Bayesian methods. Presupposes high mathematical sophistication and a background in classical probability and statistics to the level of DeGroot and Schervish or Larsen and Marx's books.

- Larsen, Richard J. and Morris L. Marx. 2005. *Introduction to Mathematical Statistics and Its Applications*, Fourth Edition. Prentice-Hall.

  A nice intro to classical frequentist statistics.

- Agresti, A. 2002. *Categorical Data Analysis*, Second Edition. Wiley.

  A thorough survey of classical approaches to analyzing categorical data, focusing on contingency tables.

E.3 Machine Learning

Machine learning, in contrast to statistics, focuses on implementation details and tends to concentrate on predictive inference for larger scale applications. As such, understanding machine learning requires a background in both statistics and algorithms.

- Bishop, Christopher M. 2007. *Pattern Recognition and Machine Learning*. Springer.

  Introduction to general machine learning techniques that is self contained, but assumes more mathematical sophistication than this book.

- Hastie, Trevor, Robert Tibshirani and Jerome Friedman. 2009. *The Elements of Statistical Learning*, Second Edition. Springer.

  Presupposes a great deal of mathematics, providing fairly rigorous introductions to a wide range of algorithms and statistical models.

- MacKay, David J. C. 2002. *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press.

  More of a set of insightful case studies than a coherent textbook. Presupposes a high degree of mathematical sophistication.

- Witten, Ian H. and Eibe Frank. 2005. *Data Mining: Practical Machine Learning Tools and Techniques* (Second Edition). Elsevier.

  The most similar to this book, explaining general machine learning techniques at a high level and linking to the Java implementations (from their widely used Weka toolkit).

E.4 Linguistics

Natural language processing is about language. To fully appreciate the models in LingPipe, it is necessary to have some background in linguistics. Unfortunately, most of the textbooks out there are based on traditional Chomskyan theoretical linguistics which divorces language from its processing, and from which natural language processing diverged several decades ago.

- Bergmann, Anouschka, Kathleen Currie Hall, and Sharon Miriam Ross (editors). 2007. *Language Files*, Tenth Edition. Ohio State University Press.

  Easy to follow and engaging overview of just about every aspect of human language and linguistics. Books on specific areas of linguistics tend to be more theory specific.

E.5 Natural Language Processing

Natural language processing is largely about designing and coding computer programs that operate over natural language data. Because most NLP is statistical these days, this requires not only a knowledge of linguistics, but also of algorithms and statistics.

- Jurafsky, Daniel, James H. Martin. 2008. *Speech and Language Processing*, Second Edition. Prentice-Hall.

  High-level overview of both speech and language processing.

- Manning, Christopher D., Raghavan Prabhakar, and Hinrich Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press.

  Good intro to IR and better intro to some NLP than Manning and Schütze's previous book, which is starting to feel dated.

- Manning, Christopher D. and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing.* MIT Press.

  Needs a rewrite to bring it up to date, but still worth reading given the lack of alternatives.

Appendix F

Licenses

This chapter includes the text of the licenses for all of the software used for demonstration purposes.

F.1 LingPipe License

F.1.1 Commercial Licensing

Customized commercial licenses for LingPipe or components of LingPipe are avaialble through LingPipe, Inc.

