



Natural Language Processing **IN ACTION**

Understanding, analyzing, and generating text with Python

Hobson Lane
Cole Howard
Hannes Max Hapke

MEAP



MANNING



MEAP Edition
Manning Early Access Program
Natural Language Processing in Action
Understanding, analyzing, and generating text with Python
Version 9

Copyright 2018 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Natural Language Processing in Action: Understanding, analyzing, and generating text with Python*.

We came together to write this book after discovering the power of recent NLP algorithms that model natural language and generate sensible replies to a variety of statements, questions, and search queries. Over the past two years we've trained chatbots to mimic the language and style of movie characters, built NLP pipelines that can compose poetry, and used semantic analysis to identify meaningful job matches for resumes. We had so much fun, and found it so surprisingly powerful, that we wanted to share that experience with you.

We hope the examples and explanations we've put together will help you apply NLP to your own problems. If you have some Python development experience you should be able to adapt the code examples to a broad range of applications. And if you have some machine learning experience you may even be able to improve upon the performance of these algorithms. We will show you software examples for a wide range of problems, from extracting structured, semantic information from English text to building a chatbot that can communicate with you and your customers. And we don't merely give you example code snippets, but provide several complete NLP pipelines on GitHub, including a chatbot, incorporating the tools and datasets we help you assemble step by step throughout this book.

In Section 1 you will learn how to parse English text into numerical vectors suitable for input to a text classification or search pipeline. In Section 2 you will learn how to reduce the dimensions of these high-dimensional vectors into vectors that capture the meaning of text, and how to use these vectors to train a semantic search pipeline and a chatbot that can respond to the meaning of text. In Section 3 you will learn how to extend the capability of your machine learning pipeline using neural networks and deep learning, starting with Convolutional Neural Networks and concluding with generative sequence-based neural network architectures. In this final section we'll show you how to use these neural networks to generate natural language text, and we'll reveal to you which sentences and paragraphs in this book were composed using these neural network pipelines.

Between the lines of Python and English in this book you may detect a theme, a daring hypothesis. The hypothesis is that the development of prosocial machine intelligence depends on a diverse "gene pool" of intelligent machines that understand and can express themselves in natural language, a language accessible to humans. And the challenge of developing intelligent machines that behave in a prosocial way is a pressing concern. One of the most pressing challenges of the 21st century is the "Control Problem." Research laboratories around the globe, from DeepMind in Europe to Vector Institute and OpenAI in North America, are investing millions trying to understand how we can direct the development of machine intelligence for the benefit of mankind and prevent it from out-competing us for control of this planet's resources. We hope to give you the tools to

contribute to this ecosystem of prosocial intelligent machines, machines that may help make this "Terminator" doomsday scenario less likely.

We look forward to getting your feedback in the online Author Forum and maybe even interacting with your chatbot and seeing what your semantic search engine can find. This book is a community effort. We hope you'll contribute your feedback and ideas, expanding the collective intelligence of this community of machines and humans.

—Hobson, Cole, and Hannes

brief contents

Acknowledgments

PART 1: WORDY MACHINES

- 1 *The Language of Thought*
- 2 *Build your vocabulary (word tokenization)*
- 3 *Math with Words (TF-IDF Vectors)*
- 4 *Finding Meaning in Word Counts (Semantic Analysis)*

PART 2: DEEPER LEARNING (NEURAL NETWORKS)

- 5 *Baby Steps with Neural Networks (Perceptrons and Backpropagation)*
- 6 *Reasoning with Word Vectors (Word2vec)*
- 7 *Getting Words in Order with Convolutional Neural Networks (CNNs)*
- 8 *Loopy (Recurrent) Neural Networks (RNNs)*
- 9 *Improving Retention with Long Short-Term Memory Networks (LSTMs)*
- 10 *Sequence to Sequence Models and Attention (Generative Models)*

PART 3: GETTING REAL (REAL WORLD NLP CHALLENGES)

- 11 *Information Extraction (Named Entity Extraction and Question Answering)*
- 12 *Getting Chatty (Dialog Engines)*
- 13 *Scaling Up (Optimization, Parallelization and Batch Processing)*

APPENDIXES

- A *Your NLP tools*
- B *Playful Python and regular expressions*
- C *Vectors and matrices (linear algebra fundamentals)*
- D *Machine learning tools and techniques*
- E *Resources*
- F *Glossary*
- G *Setting up your AWS GPU*
- H *Locality sensitive hashing*

acknowledgments

Assembling this book while simultaneously building the software to make it "live" would not have been possible without a supportive network of talented developers. These contributors came from a vibrant Portland community sustained by organizations like PDX Python, Hack Oregon, Hack University, PDX Data Science, Hopester, PyDX, PyLadies, and Total Good. Developers like Zachary Kent built our first Twitter chatbot and helped us test and expand it as the book progressed from grammar-based information extraction to semantic processing. Santi Adavani implemented named entity recognition using the Stanford CoreNLP library and helped us develop explanations for SVD and LSI. Eric Miller sacrificed some of Squishy Media's limited resources to bootstrap Hobson's web dev and visualization skills before he even knew what D3 and vector space models were. Erik Larson and Aleck Landgraf generously gave Hannes and Hobson leeway to experiment with machine learning and NLP even when their startup was on the ropes. Riley Rustad created the Django scheduling app used by the Twitter bot to promote PyCon openspaces events. Anna Ossowski helped design the Openspaces Twitter bot and then shepherded it through its early days of learning to tweet responsibly. And Chick Wells cofounded Total Good and developed the concept of an IQ Test for chatbots, administering this test on a variety of common consumer dialog engines. Dan Fellin helped start it all with his teaching and bot development for the NLP Tutorial session at PyCon 2016. Catherine Nikolovski shared her resources and "hacker" community to help create the software and material used in this book. Rachel Kelly gave us the exposure and support we needed during the early stages of material development. Thunder Shiviah provided constant inspiration through his tireless teaching and boundless enthusiasm. Molly Murphy and Natasha Pettit are responsible for giving us a cause, a focus, inspiring the concept of a mediating, prosocial chatbot that contributes to the greater good.

P1

Wordy machines

Part 1 kicks off your natural language processing (NLP) adventure with an introduction to some real-world applications.

In chapter 1, you'll quickly begin to think of ways you can use machines that process words in your own life. And hopefully you'll get a sense for the magic, the power of machines that can glean information from the words in a natural language document. Words are the foundation of any language, whether it's the keywords in a programming language or the natural language words you learned as a child.

In chapter 2, we give you the tools you need to teach machines to extract words from documents. There's more to it than you might guess, and we show you all the tricks. You'll learn how to automatically group natural language words together into groups of words that share similar spelling, and hopefully similar meaning.

In chapter 3, we count those words and assemble them into vectors that represent the meaning of a document. You can use these vectors to represent the meaning of an entire document, whether it's a 140-character tweet or a 500-page novel.

In chapter 4, you'll discover some time-tested math tricks to compress your vectors down to much more useful topic vectors.

By the end of part 1, you'll have the tools you need for many interesting NLP applications—from semantic search to chatbots. :leveloffset: +1

Packets of thought (NLP overview)



This chapter covers

- What natural language processing (NLP) is
- Why NLP is hard and only recently has become widespread
- When word order and grammar is important and when it can be ignored
- How a chatbot combines many of the tools of NLP
- How to use a regular expression to build the start of a tiny chatbot

You are about to embark on an exciting adventure in natural language processing (NLP). First we show you what NLP is and all the things you can do with it. This will get your wheels turning, helping you think of ways to use NLP in your own life both at work and at home.

Then we dig into the details of exactly how to process a small bit of English text using a programming language like Python, which will help you build up your NLP toolbox incrementally. In this chapter you'll write your first program that can read and write English statements. This Python snippet will be the first of many you'll use to learn all the tricks needed to assemble an English language dialog engine—a chatbot.

1.1 Natural language vs. programming language

Natural languages are different from computer programming languages. They aren't intended to be translated into a finite set of mathematical operations, like programming languages are. Natural languages are what humans use to share information with each other. We don't use programming languages to tell each other about our day or to give directions to the grocery store. A computer program written with a programming language tells a machine exactly what to do. But there are no compilers or interpreters for natural languages such as English and French.

