



Quick answers to common problems

Natural Language Processing with Java and LingPipe Cookbook

Over 60 effective recipes to develop your Natural Language Processing (NLP) skills quickly and effectively

Breck Baldwin Krishna Dayanidhi

[PACKT] open source*
PUBLISHING community experience distilled

Natural Language Processing with Java and LingPipe Cookbook

Over 60 effective recipes to develop your Natural Language Processing (NLP) skills quickly and effectively

Breck Baldwin

Krishna Dayanidhi



BIRMINGHAM - MUMBAI

Natural Language Processing with Java and LingPipe Cookbook

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: November 2014

Production reference: 1241114

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-467-2

www.packtpub.com

Credits

Authors

Breck Baldwin
Krishna Dayanidhi

Reviewers

Aria Haghighi
Kshitij Judah
Karthik Raghunathan
Ataf Rahman

Commissioning Editor

Kunal Parikh

Acquisition Editor

Sam Wood

Content Development Editor

Ruchita Bhansali

Technical Editors

Mrunal M. Chavan
Shiny Poojary
Sebastian Rodrigues

Copy Editors

Janbal Dharmaraj
Karuna Narayanan
Merilyn Pereira

Project Coordinator

Kranti Berde

Proofreaders

Bridget Braund
Maria Gould
Ameesha Green
Lucy Rowland

Indexers

Monica Ajmera Mehta
Tejal Soni

Production Coordinator

Melwyn D'sa

Cover Work

Melwyn D'sa

About the Authors

Breck Baldwin is the Founder and President of Alias-i/LingPipe. The company focuses on system building for customers, education for developers, and occasional forays into pure research. He has been building large-scale NLP systems since 1996. He enjoys telemark skiing and wrote *DIY RC Airplanes from Scratch: The Brooklyn Aerodrome Bible for Hacking the Skies*, McGraw-Hill/TAB Electronics.

This book is dedicated to Peter Jackson, who hired me as a consultant for Westlaw, before I founded the company, and gave me the confidence to start it. He served on my advisory board until his untimely death, and I miss him terribly.

Fellow Aristotelian, Bob Carpenter, is the architect and developer behind the LingPipe API. It was his idea to make LingPipe open source, which opened many doors and led to this book.

Mitzi Morris has worked with us over the years and has been instrumental in our challenging NIH work, the author of tutorials, packages, and pitching in where it was needed.

Jeff Reynar was my office mate in graduate school when we hatched the idea of entering the MUC-6 competition, which was the prime mover for creation of the company; he now serves our advisory board.

Our volunteer reviewers deserve much credit; Doug Donahue and Rob Stupay were a big help. Packt Publishing reviewers made the book so much better; I thank Karthik Raghunathan, Altaf Rahman, and Kshitij Judah for their attention to detail and excellent questions and suggestions.

Our editors were the ever patient; Ruchita Bhansali who kept the chapters moving and provided excellent commentary, and Shiny Poojary, our thorough technical editor, who suffered so that you don't have to. Much thanks to both of you.

I could not have done this without my co-author, Krishna, who worked full-time and held up his side of the writing.

Many thanks to my wife, Karen, for her support throughout the book-writing process.

Krishna Dayanidhi has spent most of his professional career focusing on Natural Language Processing technologies. He has built diverse systems, from a natural dialog interface for cars to Question Answering systems at (different) Fortune 500 companies. He also confesses to building those automated speech systems for very large telecommunication companies. He's an avid runner and a decent cook.

I'd like to thank Bob Carpenter for answering many questions and for all his previous writings, including the tutorials and Javadocs that have informed and shaped this book. Thank you, Bob! I'd also like to thank my co-author, Breck, for convincing me to co-author this book and for tolerating all my quirks throughout the writing process.

I'd like to thank the reviewers, Karthik Raghunathan, Altaf Rahman, and Kshitij Judah, for providing essential feedback, which in some cases changed the entire recipe. Many thanks to Ruchita, our editor at Packt Publishing, for guiding, cajoling, and essentially making sure that this book actually came to be. Finally, thanks to Latha for her support, encouragement, and tolerance.

About the Reviewers

Karthik Raghunathan is a scientist at Microsoft, Silicon Valley, working on Speech and Natural Language Processing. Since first being introduced to the field in 2006, he has worked on diverse problems such as spoken dialog systems, machine translation, text normalization, coreference resolution, and speech-based information retrieval, leading to publications in esteemed conferences such as SIGIR, EMNLP, and AAAI. He has also had the privilege to be mentored by and work with some of the best minds in Linguistics and Natural Language Processing, such as Prof. Christopher Manning, Prof. Daniel Jurafsky, and Dr. Ron Kaplan.