F.1.2 Royalty-Free License

Both LingPipe and the code in this book are distributed under the Alias-i Royalty Free License Version 1.

```
Alias-i ROYALTY FREE LICENSE VERSION 1

Copyright (c) 2003-2010 Alias-i, Inc All Rights Reserved

1.  This Alias-i Royalty Free License Version 1 ("License") governs the copying, modifying, and distributing of the computer
program or work containing a notice stating that it is subject to the terms of this License and any derivative works of that
computer program or work.  The computer program or work and any derivative works thereof are the "Software." Your copying,
modifying, or distributing of the Software constitutes acceptance of this License.  Although you are not required to accept
this License, since you have not signed it, nothing else grants you permission to copy, modify, or distribute the Software.  If
you wish to receive a license from Alias-i under different terms than those contained in this License, please contact Alias-i.
Otherwise, if you do not accept this License, any copying, modifying, or distributing of the Software is strictly prohibited by
law.

2.  You may copy or modify the Software or use any output of the Software (i) for internal non-production trial, testing and
evaluation of the Software, or (ii) in connection with any product or service you provide to third parties for free.  Copying or
modifying the Software includes the acts of "installing", "running", "using", "accessing" or "deploying" the Software as those
terms are understood in the software industry.  Therefore, those activities are only permitted under this License in the ways
that copying or modifying are permitted.

3.  You may distribute the Software, provided that you:  (i) distribute the Software only under the terms of this License, no
more, no less; (ii) include a copy of this License along with any such distribution; (iii) include the complete corresponding
machine-readable source code of the Software you are distributing; (iv) do not remove any copyright or other notices from the
Software; and, (v) cause any files of the Software that you modified to carry prominent notices stating that you changed the
Software and the date of any change so that recipients know that they are not receiving the original Software.

4.  Whether you distribute the Software or not, if you distribute any computer program that is not the Software, but that (a)
is distributed in connection with the Software or contains any part of the Software, (b) causes the Software to be copied or
modified (i.e., ran, used, or executed), such as through an API call, or (c) uses any output of the Software, then you must
distribute that other computer program under a license defined as a Free Software License by the Free Software Foundation or an
Approved Open Source License by the Open Source Initiative.

5.  You may not copy, modify, or distribute the Software except as expressly provided under this License, unless you receive a
different written license from Alias-i to do so.  Any attempt otherwise to copy, modify, or distribute the Software is without
```

F.2 Java Licenses

The Java executables comes with two licenses, one for the Java runtime environment (JRE), which is required to execute Java programs, and one for the Java development kit (JDK), required to compile programs. There is a third license for the Java source code.

F.2.1 Sun Binary License

Both the JRE and JDK are subject to Sun's Binary Code License Agreement.

7. TERMINATION. This Agreement is effective until terminated. You may terminate this Agreement at any time by destroying all copies of Software. This Agreement will terminate immediately without notice from Sun if you fail to comply with any provision of this Agreement. Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right. Upon Termination, you must destroy all copies of Software.

8. EXPORT REGULATIONS. All Software and technical data delivered under this Agreement are subject to US export control laws and may be subject to export or import regulations in other countries. You agree to comply strictly with all such laws and regulations and acknowledge that you have the responsibility to obtain such licenses to export, re-export, or import as may be required after delivery to you.

9. TRADEMARKS AND LOGOS. You acknowledge and agree as between you and Sun that Sun owns the SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET trademarks and all SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET-related trademarks, service marks, logos and other brand designations ("Sun Marks"), and you agree to comply with the Sun Trademark and Logo Usage Requirements currently located at http://www.sun.com/policies/trademarks. Any use you make of the Sun Marks inures to Sun's benefit.

10. U.S. GOVERNMENT RESTRICTED RIGHTS. If Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in Software and accompanying documentation will be only as set forth in this Agreement; this is in accordance with 48 CFR 227.7201 through 227.7202-4 (for Department of Defense (DOD) acquisitions) and with 48 CFR 2.101 and 12.212 (for non-DOD acquisitions).

11. GOVERNING LAW. Any action related to this Agreement will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

12. SEVERABILITY. If any provision of this Agreement is held to be unenforceable, this Agreement will remain in effect with the provision omitted, unless omission would frustrate the intent of the parties, in which case this Agreement will immediately terminate.

13. INTEGRATION. This Agreement is the entire agreement between you and Sun relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification of this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

JRE Supplemental Terms

In addition to the binary license, the JRE is additionally subject to the following supplemental license terms.

SUPPLEMENTAL LICENSE TERMS

These Supplemental License Terms add to or modify the terms of the Binary Code License Agreement. Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Binary Code License Agreement. These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Binary Code License Agreement, or in any license contained within the Software.

1. Software Internal Use and Development License Grant. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software "README" file incorporated herein by reference, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce internally and use internally the Software complete and unmodified for the purpose of designing, developing, and testing your Programs.