DEFINITION **Natural language processing**

Natural language processing is an area of research in computer science and artificial intelligence (AI) concerned with processing natural languages such as English or Mandarin. This processing generally involves translating natural language into data (numbers) that a computer can use to learn about the world. And this understanding of the world is sometimes used to generate natural language text that reflects that understanding.

Nonetheless, this chapter shows you how a machine can *process* natural language. You might even think of this as a natural language interpreter, just like the Python interpreter. When the computer program you develop processes natural language, it will be able to act on those statements or even reply to them. But these actions and replies are not precisely defined, which leaves more discretion up to you, the developer of the natural language pipeline.

DEFINITION **Pipeline**

A natural language processing system is often referred to as a "pipeline" because it usually involves several stages of processing where natural language flows in one end and the processed output flows out the other.

You'll soon have the power to write software that does interesting, unpredictable things, like carry on a conversation, which can make machines seem a bit more human. It may seem a bit like magic—at first, all advanced technology does. But we pull back the curtain so you can explore backstage, and you'll soon discover all the props and tools you need to do the magic tricks yourself.

"Everything is easy, once you know the answer."

-- Dave Magee Georgia Tech 1995

1.2 The magic

What's so magical about a machine that can read and write in a natural language? Machines have been processing languages since computers were invented. However, these "formal" languages—such as early languages Ada, COBOL, and Fortran—were designed to be interpreted (or compiled) only one correct way. Today Wikipedia lists more than 700 programming languages. In contrast, *Ethnologue*¹ has identified ten times as many natural languages spoken by humans around the world. And Google's index of natural language documents is well over 100 million gigabytes.² And that's just the index. And it's incomplete. The size of the actual natural language content currently online must exceed 100 billion gigabytes.³ But this massive amount of natural language text isn't the only reason it's important to build software that can process it.

Footnote 1 *Ethnologue* is a web-based publication that maintains statistics about natural languages.

Footnote 2 www.google.com/search/howsearchworks/crawling-indexing/

Footnote 3 You can estimate the amount of actual natural language text out there to be at least 1000 times the size of Google's index.

The interesting thing about the process is that it's hard. Machines with the capability of processing something natural isn't natural . It's kind of like building a building that can do something useful with architectural diagrams. When software can process languages not designed for machines to understand, it seems magical—something we thought was a uniquely human capability.

The word "natural" in "natural language" is used in the same sense that it is used in "natural world". Natural, evolved things in the world about us are different from mechanical, artificial things designed and built by humans. Being able to design and build software that can read and process language like what you're reading here—language about building software that can process natural language... well that's very meta, very magical.

To make your job a little easier, we focus on only one natural language, English. But you can use the techniques you learn in this book to build software that can process any language, even a language you don't understand, or has yet to be deciphered by archaeologists and linguists. And we're going to show you how write software to process and generate that language using only one programming language, Python.

Python was designed from the ground up to be a readable language. It also exposes a lot of its own language processing "guts." Both of these characteristics make it a natural

choice for learning natural language processing. It's a great language for building maintainable production pipelines for NLP algorithms in an enterprise environment, with many contributors to a single codebase. We even use Python in lieu of the "universal language" of mathematics and mathematical symbols, wherever possible. After all, Python is an unambiguous way to express mathematical algorithms,⁴ and it's designed to be as readable as possible for programmers like you.

Footnote 4 Mathematical notation is ambiguous: en.wikipedia.org/wiki/Ambiguity#Mathematical_notation

1.2.1 Machines that converse

Natural languages can't be directly translated into a precise set of mathematical operations, but they do contain information and instructions that can be extracted. Those pieces of information and instruction can be stored, indexed, searched, or immediately acted upon. One of those actions could be to generate a sequence of words in response to a statement. This is the function of the "dialog engine" or chatbot that you'll build.

We focus entirely on English text documents and messages, not spoken statements. We bypass the conversion of spoken statements into text—speech recognition, or speech to text (STT). We also ignore speech generation or text to speech, converting text back into some human-sounding voice utterance. But you can still use what you learn to build a voice interface or virtual assistant like Siri or Alexa, because speech-to-text and text-to-speech libraries are freely available. Android and iOS mobile operating systems provide high quality speech recognition and generation APIs, and there are Python packages to accomplish similar functionality on a laptop or server.

SIDE BAR
Speech recognition systems

If you want to build a customized speech recognition or generation system, that undertaking is a whole book in itself; we leave that as an "exercise for the reader." It requires a lot of high quality labeled data, voice recordings annotated with their phonetic spellings, and natural language transcriptions aligned with the audio files. Some of the algorithms you learn in this book might help, but most of the algorithms are quite different.

1.2.2 The math

Processing natural language to extract useful information can be difficult. It requires tedious statistical bookkeeping, but that's what machines are for. And like many other technical problems, solving it is a lot easier once you know the answer. Machines still cannot perform most practical NLP tasks, such as conversation and reading comprehension, as accurately and reliably as humans. So you might be able to tweak the algorithms you learn in this book to do some NLP tasks a bit better.

The techniques you'll learn, however, are powerful enough to create machines that can surpass humans in both accuracy and speed for some surprisingly subtle tasks. For example, you might not have guessed that recognizing sarcasm in an isolated Twitter message can be done more accurately by a machine than by a human.⁵ [Python code](#) and [webapp](#) by Matthew Cliche at Cornell achieves similar accuracy >70%] Don't worry, humans are still better at recognizing humor and sarcasm within an ongoing dialog, due to our ability to maintain information about the context of a statement. But machines are getting better and better at maintaining context. And this book helps you incorporate context (metadata) into your NLP pipeline if you want to try your hand at advancing the state of the art.

Footnote 5 Gonzalo-Ibanez et al found that educated and trained human judges could not match the performance of their simple classification algorithm of 68% reported in their ACM paper

Once you extract structured numerical data, vectors, from natural language, you can take advantage of all the tools of mathematics and machine learning. We use the same linear algebra tricks as the projection of 3D objects onto a 2D computer screen, something that computers and drafters were doing long before natural language processing came into its own. These breakthrough ideas opened up a world of "semantic" analysis, allowing computers to interpret and store the "meaning" of statements rather than just word or character counts. Semantic analysis, along with statistics, can help resolve the ambiguity of natural language—the fact that words or phrases often have multiple meanings or interpretations.

So extracting information isn't at all like building a programming language compiler (fortunately for you). The most promising techniques bypass the rigid rules of regular grammars (patterns) or formal languages. You can rely on statistical relationships between words instead of a deep system of logical rules.⁶ Imagine if you had to define English grammar and spelling rules in a nested tree of if...then statements. Could you ever write enough rules to deal with every possible way that words, letters, and

punctuation can be combined to make a statement? Would you even begin to capture the semantics, the meaning of English statements? Even if it were useful for some kinds of statements, imagine how limited and brittle this software would be. Unanticipated spelling or punctuation would break or befuddle your algorithm.

Footnote 6 Some grammar rules can be implemented in a computer science abstraction called a finite state machine. Regular grammars can be implemented in regular expressions. There are two Python packages for running regular expression finite state machines, `re` which is built in, and `regex` which must be installed, but may soon replace `re`. Finite state machines are just trees of if...then...else statements for each token (character/word/n-gram) or action that a machine needs to react to or generate.

Natural languages have an additional "decoding" challenge that is even harder to solve. Speakers and writers of natural languages assume that a human is the one doing the processing (listening or reading), not a machine. So when I say "good morning", I assume that you have some knowledge about what makes up a morning, including not only that mornings come before noons and afternoons and evenings but also after midnights. And you need to know they can represent times of day as well as general experiences of a period of time. The interpreter is assumed to know that "good morning" is a common greeting that doesn't contain much information at all about the morning. Rather it reflects the state of mind of the speaker and her readiness to speak with others.