Karthik currently works at the Bing Speech and Language Sciences group at Microsoft, where he builds speech-enabled conversational understanding systems for various Microsoft products such as the Xbox gaming console and the Windows Phone mobile operating system. He employs various techniques from speech processing, Natural Language Processing, machine learning, and data mining to improve systems that perform automatic speech recognition and natural language understanding. The products he has recently worked on at Microsoft include the new improved Kinect sensor for Xbox One and the Cortana digital assistant in Windows Phone 8.1. In his previous roles at Microsoft, Karthik worked on shallow dependency parsing and semantic understanding of web queries in the Bing Search team and on statistical spellchecking and grammar checking in the Microsoft Office team.

Prior to joining Microsoft, Karthik graduated with an MS degree in Computer Science (specializing in Artificial Intelligence), with a distinction in Research in Natural Language Processing from Stanford University. While the focus of his graduate research thesis was coreference resolution (the coreference tool from his thesis is available as part of the Stanford CoreNLP Java package), he also worked on the problems of statistical machine translation (leading Stanford's efforts for the GALE 3 Chinese-English MT bakeoff), slang normalization in text messages (codeveloping the Stanford SMS Translator), and situated spoken dialog systems in robots (helped in developing speech packages, now available as part of the open source Robot Operating System (ROS)).

Karthik's undergraduate work at the National Institute of Technology, Calicut, focused on building NLP systems for Indian languages. He worked on restricted domain-spoken dialog systems for Tamil, Telugu, and Hindi in collaboration with IIIT, Hyderabad. He also interned with Microsoft Research India on a project that dealt with scaling statistical machine translation for resource-scarce languages.

Karthik Raghunathan maintains a homepage at nlp.stanford.edu/~rkarthik/ and can be reached at kr@cs.stanford.edu.

Altaf Rahman is currently a research scientist at Yahoo Labs in California, USA. He works on search queries, understanding problems such as query tagging, query interpretation ranking, vertical search triggering, module ranking, and others. He earned his PhD degree from The University of Texas at Dallas on Natural Language Processing. His dissertation was on the conference resolution problem. Dr. Rahman has publications in major NLP conferences with over 200 citations. He has also worked on other NLP problems: Named Entity Recognition, Part of Speech Tagging, Statistical Parsers, Semantic Classifier, and so on. Earlier, he worked as a research intern in IBM Thomas J. Watson Research Center, Université Paris Diderot, and Google.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print, and bookmark content
- ▶ On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Simple Classifiers	7
Introduction	8
Deserializing and running a classifier	11
Getting confidence estimates from a classifier	14
Getting data from the Twitter API	19
Applying a classifier to a .csv file	22
Evaluation of classifiers – the confusion matrix	24
Training your own language model classifier	29
How to train and evaluate with cross validation	32
Viewing error categories – false positives	37
Understanding precision and recall	39
How to serialize a LingPipe object – classifier example	40
Eliminate near duplicates with the Jaccard distance	42
How to classify sentiment – simple version	45
Chapter 2: Finding and Working with Words	51
Introduction	51
Introduction to tokenizer factories – finding words in a character stream	52
Combining tokenizers – lowercase tokenizer	56
Combining tokenizers – stop word tokenizers	58
Using Lucene/Solr tokenizers	60
Using Lucene/Solr tokenizers with LingPipe	62
Evaluating tokenizers with unit tests	66
Modifying tokenizer factories	68
Finding words for languages without white spaces	70

Chapter 3: Advanced Classifiers	75
Introduction	75
A simple classifier	76
Language model classifier with tokens	78
Naïve Bayes	79
Feature extractors	85
Logistic regression	87
Multithreaded cross validation	93
Tuning parameters in logistic regression	97
Customizing feature extraction	103
Combining feature extractors	105
Classifier-building life cycle	106
Linguistic tuning	114
Thresholding classifiers	119
Train a little, learn a little – active learning	126
Annotation	136
Chapter 4: Tagging Words and Tokens	141
Introduction	141
Interesting phrase detection	142
Foreground- or background-driven interesting phrase detection	145
Hidden Markov Models (HMM) – part-of-speech	149
N-best word tagging	151
Confidence-based tagging	153
Training word tagging	154
Word-tagging evaluation	160
Conditional random fields (CRF) for word/token tagging	163
Modifying CRFs	167
Chapter 5: Finding Spans in Text – Chunking	173
Introduction	174
Sentence detection	174
Evaluation of sentence detection	178
Tuning sentence detection	182
Marking embedded chunks in a string – sentence chunk example	184
Paragraph detection	186
Simple noun phrases and verb phrases	189
Regular expression-based chunking for NER	191
Dictionary-based chunking for NER	193
Translating between word tagging and chunks – BIO codec	195