2. License to Distribute Software. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software README file, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce and distribute the Software (except for the JavaFX Runtime), provided that (i) you distribute the Software complete and unmodified and only bundled as part of, and for the sole purpose of running, your Programs, (ii) the Programs add significant and primary functionality to the Software, (iii) you do not distribute additional software intended to replace any component(s) of the Software, (iv) you do not remove or alter any proprietary legends or notices contained in the Software, (v) you only distribute the Software subject to a license agreement that protects Sun's interests consistent with the terms contained in this Agreement, and (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

3. Java Technology Restrictions. You may not create, modify, or change the behavior of, or authorize your licensees to create, modify, or change the behavior of, classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any naming convention designation.

4. Source Code. Software may contain source code that, unless expressly licensed for other purposes, is provided solely for reference purposes pursuant to the terms of this Agreement. Source code may not be redistributed unless expressly provided for in this Agreement.

5. Third Party Code. Additional copyright notices and license terms applicable to portions of the Software are set forth in the THIRDPARTYLICENSEREADME.txt file. In addition to any terms and conditions of any third party opensource/freeware license identified in the THIRDPARTYLICENSEREADME.txt file, the disclaimer of warranty and limitation of liability provisions in paragraphs 5 and 6 of the Binary Code License Agreement shall apply to all Software in this distribution.

6. Termination for Infringement. Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right.

7. Installation and Auto-Update. The Software's installation and auto-update processes transmit a limited amount of data to Sun (or its service provider) about those specific processes to help Sun understand and optimize them. Sun does not associate the data with personally identifiable information. You can find more information about the data Sun collects at http://java.com/data/.

For inquiries please contact: Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A.

JDK Supplemental License Terms

The JDK is additionally subject to the following supplemental license terms.

F.2.2 Sun Community Source License Version 2.8

The license for the Java source code is

SUN COMMUNITY SOURCE LICENSE Version 2.8 (Rev. Date January 17, 2001)

RECITALS

Original Contributor has developed Specifications and Source Code implementations of certain Technology; and

Original Contributor desires to license the Technology to a large community to facilitate research, innovation andproduct development while maintaining compatibility of such products with the Technology as delivered by Original Contributor; and

Original Contributor desires to license certain Sun Trademarks for the purpose of branding products that are compatible with the relevant Technology delivered by Original Contributor; and

You desire to license the Technology and possibly certain Sun Trademarks from Original Contributor on the terms and conditions specified in this License.

In consideration for the mutual covenants contained herein, You and Original Contributor agree as follows:

AGREEMENT

1. Introduction. The Sun Community Source License and effective attachments ("License") may include five distinct licenses: Research Use, TCK, Internal Deployment Use, Commercial Use and Trademark License. The Research Use license is effective when You execute this License. The TCK and Internal Deployment Use licenses are effective when You execute this License, unless otherwise specified in the TCK and Internal Deployment Use attachments. The Commercial Use and Trademark licenses must be signed by You and Original Contributor in order to become effective. Once effective, these licenses and the associated requirements and responsibilities are cumulative. Capitalized terms used in this License are defined in the Glossary.

2. License Grants.

2.1. Original Contributor Grant. Subject to Your compliance with Sections 3, 8.10 and Attachment A of this License, Original Contributor grants to You a worldwide, royalty-free, non-exclusive license, to the extent of Original Contributor's Intellectual Property Rights covering the Original Code, Upgraded Code and Specifications, to do the following:

a) Research Use License: (i) use, reproduce and modify the Original Code, Upgraded Code and Specifications to create Modifications and Reformatted Specifications for Research Use by You, (ii) publish and display Original Code, Upgraded Code and Specifications with, or as part of Modifications, as permitted under Section 3.1 b) below, (iii) reproduce and distribute copies of Original Code and Upgraded Code to Licensees and students for Research Use by You, (iv) compile, reproduce and distribute Original Code and Upgraded Code in Executable form, and Reformatted Specifications to anyone for Research Use by You.

b) Other than the licenses expressly granted in this License, Original Contributor retains all right, title, and interest in Original Code and Upgraded Code and Specifications. 2.2. Your Grants.