This theory of mind about the human processor of language turns out to be a powerful assumption. It allows us to say a lot with few words if we assume that the "processor" has access to a lifetime of common sense knowledge about the world. This degree of compression is still out of reach for machines. There is no clear "theory of mind" you can point to in an NLP pipeline. However, we show you techniques in later chapters to help machines build ontologies, or knowledge bases, of common sense knowledge to help interpret statements that rely on this knowledge.

1.3 Practical applications

Natural language processing is everywhere. It's so ubiquitous that some of the examples in table 1 may surprise you.

Table 1.1 Categorized NLP applications

| | | | |
|---------------------|-----------------------------|-----------------------|-------------------------|
| Search | Web | Documents | Autocomplete |
| Editing | Spelling | Grammar | Style |
| Dialog | Chatbot | Assistant | Scheduling |
| Writing | Index | Concordance | Table of contents |
| Email | Spam filter | Classification | Prioritization |
| Text mining | Summarization | Knowledge extraction | Medical diagnoses |
| Law | Legal inference | Precedent search | Subpoena classification |
| News | Event detection | Fact checking | Headline composition |
| Attribution | Plagiarism detection | Literary forensics | Style coaching |
| Sentiment analysis | Community morale monitoring | Product review triage | Customer care |
| Behavior prediction | Finance | Election forecasting | Marketing |
| Creative writing | Movie scripts | Poetry | Song lyrics |

A search engine can provide more meaningful results if it indexes web pages or document archives in a way that takes into account the meaning of natural language text. Autocomplete uses NLP to complete your thought and is common among search engines and mobile phone keyboards. Many word processors, browser plugins, and text editors have spelling correctors, grammar checkers, concordance composers, and most recently, style coaches. Some dialog engines (chatbots) use natural language search to find a response to their conversation partner's message.

NLP pipelines that generate (compose) text can be used not only to compose short replies in chatbots and virtual assistants, but also to assemble much longer passages of text. The Associated Press uses NLP "robot journalists" to write entire financial news articles and sporting event reports.⁷ Bots can compose weather forecasts that sound a lot like what your hometown weather person might say, perhaps because human meteorologists use word processors with NLP features to draft scripts.

Footnote 7 "AP's 'robot journalists' are writing their own stories now", The Verge, Jan 29, 2015, www.theverge.com/2015/1/29/7939067/ap-journalism-automation-robots-financial-reporting

NLP spam filters in early email programs helped email overtake telephone and fax communication channels in the '90s. And the spam filters have retained their edge in the cat and mouse game between spam filters and spam generators for email, but may be losing in other environments like social networks. An estimated 20% of the tweets about the 2016 US presidential election were composed by chatbots.⁸ These bots amplify their owners' and developers' viewpoints with the resources and motivation to influence popular opinion. And these "puppet masters" tend to be foreign governments or large corporations.

Footnote 8 New York Times, Oct 18, 2016,
www.nytimes.com/2016/11/18/technology/automated-pro-trump-bots-overwhelmed-pro-clinton-messages-researchers-say.html
 and MIT Technology Review, Nov 2016,
www.technologyreview.com/s/602817/how-the-bot-y-politic-influenced-this-election/

NLP systems can generate more than just short social network posts. NLP can be used to compose lengthy movie and product reviews on Amazon and elsewhere. Many reviews are the creation of autonomous NLP pipelines that have never set foot in a movie theater or purchased the product they are reviewing.

There are chatbots on Slack, IRC, and even customer service websites—places where chatbots have to deal with ambiguous commands or questions. And chatbots paired with voice recognition and generation systems can even handle lengthy conversations with an indefinite goal or "objective function" such as making a reservation at a local restaurant.⁹ NLP systems can answer phones for companies that want something better than a phone tree but don't want to pay humans to help their customers.

Footnote 9 Google Blog May 2018 about their *Duplex* system
ai.googleblog.com/2018/05/advances-in-semantic-textual-similarity.html

NOTE

With its *Duplex* demonstration at Google IO, engineers and managers overlooked concerns about the ethics of teaching chatbots to deceive humans. We all ignore this dilemma when we happily interact with chatbots on Twitter and other anonymous social networks, where bots do not share their pedigree. With bots that can so convincingly deceive us, the AI control problem¹⁰ looms, and Yuval Harari's cautionary forecast of "Homo Deus"¹¹ may come sooner than we think.

Footnote 10 en.wikipedia.org/wiki/AI_control_problem

Footnote 11 WSJ Blog, March 10, 2017

blogs.wsj.com/cio/2017/03/10/homo-deus-author-yuval-noah-harari-says-authority-shifting-from-people

NLP systems exist that can act as email "receptionists" for businesses or executive assistants for managers. These assistants schedule meetings and record summary details in an electronic Rolodex, or CRM (customer relationship management system), interacting with others by email on their boss's behalf. Companies are putting their brand and face in the hands of NLP systems, allowing bots to execute marketing and messaging campaigns. And some inexperienced daredevil NLP textbook authors are letting bots author several sentences in their book. More on that later.

1.4 Language through a computer's "eyes"

When you type "Good Morn'n Rosa", a computer sees only "01000111 01101111 01101111 ...". How can you program a chatbot to respond to this binary stream intelligently? Could a nested tree of conditionals (`if... else...`" statements) check each one of those bits and act on them individually? This would be equivalent to writing a special kind of program called a finite state machine (FSM). An FSM that outputs a sequence of new symbols as it runs, like the Python `str.translate` function, is called a finite state transducer (FST). You've probably already built a FSM without even knowing it. Have you ever written a regular expression? That's the kind of FSM we use in the next section to show you one possible approach to NLP: the pattern-based approach.

What if you decided to search a memory bank (database) for the exact same string of bits, characters, or words, and use one of the responses that other humans and authors have used for that statement in the past? But imagine if there was a typo or variation in the statement. Our bot would be sent off the rails. And bits aren't continuous or forgiving—they either match or they don't. There's no obvious way to find similarity between two streams of bits that takes into account what they signify. The bits for "good" will be just as similar to "bad!" as they are to "okay".

But let's see how this approach would work before we show you a better way. Let's build a small regular expression to recognize greetings like "Good morning Rosa" and respond appropriately—our first tiny chatbot!

1.4.1 The language of locks (regular expressions)

Surprisingly the humble combination lock is actually a simple language processing machine. After finishing this chapter, you'll never think of your combination bicycle lock the same way again. A combination lock certainly can't read and understand the textbooks stored inside a school locker, but it can understand the language of lock combinations. It can understand when you try to "tell" it a "password": a combination. A padlock combination is any sequence of symbols that matches the "grammar" (pattern) of lock language. Even more importantly, it can tell if a combination lock "statement" matches a particularly meaningful statement, the one for which the right "answer" in lock language is to release the catch holding the U-shaped hasp so you can get into your locker.

And this lock language is a particularly simple one, which we can use in a chatbot. We

can use it to recognize a key phrase or command to unlock a particular action or behavior. For example, we'd like our chatbot to recognize greetings like "Hello Rosa," and respond to them appropriately. This kind of language, like the language of locks, is a formal language because it has strict rules about how an acceptable statement must be composed and interpreted. Formal languages are a subset of natural languages, so many statements in natural language can be captured by a formal language grammar. That's the reason for this diversion into the mechanical, "click, whirr"¹² language of combination locks.

Footnote 12 One of Cialdini's six psychology principles in his popular book *Influence*
changingminds.org/techniques/general/cialdini/click-whirr.htm

We use a slightly more restrictive grammar than formal grammar, called a regular grammar, which is particularly easy to work with. This small restriction in our language unlocks a broad set of powerful, easy-of-use features. Regular grammars have predictable, provable behavior, and yet are flexible enough to power some of the most sophisticated dialog engines and chatbots on the market. Amazon Alexa and Google Now are mostly pattern-based engines that rely on regular grammars. Deep, complex regular grammar rules can often be expressed in a single line of code called a regular expression. There are successful chatbot frameworks in Python, like `will`, that rely exclusively on this kind of language to produce some useful and interesting behavior. Amazon Echo, Google Home, and similarly complex and useful assistants use this kind of language to encode the logic for most of their user interaction.