HMM-based NER	198
Mixing the NER sources	205
CRFs for chunking	208
NER using CRFs with better features	214
Chapter 6: String Comparison and Clustering	221
Introduction	221
Distance and proximity – simple edit distance	222
Weighted edit distance	224
The Jaccard distance	227
The Tf-Idf distance	230
Using edit distance and language models for spelling correction	234
The case restoring corrector	239
Automatic phrase completion	240
Single-link and complete-link clustering using edit distance	243
Latent Dirichlet allocation (LDA) for multitopic clustering	248
Chapter 7: Finding Coreference Between Concepts/People	257
Introduction	257
Named entity coreference with a document	258
Adding pronouns to coreference	261
Cross-document coreference	266
The John Smith problem	281
Index	291

Preface

Welcome to the book you will want to have by your side when you cross the door of a new consulting gig or take on a new Natural Language Processing (NLP) problem. This book starts as a private repository of LingPipe recipes that Baldwin continually referred to when facing repeated but twitchy NLP problems with system building. We are an open source company but the code never merited sharing. Now they are shared.

Honestly, the LingPipe API is an intimidating and opaque edifice to code against like any rich and complex Java API. Add in the "black arts" quality needed to get NLP systems working and we have the perfect conditions to satisfy the need for a recipe book that minimizes theory and maximizes the practicality of getting the job done with best practices sprinkled in from 20 years in the business.

This book is about getting the job done; damn the theory! Take this book and build the next generation of NLP systems and send us a note about what you did.

LingPipe is the best tool on the planet to build NLP systems with; this book is the way to use it.

What this book covers

Chapter 1, Simple Classifiers, explains that a huge percentage of NLP problems are actually classification problems. This chapter covers very simple but powerful classifiers based on character sequences and then brings in evaluation techniques such as cross-validation and metrics such as precision, recall, and the always-BS-resisting confusion matrix. You get to train yourself on your own and download data from Twitter. The chapter ends with a simple sentiment example.

Chapter 2, Finding and Working with Words, is exactly as boring as it sounds but there are some high points. The last recipe will show you how to tokenize Chinese/Japanese/Vietnamese languages, which doesn't have whitespaces, to help define words. We will show you how to wrap Lucene tokenizers, which cover all kinds of fun languages such as Arabic. Almost everything later in the book relies on tokenization.

Chapter 3, Advanced Classifiers, introduces the star of modern NLP systems—logistic regression classifiers. 20 years of hard-won experience lurks in this chapter. We will address the life cycle around building classifiers and how to create training data, cheat when creating training data with active learning, and how to tune and make the classifiers work faster.

Chapter 4, Tagging Words and Tokens, explains that language is about words. This chapter focuses on ways of applying categories to tokens, which in turn drives many of the high-end uses of LingPipe such as entity detection (people/places/orgs in text), part-of-speech tagging, and more. It starts with tag clouds, which have been described as "mullet of the Internet" and ends with a foundational recipe for conditional random fields (CRF), which can provide state-of-the-art performance for entity-detection tasks. In between, we will address confidence-tagged words, which is likely to be a very important dimension of more sophisticated systems.

Chapter 5, Finding Spans in Text – Chunking, shows that text is not words alone. It is collections of words, usually in spans. This chapter will advance from word tagging to span tagging, which brings in capabilities such as finding sentences, named entities, and basal NPs and VPs. The full power of CRFs are addressed with discussions on feature extraction and tuning. Dictionary approaches are discussed as they are ways of combining chunkings.

Chapter 6, String Comparison and Clustering, focuses on comparing text with each other, independent of a trained classifier. The technologies range from the hugely practical spellchecking to the hopeful but often frustrating Latent Dirichlet Allocation (LDA) clustering approach. Less presumptive technologies such as single-link and complete-link clustering have driven major commercial successes for us. Don't ignore this chapter.

Chapter 7, Finding Coreference Between Concepts/People, lays the future but unfortunately, you won't get the ultimate recipe, just our best efforts so far. This is one of the bleeding edges of industrial and academic NLP efforts that has tremendous potential. Potential is why we include our efforts to help grease the way to see this technology in use.

What you need for this book

You need some NLP problems and a solid foundation in Java, a computer, and a developer-savvy approach.

Who this book is for

If you have NLP problems or you want to educate yourself in current NLP issues, this book is for you. With some creativity, you can train yourself into being a solid NLP developer, a beast so rare that they are seen about as often as unicorns, with the result of more interesting job prospects in hot technology areas such as Silicon Valley or New York City.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Java is a pretty awful language to put into a recipe book with a 66-character limit on lines for code. The overriding convention is that the code is ugly and we apologize.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Once the string is read in from the console, then `classifier.classify(input)` is called, which returns `Classification`."