a) To Other Licensees. You hereby grant to each Licensee a license to Your Error Corrections and Shared Modifications, of the same scope and extent as Original Contributor's licenses under Section 2.1 a) above relative to Research Use, Attachment C relative to Internal Deployment Use, and Attachment D relative to Commercial Use.

b) To Original Contributor. You hereby grant to Original Contributor a worldwide, royalty-free, non-exclusive, perpetual and irrevocable license, to the extent of Your Intellectual Property Rights covering Your Error Corrections, Shared Modifications and Reformatted Specifications, to use, reproduce, modify, display and distribute Your Error Corrections, Shared Modifications and Reformatted Specifications, in any form, including the right to sublicense such rights through multiple tiers of distribution.

c) Other than the licenses expressly granted in Sections 2.2 a) and b) above, and the restriction set forth in Section 3.1 d)(iv) below, You retain all right, title, and interest in Your Error Corrections, Shared Modifications and Reformatted Specifications.

2.3. Contributor Modifications. You may use, reproduce, modify, display and distribute Contributor Error Corrections, Shared Modifications and Reformatted Specifications, obtained by You under this License, to the same scope and extent as with Original Code, Upgraded Code and Specifications.

2.4. Subcontracting. You may deliver the Source Code of Covered Code to other Licensees having at least a Research Use license, for the sole purpose of furnishing development services to You in connection with Your rights granted in this License. All such Licensees must execute appropriate documents with respect to such work consistent with the terms of this License, and acknowledging their work-made-for-hire status or assigning exclusive right to the work product and associated Intellectual Property Rights to You.

3. Requirements and Responsibilities.

3.1. Research Use License. As a condition of exercising the rights granted under Section 2.1 a) above, You agree to comply with the following:

a) Your Contribution to the Community. All Error Corrections and Shared Modifications which You create or contribute to are automatically subject to the licenses granted under Section 2.2 above. You are encouraged to license all of Your other Modifications under Section 2.2 as Shared Modifications, but are not required to do so. You agree to notify Original Contributor of any errors in the Specification.

b) Source Code Availability. You agree to provide all Your Error Corrections to Original Contributor as soon as reasonably practicable and, in any event, prior to Internal Deployment Use or Commercial Use, if applicable. Original Contributor may, at its discretion, post Source Code for Your Error Corrections and Shared Modifications on the Community Webserver. You may also post Error Corrections and Shared Modifications on a web-server of Your choice; provided, that You must take reasonable precautions to ensure that only Licensees have access to such Error Corrections and Shared Modifications. Such precautions shall include, without limitation, a password protection scheme limited to Licensees and a click-on, download certification of Licensee status required of those attempting to download from the server. An example of an acceptable certification is attached as Attachment A-2.

c) Notices. All Error Corrections and Shared Modifications You create or contribute to must include a file documenting the additions and changes You made and the date of such additions and changes. You must also include the notice set forth in Attachment A-1 in the file header. If it is not possible to put the notice in a particular Source Code file due to its structure, then You must include the notice in a location (such as a relevant directory file), where a recipient would be most likely to look for such a notice.

d) Redistribution.

(i) Source. Covered Code may be distributed in Source Code form only to another Licensee (except for students as provided below). You may not offer or impose any terms on any Covered Code that alter the rights, requirements, or responsibilities of such Licensee. You may distribute Covered Code to students for use in connection with their course work and research projects undertaken at accredited educational institutions. Such students need not be Licensees, but must be given a copy of the notice set forth in Attachment A-3 and such notice must also be included in a file header or prominent location in the Source Code made available to such students.

(ii) Executable. You may distribute Executable version(s) of Covered Code to Licensees and other third parties only for the purpose of evaluation and comment in connection with Research Use by You and under a license of Your choice, but which limits use of such Executable version(s) of Covered Code only to that purpose.

(iii) Modified Class, Interface and Package Naming. In connection with Research Use by You only, You may use Original Contributor's class, interface and package names only to accurately reference or invoke the Source Code files You modify. Original Contributor grants to You a limited license to the extent necessary for such purposes.

(iv) Modifications. You expressly agree that any distribution, in whole or in part, of Modifications developed by You shall only be done pursuant to the term and conditions of this License.

e) Extensions.