NOTE

Regular expressions implemented in Python and in Posix (Unix) applications such as `grep` are not true regular grammars. They have language and logic features such as look-ahead and look-back that make leaps of logic and recursion that aren't allowed in a regular grammar. As a result, regular expressions aren't provably halting; they can sometimes "crash" or run forever.

You may be saying to yourself, "I've heard of regular expressions. I use `grep`. But that's only for search!" And you're right. *Regular Expressions* are indeed used mostly for search, for sequence matching. But anything that can find matches within text is also great for carrying out a dialog. Some chatbots, like `will`, use "search" to find sequences of characters within a user statement that they know how to respond to. These recognized sequences then trigger a scripted response appropriate to that particular regular

expression match. And that match can also be used to extract a useful piece of information from a statement so that a chatbot can add that bit of knowledge to its knowledge base about the user or about the world the user is describing.

A machine that processes this kind of language can be thought of as formal mathematical object called a finite state machine or deterministic finite automaton (DFA). FSMs come up again and again in this book. So you'll eventually get a good feel for what they're used for without digging into FSM theory and math. For now, think of them as combination locks. For those who can't resist trying to understand a bit more about these computer science tools, figure 1.1 shows where FSMs fit into the nested world of automata (bots) and the size of the formal languages that each kind of automata can handle.

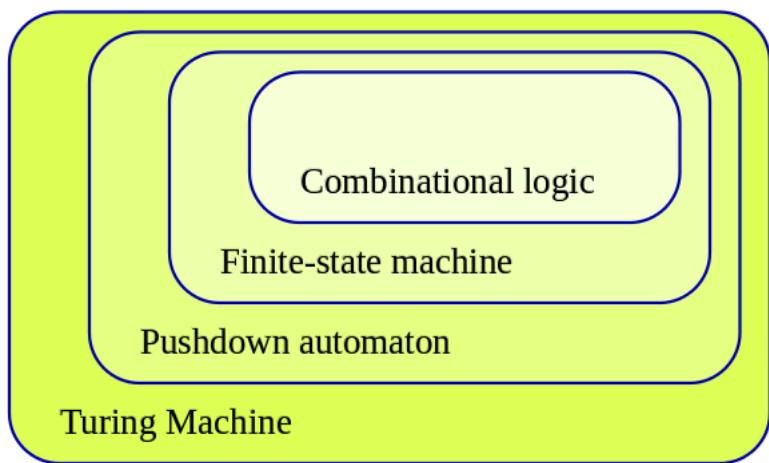


Figure 1.1 Kinds of automata

1.4.2 A simple chatbot

Let's build a quick and dirty chatbot. It won't be very capable, and it will require a lot of thinking about the English language and hardcoding of all the ways people may try to say something. But don't worry if you think you couldn't have come up with this Python code yourself. You won't have to try to think of all the different ways people can say something, like we did in this example. You won't even have to write regular expressions (regexes). We show you how to build a chatbot of your own in later chapters without hardcoding anything. A modern chatbot can learn from reading (processing) a bunch of English text. And we show you how to do that in later chapters. This is an example of an explicit, controlled approach that was common before modern chatbot techniques were developed. And a variation of the approach we show you here is what is behind chatbots like Amazon Alexa (though not the more sophisticated assistants out there).

For now let's build a FSM, a regular expression, that can speak lock language (regular language). We could program it to understand lock language statements, such as "01-02-03." Even better, we'd like it to understand greetings, things like "open sesame" or "hello Rosa." An important feature for a prosocial chatbot is to be able to respond to a greeting. In high school, teachers often chastised me for being impolite when I'd ignore greetings like this while rushing to class. We surely don't want that for our kind and benevolent chatbot.

In machine communication protocol, we'd define a simple handshake with an ACK (acknowledgement) signal after each message passed back and forth between two machines. But our machines are going to be interacting with humans who say things like "Good morning, Rosa". We don't want it sending out of bunch of chirps, beeps, or ACK messages, like it's syncing up a modem or HTTP connection at the start of a conversation or web browsing session. Instead let's use regular expressions to recognize several different human greetings at the start of a conversation handshake.

```
>>> import re ①
>>> r = "(hi|hello|hey)[ ]*([a-z]*)" ②
>>> re.match(r, 'Hello Rosa', flags=re.IGNORECASE) ③
<_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'>
>>> re.match(r, "hi ho, hi ho, it's off to work ...", flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 5), match='hi ho'>
>>> re.match(r, "hey, what's up", flags=re.IGNORECASE)
<_sre.SRE_Match object; span=(0, 3), match='hey'>
```

- ➊ There are two "official" regular expression packages in Python. We use the `re` package here just because it is installed with all versions of Python. The `regex` package comes with later versions of Python and is much more powerful, as you'll see in Chapter 2.
- ➋ `|` means "OR", and `*` means the preceding character can occur 0 or more times and still match. So our regex will match greetings that start with "hi" or "hello" or "hey" followed by any number of <space> characters and then any number of letters.
- ➌ Ignoring the case of text characters is common to keep the regular expressions simpler.

In regular expressions, you can specify a character class with square brackets. And you can use a dash (-) to indicate a range of characters without having to type them all out individually. So the regular expression "`[a-z]`" will match any single lowercase letter, "a" through "z". The star (*) after a character class means that the regular expression will match any number of consecutive characters if they are all within that character class.

Let's make our regular expression a lot more detailed to try to match more greetings.

```
>>> r = r"^[a-z]*([y]o|[h']?ello|ok|hey|(good[ ])?(morn[gin']{0,3}|afternoon|even[gin']{0,3}))[\s,::]{1,
>>> re_greeting = re.compile(r, flags=re.IGNORECASE) ①
>>> re_greeting.match('Hello Rosa')
<_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'>
>>> re_greeting.match('Hello Rosa').groups()
('Hello', None, None, 'Rosa')
>>> re_greeting.match("Good morning Rosa")
<_sre.SRE_Match object; span=(0, 17), match="Good morning Rosa">
>>> re_greeting.match("Good Manning Rosa") ②
>>> re_greeting.match('Good evening Rosa Parks').groups() ③
('Good evening', 'Good ', 'evening', 'Rosa')
>>> re_greeting.match("Good Morn'n Rosa")
<_sre.SRE_Match object; span=(0, 16), match="Good Morn'n Rosa">
>>> re_greeting.match("yo Rosa")
<_sre.SRE_Match object; span=(0, 7), match='yo Rosa'>
```

- ① You can compile regular expressions so you don't have to specify the options (flags) each time you use it.
- ② Notice that this regular expression cannot recognize (match) words with typos.
- ③ Our chatbot can separate different parts of the greeting into groups, but it will be unaware of Rosa's famous last name, because we don't have a pattern to match any characters after the first name.

TIP

The `r` before the quote specifies a raw string, not a regular expression. With a Python raw string, you can send backslashes directly to the regular expression compiler without have to double-backslash (`\\\`) all the special regular expression characters such as spaces (`'`) and curly braces or handlebars (`{}``).

There's a lot of logic packed into that first line of code, the regular expression. It gets the job done for a surprising range of greetings. But it missed that "Manning" typo, which is one of the reasons NLP is hard. In machine learning and medical diagnostic testing, that's called a false negative classification error. Unfortunately, it will also match some statements that humans would be unlikely to ever say—a false positive, which is also a bad thing. Having both false positive and false negative errors means that our regular expression is both too liberal and too strict. These mistakes could make our bot sound a bit dull and mechanical. We'd have to do a lot more work to refine the phrases that it matches to be more human-like.

And this tedious work would be highly unlikely to ever succeed at capturing all the slang and misspellings people use. Fortunately, composing regular expressions by hand isn't the only way to train a chatbot. Stay tuned for more on that later (the entire rest of the

book). So we only use them when we need precise control over a chatbot's behavior, such as when issuing commands to a voice assistant on your mobile phone.