A block of code is set as follows:

```
public static List<String[]> filterJaccard(List<String[]> texts,
    TokenizerFactory tokFactory, double cutoff) {
    JaccardDistance jaccardD = new JaccardDistance(tokFactory);
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:


```
public static void consoleInputBestCategory(
    BaseClassifier<CharSequence> classifier) throws IOException {
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(System.in));
    while (true) {
        System.out.println("\nType a string to be classified. " + "
            Empty string to quit.");
        String data = reader.readLine();
        if (data.equals("")) {
            return;
        }
        Classification classification = classifier.classify(data);
        System.out.println("Best Category: " +
            classification.bestCategory());
    }
}
```

Any command-line input or output is written as follows:

```
tar -xvzf lingpipeCookbook.tgz
```


New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on **Create a new application**."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Send hate/love/neutral e-mails to cookbook@lingpipe.com. We do care, we won't do your homework for you or prototype your startup for free, but do talk to us.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

We do offer consulting services and even have a pro-bono (free) program as well as a start up support program. NLP is hard, this book is most of what we know but perhaps we can help more.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

All the source for the book is available at <http://alias-i.com/book.html>.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

Hit <http://lingpipe.com> and go to our forum for the best place to get questions answered and see if you have a solution already.

1

Simple Classifiers

In this chapter, we will cover the following recipes:

- ▶ Deserializing and running a classifier
- ▶ Getting confidence estimates from a classifier
- ▶ Getting data from the Twitter API
- ▶ Applying a classifier to a `.csv` file
- ▶ Evaluation of classifiers – the confusion matrix
- ▶ Training your own language model classifier
- ▶ How to train and evaluate with cross validation
- ▶ Viewing error categories – false positives
- ▶ Understanding precision and recall
- ▶ How to serialize a LingPipe object – classifier example
- ▶ Eliminate near duplicates with the Jaccard distance
- ▶ How to classify sentiment – simple version

Introduction

This chapter introduces the LingPipe toolkit in the context of its competition and then dives straight into text classifiers. Text classifiers assign a category to text, for example, they assign the language to a sentence or tell us if a tweet is positive, negative, or neutral in sentiment. This chapter covers how to use, evaluate, and create text classifiers based on language models. These are the simplest machine learning-based classifiers in the LingPipe API. What makes them simple is that they operate over characters only—later, classifiers will have notions of words/tokens and even more. However, don't be fooled, character-language models are ideal for language identification, and they were the basis of some of the world's earliest commercial sentiment systems.

This chapter also covers crucial evaluation infrastructure—it turns out that almost everything we do turns out to be a classifier at some level of interpretation. So, do not skimp on the power of cross validation, definitions of precision/recall, and F-measure.

The best part is that you will learn how to programmatically access Twitter data to train up and evaluate your own classifiers. There is a boring bit concerning the mechanics of reading and writing LingPipe objects from/to disk, but other than that, this is a fun chapter. The goal of this chapter is to get you up and running quickly with the basic care and feeding of machine-learning techniques in the domain of **natural language processing (NLP)**.

LingPipe is a Java toolkit for NLP-oriented applications. This book will show you how to solve common NLP problems with LingPipe in a problem/solution format that allows developers to quickly deploy solutions to common tasks.

LingPipe and its installation

LingPipe 1.0 was released in 2003 as a dual-licensed open source NLP Java library. At the time of writing this book, we are coming up on 2000 hits on Google Scholar and have thousands of commercial installs, ranging from universities to government agencies to Fortune 500 companies.

Current licensing is either AGPL (<http://www.gnu.org/licenses/agpl-3.0.html>) or our commercial license that offers more traditional features such as indemnification and non-sharing of code as well as support.

Projects similar to LingPipe

Nearly all NLP projects have awful acronyms so we will lay bare our own. **LingPipe** is the short form for **linguistic pipeline**, which was the name of the `cvs` directory in which Bob Carpenter put the initial code.

LingPipe has lots of competition in the NLP space. The following are some of the more popular ones with a focus on Java:

- ▶ **NLTK**: This is the dominant Python library for NLP processing.
- ▶ **OpenNLP**: This is an Apache project built by a bunch of smart folks.
- ▶ **JavaNLP**: This is a rebranding of Stanford NLP tools, again built by a bunch of smart folks.
- ▶ **ClearTK**: This is a University of Boulder toolkit that wraps lots of popular machine learning frameworks.
- ▶ **DkPro**: Technische Universität Darmstadt from Germany produced this UIMA-based project that wraps many common components in a useful manner. UIMA is a common framework for NLP.
- ▶ **GATE**: GATE is really more of a framework than competition. In fact, LingPipe components are part of their standard distribution. It has a nice graphical "hook the components up" capability.
- ▶ **Learning Based Java (LBJ)**: LBJ is a special-purpose programming language based on Java, and it is geared toward machine learning and NLP. It was developed at the Cognitive Computation Group of the University of Illinois at Urbana Champaign.
- ▶ **Mallet**: This name is the short form of **MACHINE Learning for Language Toolkit**. Apparently, reasonable acronym generation is short in supply these days. Smart folks built this too.

Here are some pure machine learning frameworks that have broader appeal but are not necessarily tailored for NLP tasks:

- ▶ **Vowpal Wabbit**: This is very focused on scalability around Logistic Regression, Latent Dirichlet Allocation, and so on. Smart folks drive this.
- ▶ **Factorie**: It is from UMass, Amherst and an alternative offering to Mallet. Initially it focused primarily on graphic models, but now it also supports NLP tasks.
- ▶ **Support Vector Machine (SVM)**: SVM light and `libsvm` are very popular SVM implementations. There is no SVM implementation in LingPipe, because logistic regression does this as well.

So, why use LingPipe?

It is very reasonable to ask why choose LingPipe with such outstanding free competition mentioned earlier. There are a few reasons:

- ▶ **Documentation**: The class-level documentation in LingPipe is very thorough. If the work is based on academic work, that work is cited. Algorithms are laid out, the underlying math is explained, and explanations are precise. What the documentation lacks is a "how to get things done" perspective; however, this is covered in this book.

- ▶ **Enterprise/server optimized:** LingPipe is designed from the ground up for server applications, not for command-line usage (though we will be using the command line extensively throughout the book).
- ▶ **Coded in the Java dialect:** LingPipe is a native Java API that is designed according to standard Java class design principles (Joshua Bloch's *Effective Java*, by Addison-Wesley), such as consistency checks on construction, immutability, type safety, backward-compatible serializability, and thread safety.
- ▶ **Error handling:** Considerable attention is paid to error handling through exceptions and configurable message streams for long-running processes.
- ▶ **Support:** LingPipe has paid employees whose job is to answer your questions and make sure that LingPipe is doing its job. The rare bug gets fixed in under 24 hours typically. They respond to questions very quickly and are very willing to help people.
- ▶ **Consulting:** You can hire experts in LingPipe to build systems for you. Generally, they teach developers how to build NLP systems as a byproduct.
- ▶ **Consistency:** The LingPipe API was designed by one person, Bob Carpenter, with an obsession of consistency. While it is not perfect, you will find a regularity and eye to design that can be missing in academic efforts. Graduate students come and go, and the resulting contributions to university toolkits can be quite varied.
- ▶ **Open source:** There are many commercial providers, but their software is a black box. The open source nature of LingPipe provides transparency and confidence that the code is doing what we ask it to do. When the documentation fails, it is a huge relief to have access to code to understand it better.

Downloading the book code and data

You will need to download the source code for this cookbook, with supporting models and data from <http://alias-i.com/book.html>. Untar and uncompress it using the following command:

```
tar -xvzf lingpipeCookbook.tgz
```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Alternatively, your operating system might provide other ways of extracting the archive. All recipes assume that you are running the commands in the resulting cookbook directory.

Downloading LingPipe

Downloading LingPipe is not strictly necessary, but you will likely want to be able to look at the source and have a local copy of the Javadoc.

The download and installation instructions for LingPipe can be found at <http://alias-i.com/lingpipe/web/install.html>.

The examples from this chapter use command-line invocation, but it is assumed that the reader has sufficient development skills to map the examples to their preferred IDE/ant or other environment.

Deserializing and running a classifier

This recipe does two things: introduces a very simple and effective language ID classifier and demonstrates how to deserialize a LingPipe class. If you find yourself here from a later chapter, trying to understand deserialization, I encourage you to run the example program anyway. It will take 5 minutes, and you might learn something useful.

Our language ID classifier is based on character language models. Each language model gives you the probability of the text, given that it is generated in that language. The model that is most familiar with the text is the first best fit. This one has already been built, but later in the chapter, you will learn to make your own.

How to do it...

Perform the following steps to deserialize and run a classifier:

1. Go to the `cookbook` directory for the book and run the command for OSX, Unix, and Linux:

```
java -cp lingpipe-cookbook.1.0.jar:lib/lingpipe-4.1.0.jar com.lingpipe.cookbook.chapter1.RunClassifierFromDisk
```

For Windows invocation (quote the classpath and use `;` instead of `:`):

```
java -cp "lingpipe-cookbook.1.0.jar;lib\lingpipe-4.1.0.jar" com.lingpipe.cookbook.chapter1.RunClassifierFromDisk
```

We will use the Unix style command line in this book.

2. The program reports the model being loaded and a default, and prompts for a sentence to classify:

```
Loading: models/3LangId.LMClassifier
```

```
Type a string to be classified. Empty string to quit.
```

```
The rain in Spain falls mainly on the plain.
```

```
english
```

```
Type a string to be classified. Empty string to quit.
```

```
la lluvia en España cae principalmente en el llano.
```

```
spanish
```

```
Type a string to be classified. Empty string to quit.
```

```
スペインの雨は主に平野に落ちる。
```

```
japanese
```

3. The classifier is trained on English, Spanish, and Japanese. We have entered an example of each—to get some Japanese, go to <http://ja.wikipedia.org/wiki/>. These are the only languages it knows about, but it will guess on any text. So, let's try some Arabic:

```
Type a string to be classified. Empty string to quit.
```

```
• لهس يلع اساساً عوي اي نابسا اي ف رطلم
```

```
japanese
```

4. It thinks it is Japanese because this language has more characters than English or Spanish. This in turn leads that model to expect more unknown characters. All the Arabic characters are unknown.
5. If you are working with a Windows terminal, you might encounter difficulty entering UTF-8 characters.

How it works...

The code in the jar is `cookbook/src/com/lingpipe/cookbook/chapter1/RunClassifierFromDisk.java`. What is happening is that a pre-built model for language identification is deserialized and made available. It has been trained on English, Japanese, and Spanish. The training data came from Wikipedia pages for each language. You can see the data in `data/3LangId.csv`. The focus of this recipe is to show you how to deserialize the classifier and run it—training is handled in the *Training your own language model classifier* recipe in this chapter. The entire code for the `RunClassifierFromDisk.java` class starts with the package; then it imports the start of the `RunClassifierFromDisk` class and the start of `main()`:

```
package com.lingpipe.cookbook.chapter1;
import java.io.File;
```

```
import java.io.IOException;

import com.aliasi.classify.BaseClassifier;
import com.aliasi.util.AbstractExternalizable;
import com.lingpipe.cookbook.Util;
public class RunClassifierFromDisk {
    public static void main(String[] args) throws
        IOException, ClassNotFoundException {
```

The preceding code is a very standard Java code, and we present it without explanation. Next is a feature in most recipes that supplies a default value for a file if the command line does not contain one. This allows you to use your own data if you have it, otherwise it will run from files in the distribution. In this case, a default classifier is supplied if there is no argument on the command line:

```
String classifierPath = args.length > 0 ? args[0]
: "models/3LangId.LMClassifier";
System.out.println("Loading: " + classifierPath);
```

Next, we will see how to deserialize a classifier or another LingPipe object from disk:

```
File serializedClassifier = new File(classifierPath);
@SuppressWarnings("unchecked")
BaseClassifier<String> classifier
    = (BaseClassifier<String>)
    AbstractExternalizable.readObject(serializedClassifier);
```

The preceding code snippet is the first LingPipe-specific code, where the classifier is built using the static `AbstractExternalizable.readObject` method.

This class is employed throughout LingPipe to carry out a compilation of classes for two reasons. First, it allows the compiled objects to have final variables set, which supports LingPipe's extensive use of immutables. Second, it avoids the messiness of exposing the I/O methods required for externalization and deserialization, most notably, the no-argument constructor. This class is used as the superclass of a private internal class that does the actual compilation. This private internal class implements the required no-arg constructor and stores the object required for `readResolve()`.



The reason we use `Externalizable` instead of `Serializable` is to avoid breaking backward compatibility when changing any method signatures or member variables. `Externalizable` extends `Serializable` and allows control of how the object is read or written. For more information on this, refer to the excellent chapter on serialization in Josh Bloch's book, *Effective Java, 2nd Edition*.

`BaseClassifier<E>` is the foundational classifier interface, with `E` being the type of object being classified in `LingPipe`. Look at the Javadoc to see the range of classifiers that implements the interface—there are 10 of them. Deserializing to `BaseClassifier<E>` hides a good bit of complexity, which we will explore later in the *How to serialize a LingPipe object – classifier example* recipe in this chapter.

The last line calls a utility method, which we will use frequently in this book:

```
Util.consoleInputBestCategory(classifier);
```

This method handles interactions with the command line. The code is in `src/com/lingpipe/cookbook/Util.java`:

```
public static void consoleInputBestCategory(
    BaseClassifier<CharSequence> classifier) throws IOException {
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(System.in));
    while (true) {
        System.out.println("\nType a string to be classified. "
            + " Empty string to quit.");
        String data = reader.readLine();
        if (data.equals("")) {
            return;
        }
        Classification classification = classifier.classify(data);
        System.out.println("Best Category: " +
            classification.bestCategory());
    }
}
```

Once the string is read in from the console, then `classifier.classify(input)` is called, which returns `Classification`. This, in turn, provides a `String` label that is printed out. That's it! You have run a classifier.

Getting confidence estimates from a classifier

Classifiers tend to be a lot more useful if they give more information about how confident they are of the classification—this is usually a score or a probability. We often threshold classifiers to help fit the performance requirements of an installation. For example, if it was vital that the classifier never makes a mistake, then we could require that the classification be very confident before committing to a decision.

LingPipe classifiers exist on a hierarchy based on the kinds of estimates they provide. The backbone is a series of interfaces—don't freak out; it is actually pretty simple. You don't need to understand it now, but we do need to write it down somewhere for future reference:

- ▶ `BaseClassifier<E>`: This is just your basic classifier of objects of type `E`. It has a `classify()` method that returns a classification, which in turn has a `bestCategory()` method and a `toString()` method that is of some informative use.
- ▶ `RankedClassifier<E>` extends `BaseClassifier<E>`: The `classify()` method returns `RankedClassification`, which extends `Classification` and adds methods for `category(int rank)` that says what the 1st to *n*th classifications are. There is also a `size()` method that indicates how many classifications there are.
- ▶ `ScoredClassifier<E>` extends `RankedClassifier<E>`: The returned `ScoredClassification` adds a `score(int rank)` method.
- ▶ `ConditionalClassifier<E>` extends `RankedClassifier<E>`: `ConditionalClassification` produced by this has the property that the sum of scores for all categories must sum to 1 as accessed via the `conditionalProbability(int rank)` method and `conditionalProbability(String category)`. There's more; you can read the Javadoc for this. This classification will be the work horse of the book when things get fancy, and we want to know the confidence that the tweet is English versus the tweet is Japanese versus the tweet is Spanish. These estimates will have to sum to 1.
- ▶ `JointClassifier<E>` extends `ConditionalClassifier<E>`: This provides `JointClassification` of the input and category in the space of all the possible inputs, and all such estimates sum to 1. This is a sparse space, so values are log based to avoid underflow errors. We don't see a lot of use of this estimate directly in production.

It is obvious that there has been a great deal of thought put into the classification stack presented. This is because huge numbers of industrial NLP problems are handled by a classification system in the end.

It turns out that our simplest classifier—in some arbitrary sense of simple—produces the richest estimates, which are joint classifications. Let's dive in.

Getting ready

In the previous recipe, we blithely deserialized to `BaseClassifier<String>` that hid all the details of what was going on. The reality is a bit more complex than suggested by the hazy abstract class. Note that the file on disk that was loaded is named `3LangId.LMClassifier`. By convention, we name serialized models with the type of object it will deserialize to, which, in this case, is `LMClassifier`, and it extends `BaseClassifier`. The most specific typing for the classifier is:

```
LMClassifier<CompiledNGramBoundaryLM,  
    MultivariateDistribution> classifier  
    = (LMClassifier <CompiledNGramBoundaryLM,  
        MultivariateDistribution>) AbstractExternalizable.readObject(new  
        File(args[0]));
```

The cast to `LMClassifier<CompiledNGramBoundaryLM, MultivariateDistribution>` specifies the type of distribution to be `MultivariateDistribution`. The Javadoc for `com.aliasi.stats.MultivariateDistribution` is quite explicit and helpful in describing what this is.



`MultivariateDistribution` implements a discrete distribution over a finite set of outcomes, numbered consecutively from zero.



The Javadoc goes into a lot of detail about `MultivariateDistribution`, but it basically means that we can have an *n*-way assignment of probabilities that sum to 1.

The next class in the cast is for `CompiledNGramBoundaryLM`, which is the "memory" of the `LMClassifier`. In fact, each language gets its own. This means that English will have a separate language model from Spanish and so on. There are eight different kinds of language models that could have been used as this part of the classifier—consult the Javadoc for the `LanguageModel` interface. Each **language model (LM)** has the following properties:

- ▶ The LM will provide a probability that it generated the text provided. It is robust against data that it has not seen before, in the sense that it won't crash or give a zero probability. Arabic just comes across as a sequence of unknown characters for our example.
- ▶ The sum of all the possible character sequence probabilities of any length is 1 for boundary LMs. Process LMs sum the probability to 1 over all sequences of the same length. Look at the Javadoc for how this bit of math is done.
- ▶ Each language model has no knowledge of data outside of its category.

- ▶ The classifier keeps track of the marginal probability of the category and factors this into the results for the category. Marginal probability is saying that we tend to see two-thirds English, one-sixth Spanish, and one-sixth Japanese in Disney tweets. This information is combined with the LM estimates.
- ▶ The LM is a compiled version of `LanguageModel.Dynamic` that we will cover in the later recipes that discuss training.

`LMClassifier` that is constructed wraps these components into a classifier.

Luckily, the interface saves the day with a more aesthetic deserialization:

```
JointClassifier<String> classifier = (JointClassifier<String>)
AbstractExternalizable.readObject(new File(classifierPath));
```

The interface hides the guts of the implementation nicely and this is what we are going with in the example program.

How to do it...

This recipe is the first time we start peeling away from what classifiers can do, but first, let's play with it a bit:

1. Get your magic shell genie to conjure a command prompt with a Java interpreter and type:

```
java -cp lingpipe-cookbook.1.0.jar:lib/lingpipe-4.1.0.jar: com.
lingpipe.cookbook.chapter1.RunClassifierJoint
```

2. We will enter the same data as we did earlier:

```
Type a string to be classified. Empty string to quit.
The rain in Spain falls mainly on the plain.
Rank Categ Score    P(Category|Input) log2 P(Category,Input)
0=english -3.60092 0.9999999999          -165.64233893156052
1=spanish -4.50479 3.04549412621E-13    -207.2207276413206
2=japanese -14.369 7.6855682344E-150    -660.989401136873
```

As described, `JointClassification` carries through all the classification metrics in the hierarchy rooted at `Classification`. Each level of classification shown as follows adds to the classifiers preceding it:

- ▶ `Classification` provides the first best category as the rank 0 category.
- ▶ `RankedClassification` adds an ordering of all the possible categories with a lower rank corresponding to greater likelihood of the category. The `rank` column reflects this ordering.

- ▶ `ScoredClassification` adds a numeric score to the ranked output. Note that scores might or might not compare well against other strings being classified depending on the type of classifier. This is the column labeled `Score`. To understand the basis of this score, consult the relevant Javadoc.
- ▶ `ConditionalClassification` further refines the score by making it a category probability conditioned on the input. The probabilities of all categories will sum up to 1. This is the column labeled `P(Category|Input)`, which is the traditional way to write *probability of the category given the input*.
- ▶ `JointClassification` adds the \log_2 (log base 2) probability of the input and the category—this is the joint probability. The probabilities of all categories and inputs will sum up to 1, which is a very large space indeed with very low probabilities assigned to any pair of category and string. This is why \log_2 values are used to prevent numerical underflow. This is the column labeled `log 2 P(Category, Input)`, which is translated as the *log₂ probability of the category and input*.

Look at the Javadoc for the `com.aliasei.classify` package for more information on the metrics and classifiers that implement them.

How it works...

The code is in `src/com/lingpipe/cookbook/chapter1/RunClassifierJoint.java`, and it deserializes to a `JointClassifier<CharSequence>`:

```
public static void main(String[] args) throws IOException,
    ClassNotFoundException {
    String classifierPath = args.length > 0 ? args[0] :
        "models/3LangId.LMClassifier";
    @SuppressWarnings("unchecked")
    JointClassifier<CharSequence> classifier
        = (JointClassifier<CharSequence>)
        AbstractExternalizable.readObject(new File(classifierPath));
    Util.consoleInputPrintClassification(classifier);
}
```

It makes a call to `Util.consoleInputPrintClassification(classifier)`, which minimally differs from `Util.consoleInputBestCategory(classifier)`, in that it uses the `toString()` method of classification to print. The code is as follows:

```
public static void consoleInputPrintClassification(BaseClassifier<CharSequence>
    classifier) throws IOException {
    BufferedReader reader = new BufferedReader(new
        InputStreamReader(System.in));
    while (true) {
```

```

        System.out.println("\nType a string to be classified." + Empty
string to quit.");
        String data = reader.readLine();
        if (data.equals("")) {
            return;
        }
        Classification classification = classifier.classify(data);
        System.out.println(classification);
    }
}

```

We got a richer output than we expected, because the type is `Classification`, but the `toString()` method will be applied to the runtime type `JointClassification`.

See also

- There is detailed information in *Chapter 6, Character Language Models of Text Analysis with LingPipe 4*, by Bob Carpenter and Breck Baldwin, *LingPipe Publishing* (<http://alias-i.com/lingpipe-book/lingpipe-book-0.5.pdf>) on language models.

Getting data from the Twitter API

We use the popular `twitter4j` package to invoke the Twitter Search API, and search for tweets and save them to disk. The Twitter API requires authentication as of Version 1.1, and we will need to get authentication tokens and save them in the `twitter4j.properties` file before we get started.

Getting ready

If you don't have a Twitter account, go to `twitter.com/signup` and create an account. You will also need to go to `dev.twitter.com` and sign in to enable yourself for the developer account. Once you have a Twitter login, we'll be on our way to creating the Twitter OAuth credentials. Be prepared for this process to be different from what we are presenting. In any case, we will supply example results in the `data` directory. Let's now create the Twitter OAuth credentials:

1. Log in to `dev.twitter.com`.
2. Find the little pull-down menu next to your icon on the top bar.
3. Choose **My Applications**.
4. Click on **Create a new application**.
5. Fill in the form and click on **Create a Twitter application**.