(i) Covered Code. You may not include any Source Code of Community Code in any Extensions;

(ii) Publication. No later than the date on which You first distribute such Extension for Commercial Use, You must publish to the industry, on a non-confidential basis and free of all copyright restrictions with respect to reproduction and use, an accurate and current specification for any Extension. In addition, You must make available an appropriate test suite, pursuant to the same rights as the specification, sufficiently detailed to allow any third party reasonably skilled in the technology to produce implementations of the Extension compatible with the specification. Such test suites must be made available as soon as reasonably practicable but, in no event, later than ninety (90) days after Your first Commercial Use of the Extension. You must use reasonable efforts to promptly clarify and correct the specification and the test suite upon written request by Original Contributor.

(iii) Open. You agree to refrain from enforcing any Intellectual Property Rights You may have covering any interface(s) of Your Extension, which would prevent the implementation of such interface(s) by Original Contributor or any Licensee. This obligation does not prevent You from enforcing any Intellectual Property Right You have that would otherwise be infringed by an implementation of Your Extension.

(iv) Class, Interface and Package Naming. You may not add any packages, or any public or protected classes or interfaces with names that originate or might appear to originate from Original Contributor including, without limitation, package or class names which begin with "sun", "java", "javax", "jini", "net.jini", "com.sun" or their equivalents in any subsequent class, interface and/or package naming convention adopted by Original Contributor. It is specifically suggested that You name any new packages using the "Unique Package Naming Convention" as described in "The Java Language Specification" by James Gosling, Bill Joy, and Guy Steele, ISBN 0-201-63451-1, August 1996. Section 7.7 "Unique Package Names", on page 125 of this specification which states, in part:

"You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as "sun.com". You then reverse the name, component by component, to obtain, in this example, "Com.sun", and use this as a prefix for Your package names, using a convention developed within Your organization to further administer package names."

3.2. Additional Requirements and Responsibilities. Any additional requirements and responsibilities relating to the Technology are listed in Attachment F (Additional Requirements and Responsibilities), if applicable, and are hereby incorporated into this Section 3.

4. Versions of the License.

4.1. License Versions. Original Contributor may publish revised versions of the License from time to time. Each version will be given a distinguishing version number.

4.2. Effect. Once a particular version of Covered Code has been provided under a version of the License, You may always continue to use such Covered Code under the terms of that version of the License. You may also choose to use such Covered Code under the terms of any subsequent version of the License. No one other than Original Contributor has the right to promulgate License versions.

5. Disclaimer of Warranty.

5.1. COVERED CODE IS PROVIDED UNDER THIS LICENSE "AS IS," WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED CODE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. YOU AGREE TO BEAR THE ENTIRE RISK IN CONNECTION WITH YOUR USE AND DISTRIBUTION OF COVERED CODE UNDER THIS LICENSE. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT SUBJECT TO THIS DISCLAIMER.

5.2. You acknowledge that Original Code, Upgraded Code and Specifications are not designed or intended for use in (i) on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or (ii) in the design, construction, operation or maintenance of any nuclear facility. Original Contributor disclaims any express or implied warranty of fitness for such uses.

6. Termination.

6.1. By You. You may terminate this Research Use license at anytime by providing written notice to Original Contributor.

6.2. By Original Contributor. This License and the rights granted hereunder will terminate: (i) automatically if You fail to comply with the terms of this License and fail to cure such breach within 30 days of receipt of written notice of the breach; (ii) immediately in the event of circumstances specified in Sections 7.1 and 8.4; or (iii) at Original Contributor's discretion upon any action initiated in the first instance by You alleging that use or distribution by Original Contributor or any Licensee, of Original Code, Upgraded Code, Error Corrections or Shared Modifications contributed by You, or Specifications, infringe a patent owned or controlled by You.

6.3. Effect of Termination. Upon termination, You agree to discontinue use and return or destroy all copies of Covered Code in your possession. All sublicenses to the Covered Code which you have properly granted shall survive any termination of this License. Provisions which, by their nature, should remain in effect beyond the termination of this License shall survive including, without limitation, Sections 2.2, 3, 5, 7 and 8.