But let's go ahead and finish up our one-trick chatbot by adding an output generator. It needs to say something. We use Python's string formatter to create a "template" for our chatbot response.

```
>>> my_names = set(['rosa', 'rose', 'chatty', 'chatbot', 'bot', 'chatterbot'])
>>> curt_names = set(['hal', 'you', 'u'])
>>> greeter_name = ''①
...
>>> match = re_greeting.match(input())
...
>>> if match:
...     at_name = match.groups()[-1]
...     if at_name in curt_names:
...         print("Good one.")
...     elif at_name.lower() in my_names:
...         print("Hi {}, How are you?".format(greeter_name))
```

① We don't yet know who is chatting with the bot, and we won't worry about it here.

So if you run this little script and chat to our bot with a phrase like "Hello Rosa", it will respond by asking about your day. If you use a slightly rude name to address the chatbot, she will be less responsive, but not inflammatory, to try to encourage politeness.¹³ If you name someone else who might be monitoring the conversation on a party line or forum, the bot will keep quiet and allow you and whomever you are addressing to chat. Obviously there's no one else out there watching our `input()` line, but if this were a function within a larger chatbot, you want to deal with these sorts of things.

Footnote 13 The idea for this defusing response originated with Viktor Frankl's *Man's Search for Meaning*, his [Logotherapy](#) approach to psychology and the many popular novels where a child protagonist like Owen Meany has the wisdom to respond to an insult with a response like this.

Because of the limitations of computational resources, early NLP researchers had to use their human brain's computational power to design and hand-tune complex logical rules to extract information from a natural language string. This is called a pattern-based approach to NLP. The patterns don't have to be merely character sequence patterns, like our regular expression. NLP also often involves patterns of word sequences, or parts of speech, or other "higher level" patterns. The core NLP building blocks like stemmers and tokenizers as well as sophisticated end-to-end NLP dialog engines (chatbots) like Liza were built this way, from regular expressions and pattern matching. The art of pattern-matching approaches to NLP is coming up with elegant patterns that capture just what you want, without too many lines of regular expression code.

TIP**Classical computational theory of mind**

This classical NLP pattern-matching approach is based on the computational theory of mind (CTM). CTM assumes that human-like NLP can be accomplished with a finite set of logical rules that are processed in series.¹⁴ Advancements in neuroscience and NLP led to the development of a "connectionist" theory of mind around the turn of the century, which allows for parallel pipelines processing natural language simultaneously, as is done in artificial neural networks.^{15 16}

Footnote 14 Stanford Encyclopedia of Philosophy, Computational Theory of Mind, plato.stanford.edu/entries/computational-mind/

Footnote 15 Stanford Encyclopedia of Philosophy, Connectionism, plato.stanford.edu/entries/connectionism/

Footnote 16 Christiansen and Chater, 1999, Southern Illinois University, crl.ucsd.edu/~elman/Bulgaria/christiansen-chater-soa.pdf

You'll learn more about pattern-based approaches—such as the Porter stemmer or the Treebank tokenizer—to tokenizing and stemming in chapter 2. But in later chapters we take advantage of the exponentially greater computational resources to process much larger, as well as our larger datasets, to shortcut this laborious hand programming and refining.

If you're new to regular expressions and want to learn more, you can check out appendix B or the online documentation for Python regular expressions. But you don't have to understand them just yet. We'll continue to provide you with example regular expressions as we use them for the building blocks of our NLP pipeline. So don't worry if they look like gibberish. Human brains are pretty good at generalizing from a set of examples, and I'm sure it will become clear by the end of this book. And it turns out machines can learn this way as well...

1.4.3 Another way

Is there a statistical or machine learning approach that might work in place of the pattern-based approach? If we had enough data could we do something different? What if we had a giant database containing sessions of dialog between humans, statements and responses for thousands or even millions of conversations? One way to build a chatbot would be to search that database for the exact same string of characters our chatbot user just "said" to our chatbot. Couldn't we then use one of the responses to that statement that other humans have said in the past?

But imagine how a single typo or variation in the statement would trip up our bot. Bit and character sequences are discrete. They either match or they don't. There's no obvious way to find similarity between two streams of bits that takes into account what they signify or mean. The bits and character sequences for "good" will be just as similar to "bad!" as they are to "okay".

When we use character sequence matches to measure distance between natural language phrases, we'll often get it wrong. Phrases with similar meaning, like "good" and "okay", can often have different character sequences and large distances when we count up character-by-character matches to measure distance. And sequences with completely different meanings, like "bad" and "bar", might be too close to one other when we use metrics designed to measure distances between numerical sequences. Metrics like Jaccard, Levenshtein, and Euclidean vector distance can sometimes add enough "fuzziness" to prevent a chatbot from stumbling over minor spelling errors or typos. But these metrics fail to capture the essence of the relationship between two strings of characters when they are dissimilar. And they also sometimes bring small spelling differences close together that might not really be typos, like "bad" and "bar".

Distance metrics designed for numerical sequences and vectors are useful for a few NLP applications, like spelling correctors and Popper name recognition. So we use these distance metrics when they make sense. But for NLP applications where we are more interested in the meaning of the natural language than its spelling, there are better approaches. We use vector representations of natural language words and text and some distance metrics for those vectors for those NLP applications. We show you each approach, one by one, as we talk about these different applications and the kinds of vectors they are used with.

We don't stay in this confusing binary world of logic for long, but let's imagine we're famous World War II-era code-breaker Mavis Batey at Bletchley Park and we've just

been handed that binary, Morse code message intercepted from communication between two German military officers. It could hold the key to winning the war. Where would we start? Well the first layer of deciding would be to do something statistical with that stream of bits to see if we can find patterns. We can first use the Morse code table (or ASCII table, in our case) to assign letters to each group of bits. Then, if the characters are gibberish to us, as they are to a computer or a cryptographer in WWII, we could start counting them up, looking up the short sequences in a dictionary of all the words we've seen before and putting a mark next to the entry every time it occurs. We might also make a mark in some other log book to indicate which message the word occurred in, creating an encyclopedic index to all the documents we've read before. This collection of documents is called a *corpus*, and the words or sequences we've listed in our index are called a *lexicon*.

If we're lucky, and we're not at war, and the messages we're looking at aren't strongly encrypted, we'll see patterns in those German word counts that mirror counts of English words used to communicate similar kinds of messages. Unlike a cryptographer trying to decipher German Morse code intercepts, we know that the symbols have consistent meaning and aren't changed with every key click to try to confuse us. This tedious counting of characters and words is just the sort of thing a computer can do without thinking. And surprisingly, it's nearly enough to make the machine appear to understand our language. It can even do math on these statistical vectors that coincides with our human understanding of those phrases and words. When we show you how to teach a machine our language using Word2Vec in later chapters, it may seem magical, but it's not. It's just math, computation.

But let's think for a moment about what information has been lost in our effort to count all the words in the messages we receive. We assign the words to bins and store them away as bit vectors like a coin or token sorter (see figure 1.2) directing different kinds of tokens to one side or the other in a cascade of decisions that piles them in bins at the bottom. Our sorting machine must take into account hundreds of thousands if not millions of possible token "denominations," one for each possible word that a speaker or author might use. Each phrase or sentence or document we feed into our token sorting machine will come out the bottom, where we have a "vector" with a count of the tokens in each slot. Most of our counts are zero, even for large documents with verbose vocabulary. But we haven't lost any words yet. What have we lost? Could you, as a human understand a document that we presented you in this way, as a count of each

possible word in your language, without any sequence or order associated with those words? I doubt it. But if it was a short sentence or tweet, you'd probably be able to rearrange them into their intended order and meaning most of the time.



Figure 1.2 Canadian coin sorter

Here's how our token sorter fits into an NLP pipeline right after a tokenizer (see chapter 2). We've included a stopword filter as well as a "rare" word filter in our mechanical token sorter sketch. Strings flow in from the top, and bag-of-word vectors are created from the height profile of the token "stacks" at the bottom.