6.4. Each party waives and releases the other from any claim to compensation or indemnity for permitted or lawful termination of the business relationship established by this License.

7. Liability.

7.1. Infringement. Should any of the Original Code, Upgraded Code, TCK or Specifications ("Materials") become the subject of a claim of infringement, Original Contributor may, at its sole option, (i) attempt to procure the rights necessary for You to continue using the Materials, (ii) modify the Materials so that they are no longer infringing, or (iii) terminate Your right to use the Materials, immediately upon written notice, and refund to You the amount, if any, having then actually been paid by You to Original Contributor for the Original Code, Upgraded Code and TCK, depreciated on a straight line, five year basis.

7.2. LIMITATION OF LIABILITY. TO THE FULL EXTENT ALLOWED BY APPLICABLE LAW, ORIGINAL CONTRIBUTOR'S LIABILITY TO YOU FOR CLAIMS RELATING TO THIS LICENSE, WHETHER FOR BREACH OR IN TORT, SHALL BE LIMITED TO ONE HUNDRED PERCENT (100%) OF THE AMOUNT HAVING THEN ACTUALLY BEEN PAID BY YOU TO ORIGINAL CONTRIBUTOR FOR ALL COPIES LICENSED HEREUNDER OF THE PARTICULAR ITEMS GIVING RISE TO SUCH CLAIM, IF ANY. IN NO EVENT WILL YOU (RELATIVE TO YOUR SHARED MODIFICATIONS OR ERROR CORRECTIONS) OR ORIGINAL CONTRIBUTOR BE LIABLE FOR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THIS LICENSE (INCLUDING, WITHOUT LIMITATION, LOSS OF PROFITS, USE, DATA, OR OTHER ECONOMIC ADVANTAGE), HOWEVER IT ARISES AND ON ANY THEORY OF LIABILITY, WHETHER IN AN ACTION FOR CONTRACT, STRICT LIABILITY OR TORT (INCLUDING NEGLIGENCE) OR OTHERWISE, WHETHER OR NOT YOU OR ORIGINAL CONTRIBUTOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND NOTWITHSTANDING THE FAILURE OF ESSENTIAL PURPOSE OF ANY REMEDY.

8. Miscellaneous.

8.1. Trademark. You agree to comply with the then current Sun Trademark & Logo Usage Requirements accessible through the SCSL Webpage. Except as expressly provided in the License, You are granted no right, title or license to, or interest in, any Sun Trademarks. You agree not to (i) challenge Original Contributor's ownership or use of Sun Trademarks; (ii) attempt to register any Sun Trademarks, or any mark or logo substantially similar thereto; or (iii) incorporate any Sun Trademarks into your own trademarks, product names, service marks, company names, or domain names.

8.2. Integration. This License represents the complete agreement concerning the subject matter hereof.

8.3. Assignment. Original Contributor may assign this License, and its rights and obligations hereunder, in its sole discretion. You may assign the Research Use portions of this License to a third party upon prior written notice to Original Contributor (which may be provided via the Community Web-Server). You may not assign the Commercial Use license or TCK license, including by way of merger (regardless of whether You are the surviving entity) or acquisition, without Original Contributor's prior written consent.

8.4. Severability. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. Notwithstanding the foregoing, if You are prohibited by law from fully and specifically complying with Sections 2.2 or 3, this License will immediately terminate and You must immediately discontinue any use of Covered Code.

8.5. Governing Law. This License shall be governed by the laws of the United States and the State of California, as applied to contracts entered into and to be performed in California between California residents. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded.

8.6. Dispute Resolution.

a) Any dispute arising out of or relating to this License shall be finally settled by arbitration as set out herein, except that either party may bring any action, in a court of competent jurisdiction (which jurisdiction shall be exclusive), with respect to any dispute relating to such party's Intellectual Property Rights or with respect to Your compliance with the TCK license. Arbitration shall be administered: (i) by the American Arbitration Association (AAA), (ii) in accordance with the rules of the United Nations Commission on International Trade Law (UNCITRAL) (the "Rules") in effect at the time of arbitration as modified herein; and (iii) the arbitrator will apply the substantive laws of California and United States. Judgment upon the award rendered by the arbitrator may be entered in any court having jurisdiction to enforce such award.

b) All arbitration proceedings shall be conducted in English by a single arbitrator selected in accordance with the Rules, who must be fluent in English and be either a retired judge or practicing attorney having at least ten (10) years litigation experience and be reasonably familiar with the technology matters relative to the dispute. Unless otherwise agreed, arbitration venue shall be in London, Tokyo, or San Francisco, whichever is closest to defendant's principal business office. The arbitrator may award monetary damages only and nothing shall preclude either party from seeking provisional or emergency relief from a court of competent jurisdiction. The arbitrator shall have no authority to award damages in excess of those permitted in this License and any such award in excess is void. All awards will be payable in U.S. dollars and may include, for the prevailing party (i) pre-judgment award interest, (ii) reasonable attorneys' fees incurred in connection with the arbitration, and (iii) reasonable costs and expenses incurred in enforcing the award. The arbitrator will order each party to produce identified documents and respond to no more than twenty-five single question interrogatories.

8.7. Construction. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License.

8.8. U.S. Government End Users. The Covered Code is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Code with only those rights set forth herein. You agree to pass this notice to Your licensees.

8.9. Press Announcements. All press announcements relative to the execution of this License must be reviewed and approved by Original Contributor and You prior to release.

8.10. International Use.

a) Export/Import Laws. Covered Code is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Each party agrees to comply strictly with all such laws and regulations and acknowledges their responsibility to obtain such licenses to export, re-export, or import as may be required. You agree to pass these obligations to Your licensees.

b) Intellectual Property Protection. Due to limited intellectual property protection and enforcement in certain countries, You agree not to redistribute the Original Code, Upgraded Code, TCK and Specifications to any country other than the list of restricted countries on the SCSL Webpage.

8.11. Language. This License is in the English language only, which language shall be controlling in all respects, and all versions of this License in any other language shall be for accommodation only and shall not be binding on the parties to this License. All communications and notices made or given pursuant to this License, and all documentation and support to be provided, unless otherwise noted, shall be in the English language.

PLEASE READ THE TERMS OF THIS LICENSE CAREFULLY. BY CLICKING ON THE "ACCEPT" BUTTON BELOW YOU ARE ACCEPTING AND AGREEING TO THE TERMS AND CONDITIONS OF THIS LICENSE WITH SUN MICROSYSTEMS, INC. IF YOU ARE AGREEING TO THIS LICENSE ON BEHALF OF A COMPANY, YOU REPRESENT THAT YOU ARE AUTHORIZED TO BIND THE COMPANY TO SUCH A LICENSE. WHETHER YOU ARE ACTING ON YOUR OWN BEHALF, OR REPRESENTING A COMPANY, YOU MUST BE OF MAJORITY AGE AND BE OTHERWISE COMPETENT TO ENTER INTO CONTRACTS. IF YOU DO NOT MEET THIS CRITERIA OR YOU DO NOT AGREE TO ANY OF THE TERMS AND CONDITIONS OF THIS LICENSE, CLICK ON THE REJECT BUTTON TO EXIT.

ACCEPT REJECT

GLOSSARY

1. "Commercial Use" means any use (excluding Internal Deployment Use) or distribution, directly or indirectly of Compliant Covered Code by You to any third party, alone or bundled with any other software or hardware, for direct or indirect commercial or strategic gain or advantage, subject to execution of Attachment D by You and Original Contributor.

2. "Community Code" means the Original Code, Upgraded Code, Error Corrections, Shared Modifications, or any combination thereof.

3. "Community Webserver(s)" means the webservers designated by Original Contributor for posting Error Corrections and Shared Modifications.

4. "Compliant Covered Code" means Covered Code that complies with the requirements of the TCK.

5. "Contributor" means each Licensee that creates or contributes to the creation of any Error Correction or Shared Modification.

6. "Covered Code" means the Original Code, Upgraded Code, Modifications, or any combination thereof.

7. "Error Correction" means any change made to Community Code which conforms to the Specification and corrects the adverse effect of a failure of Community Code to perform any function set forth in or required by the Specifications.