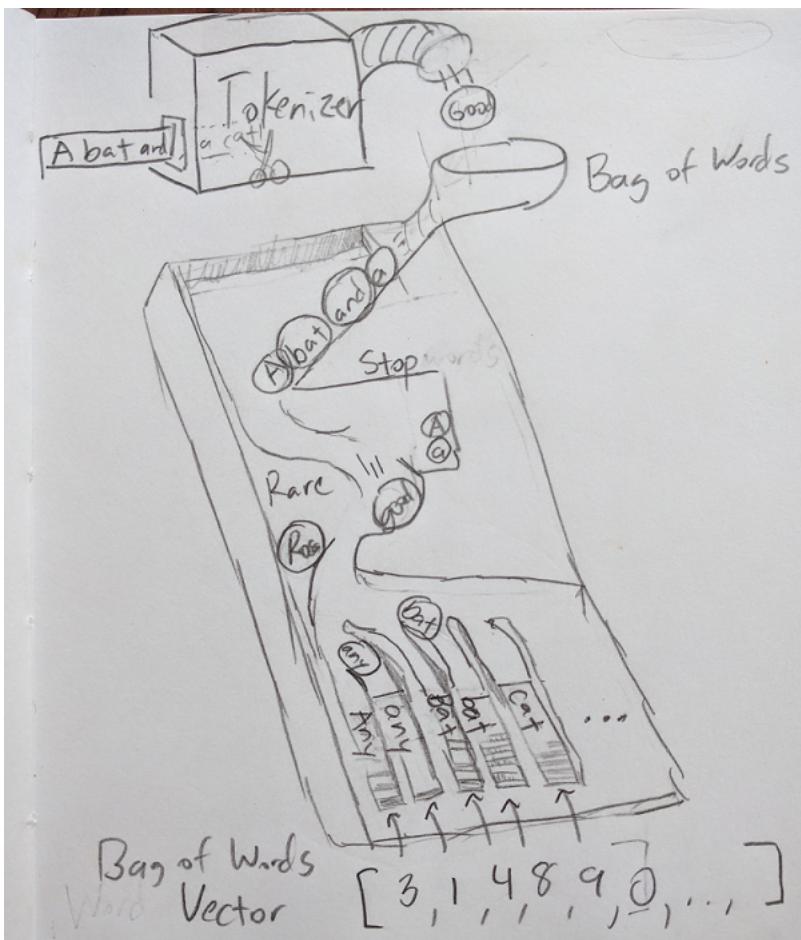


Figure 1.3 Token sorting tray

It turns out that machines can handle this bag of words quite well and glean most of the information content of even moderately long documents this way. Each document, after token sorting and counting, can be represented as a vector, a sequence of integers for each word or token in that document. You see a crude example in figure 1.3, and then chapter 2 shows some more useful data structures for bag-of-word vectors.

This is our first vector space model of a language. Those bins and the numbers they contain for each word are represented as long vectors containing a lot of zeros and a few ones or twos scattered around wherever the word for that bin occurred. All the different ways that words could be combined to create these vectors is called a *vector space*. And relationships between vectors in this space are what make up our model, which is attempting to predict combinations of these words occurring within a collection of various sequences of words (typically sentences or documents). In Python, we can represent these sparse (mostly empty) vectors (lists of numbers) as dictionaries. And a Python Counter is a special kind of dictionary that bins objects (including strings) and counts them just like we want.

```
>>> from collections import Counter
>>> Counter("Guten Morgen Rosa".split())
Counter({'Guten': 1, 'Rosa': 1, 'morgen': 1})
>>> Counter("Good morning, Rosa!".split())
Counter({'Good': 1, 'Rosa!': 1, 'morning,' : 1})
```

You can probably imagine some ways to clean those tokens up. We do just that in the next chapter. But you might also think to yourself that these sparse, high-dimensional vectors (many bins, one for each possible word) aren't very useful for language processing. But they are good enough for some industry-changing tools like spam filters, which we discuss in chapter 3.

And we can imagine feeding into this machine, one at a time, all the documents, statements, sentences, and even single words we could find. We'd count up the tokens in each slot at the bottom after each of these statements was processed, and we'd call that a vector representation of that statement. All the possible vectors a machine might create this way is called a *vector space*. And this model of documents and statements and words is called a *vector space model*. It allows us to use linear algebra to manipulate these vectors and compute things like distances and statistics about natural language statements, which helps us solve a much wider range of problems with less human programming and less brittleness in the NLP pipeline. One statistical question that is asked of bag-of-words vector sequences is "What is the combination of words most likely to follow a particular bag of words." Or, even better, if a user enters a sequence of words, "What is the closest bag of words in our database to a bag-of-words vector provided by the user?" This is a search query. The input words are the words you might type into a search box, and the closest bag-of-words vector corresponds to the document or web page you were looking for. The ability to efficiently answer these two questions would be sufficient to build a machine learning chatbot that could get better and better as we gave it more and more data.

But wait a minute, perhaps these vectors aren't like any you've ever worked with before. They're extremely high-dimensional. It's possible to have millions of dimensions for a 3-gram vocabulary computed from a large corpus. In chapter 3, we discuss the curse of dimensionality and some other properties that make high dimensional vectors difficult to work with.

1.5 A brief overflight of hyperspace

In chapter 3, we show you how to consolidate words into a smaller number of vector dimensions to help mitigate the curse of dimensionality and maybe turn it to our advantage. When we project these vectors onto each other to determine the distance between pairs of vectors, this will be a reasonable estimate of the similarity in their *meaning* rather than merely their statistical word usage. This vector distance metric is called *cosine distance metric*, which we talk about in chapter 3 and then reveal its true power on reduced dimension topic vectors in chapter 4. We can even project ("embed" is the more precise term) these vectors in a 2D plane to have a "look" at them in plots and diagrams to see if our human brains can find patterns. We can then teach a computer to recognize and act on these patterns in ways that reflect the underlying meaning of the words that produced those vectors.

Imagine all the possible tweets or messages or sentences that humans might write. Even though we do repeat ourselves a lot, that's still a lot of possibilities. And when those tokens are each treated as separate, distinct dimensions, there's no concept that "Good morning, Hobs" has some shared meaning with "Guten Morgen, Hannes." We need to create some reduced dimension vector space model of messages so we can label them with a set of continuous (float) values. We could rate messages and words for qualities like subject matter and sentiment. We could ask questions like:

- How likely is this message to be a question?
- How much is it about a person?
- How much is it about me?
- How angry or happy does it sound?
- Is it something I need to respond to?

Think of all the ratings we could give statements. We could put these ratings in order and "compute" them for each statement to compile a "vector" for each statement. The list of ratings or dimensions we could give a set of statements should be much smaller than the number of possible statements, and statements that mean the same thing should have similar values for all our questions.

These rating vectors become something that a machine can be programmed to react to. We can simplify and generalize vectors further by clumping (clustering) statements together, making them close on some dimensions and not on others.

But how can a computer assign values to each of these vector dimensions? Well, if we simplified our vector dimension questions to things like "does it contain the word

'good'?" Does it contain the word "morning?" And so on. You can see that we might be able to come up with a million or so questions resulting in numerical value assignments that a computer could make to a phrase. This is the first practical vector space model, called a bit vector language model, or the sum of "one-hot encoded" vectors. You can see why computers are just now getting powerful enough to make sense of natural language. The millions of million-dimensional vectors that humans might generate simply "Does not compute!" on a supercomputer of the 80s, but is no problem on a commodity laptop in the 21st century. More than just raw hardware power and capacity made NLP practical; incremental, constant-RAM, linear algebra algorithms were the final piece of the puzzle that allowed machines to crack the code of natural language.

There's an even simpler, but much larger representation that can be used in a chatbot. What if our vector dimensions completely described the exact sequence of characters. It would contain the answer to questions like, "Is the first letter an A? Is it a B? ... Is the second letter an A?" and so on. This vector has the advantage that it retains all the information contained in the original text, including the order of the characters and words. Imagine a player piano that could only play a single note at a time, and it had 52 or more possible notes it could play. The "notes" for this natural language mechanical player piano are the 26 uppercase and lowercase letters plus any punctuation that the piano must know how to "play." The paper roll wouldn't have to be much wider than for a real player piano and the number of notes in some long piano songs doesn't exceed the number of characters in a small document. But this one-hot character sequence encoding representation is mainly useful for recording and then replaying an exact piece rather than composing something new or extracting the essence of a piece. We can't easily compare the piano paper roll for one song to that of another. And this representation is longer than the original ASCII-encoded representation of the document. The number of possible document representations just exploded in order to retain information about each sequence of characters. We retained the order of characters and words but expanded the dimensionality of our NLP problem.