8. "Executable" means Covered Code that has been converted to a form other than Source Code.

9. "Extension(s)" means any additional classes or other programming code and/or interfaces developed by or for You which: (i) are designed for use with the Technology; (ii) constitute an API for a library of computing functions or services; and (iii) are disclosed to third party software developers for the purpose of developing software which invokes such additional classes or other programming code and/or interfaces. The foregoing shall not apply to software development by Your subcontractors to be exclusively used by You.

10. "Intellectual Property Rights" means worldwide statutory and common law rights associated solely with (i) patents and patent applications; (ii) works of authorship including copyrights, copyright applications, copyright registrations and "moral rights"; (iii) the protection of trade and industrial secrets and confidential information; and (iv) divisions, continuations, renewals, and re-issuances of the foregoing now existing or acquired in the future.

11. "Internal Deployment Use" means use of Compliant Covered Code (excluding Research Use) within Your business or organization only by Your employees and/or agents, subject to execution of Attachment C by You and Original Contributor, if required.

12. "Licensee" means any party that has entered into and has in effect a version of this License with Original Contributor.

13. "Modification(s)" means (i) any change to Covered Code; (ii) any new file or other representation of computer program statements that contains any portion of Covered Code; and/or (iii) any new Source Code implementing any portion of the Specifications.

14. "Original Code" means the initial Source Code for the Technology as described on the Technology Download Site.

15. "Original Contributor" means Sun Microsystems, Inc., its affiliates and its successors and assigns.

16. "Reformatted Specifications" means any revision to the Specifications which translates or reformats the Specifications (as for example in connection with Your documentation) but which does not alter, subset or superset the functional or operational aspects of the Specifications.

17. "Research Use" means use and distribution of Covered Code only for Your research, development, educational or personal and individual use, and expressly excludes Internal Deployment Use and Commercial Use.

18. "SCSL Webpage" means the Sun Community Source license webpage located at http://sun.com/software/communitysource, or such other url that Original Contributor may designate from time to time.

19. "Shared Modifications" means Modifications provided by You, at Your option, pursuant to Section 2.2, or received by You from a Contributor pursuant to Section 2.3.

20. "Source Code" means computer program statements written in any high-level, readable form suitable for modification and development.

21. "Specifications" means the specifications for the Technology and other documentation, as designated on the Technology Download Site, as may be revised by Original Contributor from time to time.

22. "Sun Trademarks" means Original Contributor's SUN, JAVA, and JINI trademarks and logos, whether now used or adopted in the future.

23. "Technology" means the technology described in Attachment B, and Upgrades.

24. "Technology Compatibility Kit" or "TCK" means the test programs, procedures and/or other requirements, designated by Original Contributor for use in verifying compliance of Covered Code with the Specifications, in conjunction with the Original Code and Upgraded Code. Original Contributor may, in its sole discretion and from time to time, revise a TCK to correct errors and/or omissions and in connection with Upgrades.

25. "Technology Download Site" means the site(s) designated by Original Contributor for access to the Original Code, Upgraded Code, TCK and Specifications.

F.3 Apache License 2.0

NekoHTML is licensed under version 2 of the Apache License.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

1. You must give any other recipients of the Work or Derivative Works a copy of this License; and

2. You must cause any modified files to carry prominent notices stating that You changed the files; and

3. You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

4. If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided long with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

F.4 Common Public License 1.0

JUnit is distributed under the Common Public License, version 1.0

Common Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

b) in the case of each subsequent Contributor:

i) changes to the Program, and

ii) additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and

b) its license agreement:

i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

a) it must be made available under this Agreement; and

b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions

F.5 X License

ICU is distributed under the X License, also known as the MIT License.

F.6 Creative Commons Attribution-Sharealike 3.0 Unported License

Text drawn from the Wikipedia is licensed under the Creative Commons Attribution-Sharealike (CC-BY-SA) 3.0 Unported License, which we reproduce below. The creative commons website is `http://creativecommons.org/`.

F.6.1 Creative Commons Deed

5. For the avoidance of doubt:

1. (intentionally blank)

2. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

3. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

4. Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

1. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

2. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.

3. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

4. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

1. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

Index