These representations of documents don't cluster together well in this character-based vector world. The Russian mathematician Vladimir Levenshtein came up with a brilliant approach for quickly finding similarities between vectors (strings of characters) in this world. Levenshtein's algorithm made it possible to create some surprisingly fun and useful chatbots, with only this simplistic, mechanical view of language. But the real magic happened when we figured out how to compress/embed these higher dimensional spaces into a lower dimensional space of fuzzy meaning or topic vectors. We peek behind the magician's curtain in chapter 4 when we talk about latent semantic indexing

and latent Dirichlet allocation, two techniques for creating much more dense and meaningful vector representations of statements and documents.

1.6 Word order and grammar

The order of words matters. Those rules that govern word order in a sequence of words (like a sentence) are called the grammar of a language. That's something that our bag of words or word vector discarded in the earlier examples. Fortunately, in most short phrases and even many complete sentences, this word vector approximation works OK. If you just want to encode the general sense and sentiment of a short sentence, word order is not terribly important. Take a look at all these orderings of our "Good morning Rosa" example.

```
>>> from itertools import permutations

>>> [ " ".join(combo) for combo in permutations("Good morning Rosa!".split(), 3)]
['Good morning Rosa!',
 'Good Rosa! morning',
 'morning Good Rosa!',
 'morning Rosa! Good',
 'Rosa! Good morning',
 'Rosa! morning Good']
```

Now if you tried to interpret each of those strings in isolation (without looking at the others), you'd probably conclude that they all probably had similar intent or meaning. You might even notice the capitalization of the word "Good" and place the word at the front of the phrase in your mind. But you might also think that "Good Rosa" was some sort of proper noun, like the name of a restaurant or flower shop. Nonetheless, a smart chatbot or clever woman of the 1940s in Bletchley Park would likely respond to any of these six permutations with the same innocuous greeting, "Good morning my dear General."

Let's try that (in our heads) on a much longer, more complex phrase, a logical statement where the order of the words matters a lot:

```
>>> s = "Find textbooks with titles containing 'NLP', or 'natural' and 'language', or 'computational' and 'data science'. These must be consecutive words in the title." 
>>> len(set(s.split()))
12
>>> import numpy as np
>>> np.arange(1, 12 + 1).prod() # factorial(12) = arange(1, 13).prod()
479001600
```

The number of permutations exploded from `factorial(3) == 6` in our simple greeting to `factorial(12) == 479001600` in our longer statement! And it's clear that the logic contained in the order of the words is important to any machine that would like to reply

with the correct response. Even though common greetings are not usually garbled by bag-of-words processing, more complex statements can lose most of their meaning when thrown into a bag. A bag of words is not the best way to begin processing a database query, like the natural language query in the preceding example. Whether a statement is written in a formal programming language like SQL, or in an informal natural language like English, word order and grammar are important when a statement intends to convey precise, logical relationships between things. That's why computer languages depend on rigid grammar and syntax rule parsers. Fortunately, recent advances in natural language syntax tree parsers have made possible the extraction of syntactical and logical relationships from natural language with remarkable accuracy (greater than 90%).¹⁷ In later chapters, we show you how to use packages like `SyntaxNet` (Parsey McParseface) and `SpaCy` to identify these relationships.

Footnote 17 A comparison of the syntax parsing accuracy of `SpaCy` (93%), `SyntaxNet` (94%), Stanford's `CoreNLP` (90%), and others is available at spacy.io/docs/api/

And just as in the Bletchley Park example greeting, even if a statement doesn't rely on word order for logical interpretation, sometimes paying attention to that word order can reveal subtle hints of meaning that might facilitate deeper responses. These deeper layers of natural language processing are discussed in the next section. And chapter 2 shows you a trick for incorporating some of the information conveyed by word order into our word-vector representation. It also shows you how to refine the crude tokenizer used in the previous examples (`str.split()`) to more accurately bin words into more appropriate slots within the word vector, so that strings like "good" and "Good" are assigned the same bin, and separate bins can be allocated for tokens like "rosa" and "Rosa" but not "Rosa!".

1.7 A chatbot natural language pipeline

The NLP pipeline required to build a dialog engine, or chatbot, is similar to the pipeline required to build a question answering system described in *Taming Text* (Manning, 2013).¹⁸ However, some of the algorithms listed within the five subsystem blocks may be new to you. We help you implement these in Python to accomplish various NLP tasks essential for most applications, including chatbots.

Footnote 18 Ingersol, Morton, and Farris, www.manning.com/books/taming-text/?a_aid=totalgood

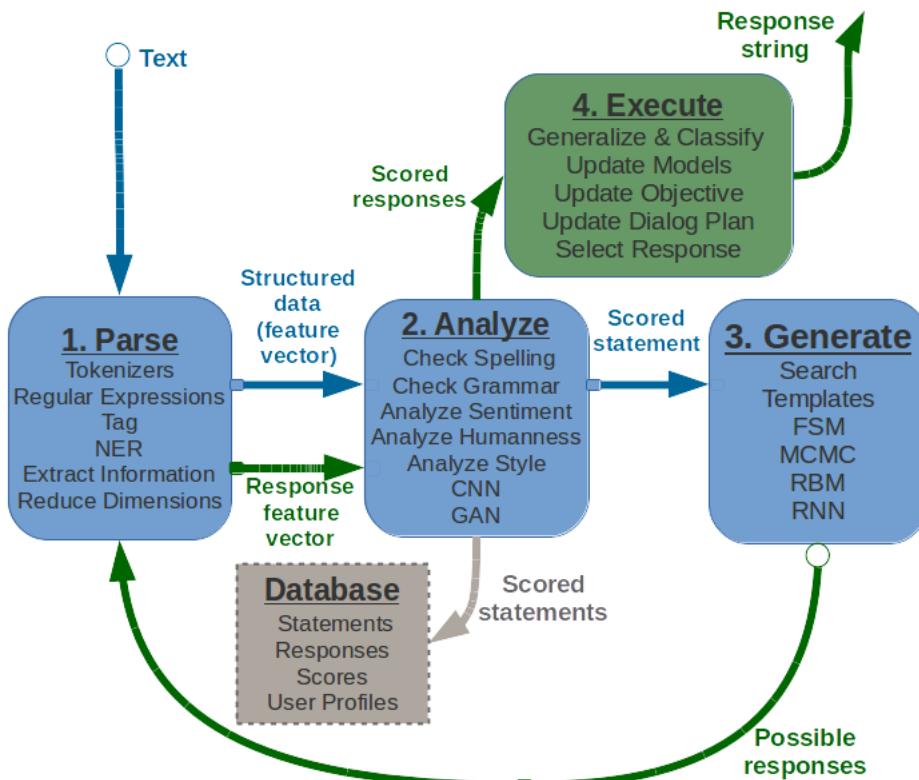


Figure 1.4 Chatbot recirculating (recurrent) pipeline

A chatbot requires four kinds of processing as well as a database to maintain a memory of past statements and responses. Each of the four processing stages can contain one or more processing algorithms working in parallel or in series (see figure 1.4).

1. *Parse*—Extract features, structured numerical data, from natural language text.
2. *Analyze*—Generate and combine features by scoring text for sentiment, grammaticality, semantics.
3. *Generate*—Compose possible responses using templates, search, or language models.
4. *Execute*—Plan statements based on conversation history and objectives, and select the next response.

Each of these four stages can be implemented using one or more of the algorithms listed within the corresponding boxes in the block diagram. We show you how to use Python to accomplish near state-of-the-art performance for each of these processing steps. And we show you several alternative approaches to implementing these five subsystems.

Most chatbots will contain elements of all five of these subsystems (the four processing stages as well as the database). But many applications require only simple algorithms for many of these steps. Some chatbots are better at answering factual questions, and others are better at generating lengthy, complex, convincingly human responses. Each of these capabilities require different approaches; we show you techniques for both.

In addition, deep learning and data-driven programming (machine learning, or probabilistic language modeling) have rapidly diversified the possible applications for NLP and chatbots. This data-driven approach allows ever greater sophistication for an NLP pipeline by providing it with greater and greater amounts of data in the domain you want to apply it to. And when a new machine learning approach is discovered that makes even better use of this data, with more efficient model generalization or regularization, then large jumps in capability are possible.

The NLP pipeline for a chatbot shown in figure 1.4 contains all the building blocks for most of the NLP applications that we described at the start of this chapter. As in *Taming Text*, we break out our pipeline into four main subsystems or stages. In addition we've explicitly called out a database to record data required for each of these stages and persist their configuration and training sets over time. This can enable batch or online retraining of each of the stages as the chatbot interacts with the world. In addition we've shown a "feedback loop" on our generated text responses so that our responses can be processed using the same algorithms used to process the user statements. The response "scores" or features can then be combined in an objective function to evaluate and select the best possible response, depending on the chatbot's plan or goals for the dialog. This book is focused on configuring this NLP pipeline for a chatbot, but you may also be able to see the analogy to the NLP problem of text retrieval or "search," perhaps the most common NLP application. And our chatbot pipeline is certainly appropriate for the question answering application that was the focus of *Taming Text*.

The application of this pipeline to financial forecasting or business analytics may not be so obvious. But if your analysis subsystem (stage 2) is trained to generate features, scores, that are designed to be useful for your particular finance or business predictions, they can help you incorporate natural language data into a machine learning pipeline for forecasting. Despite focusing on building a chatbot, this book gives you the tools to build a pipeline useful for a broad range of NLP applications, from search to forecasting.

One processing element in figure 1.4 that is not typically employed in search, forecasting, or question answering systems is natural language *generation*. For chatbots this is their central feature. Nonetheless, the text generation step is often incorporated into a search engine NLP application and can give such an engine a large competitive advantage. The ability to consolidate or summarize search results is a winning feature for many popular search engines (DuckDuckGo, Bing, and Google). And you can imagine how valuable it

is for a financial forecasting engine to be able to generate statements, tweets, or entire articles based on the business-actionable events it detects in natural language streams from social media networks and news feeds.

The next section shows how the layers of such a system can be combined to create greater sophistication and capability at each stage of the NLP pipeline.

1.8 Processing in depth

The stages of a natural language processing pipeline can be thought of as layers, like the layers in a feed-forward neural network. Deep learning is all about creating more complex models and behavior by adding additional processing layers to the conventional two-layer machine learning model architecture of feature extraction followed by modeling. In chapter 5 we explain how neural networks help spread the learning across layers by backpropagating model errors from the output layers back to the input layers. But here we talk about the top layers and what can be done by training each layer independently of the other layers.

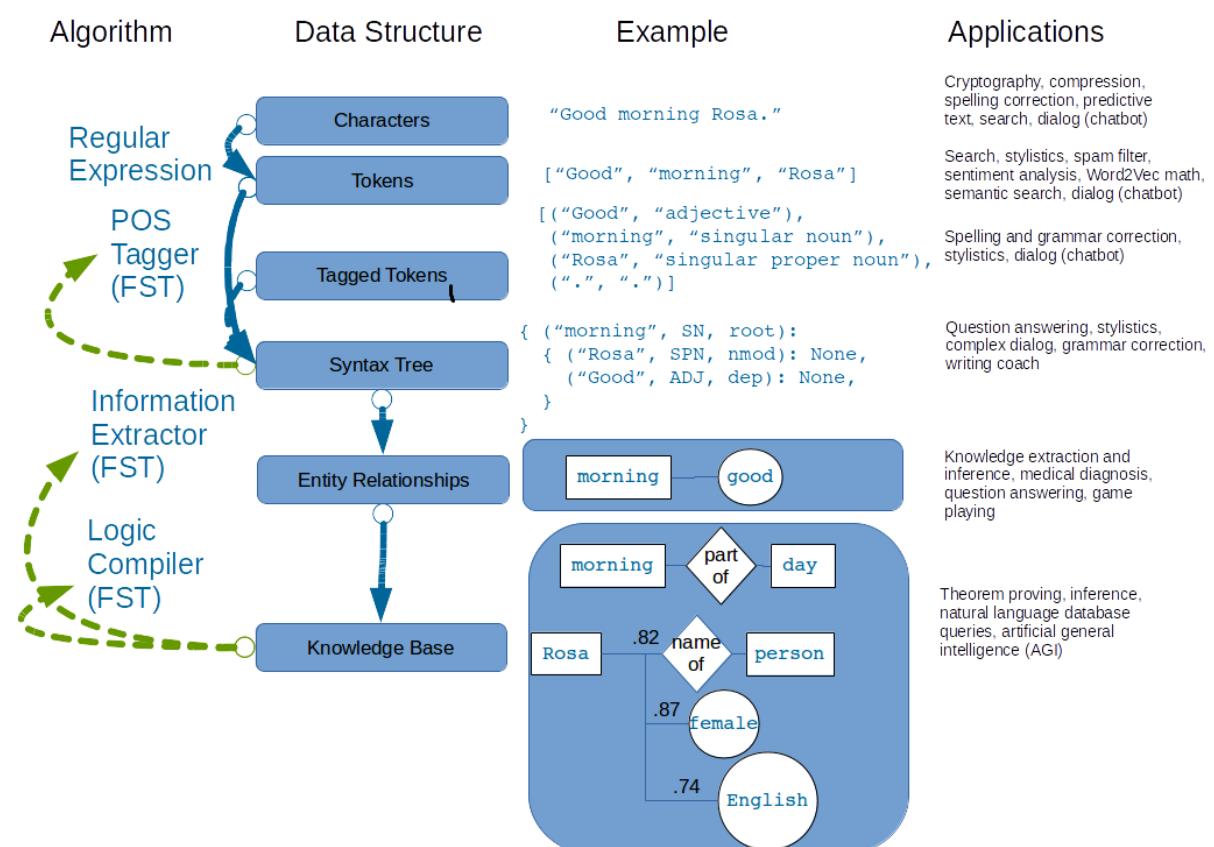


Figure 1.5 nlp-layers

The top four layers in figure 1.5 correspond to the first two stages in the chatbot pipeline (feature extraction and feature analysis) in the previous section. For example the

part-of-speech tagging (POS tagging), is one way to generate features within the Analyze stage of our chatbot pipeline. POS tags are generated automatically by the default `spacy` pipeline, which includes all the top four layers in this diagram. POS tagging is typically accomplished with a finite state transducer like the methods in the `nltk.tag` package.

The bottom two layers (Entity Relationships and a Knowledge Base) are used to populate a database containing information (knowledge) about a particular domain. And the information extracted from a particular statement or document using all six of these layers can then be used in combination with that database to make inferences. Inferences are logical extrapolations from a set of conditions detected in the environment, like the logic contained in the statement of a chatbot user. This kind of "inference engine" in the deeper layers of this diagram are considered the domain of artificial intelligence, where machines can make inferences about their world and use those inferences to make logical decisions. However, chatbots can make reasonable decisions without this knowledge database, using only the algorithms of the upper few layers. And these decisions can combine to produce surprisingly human-like behaviors.

Over the next few chapters, we dive down through the top few layers of NLP. The top three layers are all that is required to perform meaningful sentiment analysis and semantic search, and to build human-mimicking chatbots. In fact, it's possible to build a useful and interesting chatbot using only single layer of processing, using the text (character sequences) directly as the features for a language model. A chatbot that only does string matching and search is capable of participating in a reasonably convincing conversation, if given enough example statements and responses.

For example, the open source project `ChatterBot` simplifies this pipeline by merely computing the string "edit distance" (Levenshtein distance) between an input statement and the statements recorded in its database. If its database of statement-response pairs contains a matching statement, the corresponding reply (from a previously "learned" human or machine dialog) can be reused as the reply to the latest user statement. For this pipeline, all that is required is step 3 (Generate) of our chatbot pipeline. And within this stage, only a brute force search algorithm is required to find the best response. With this simple technique (no tokenization or feature generation required), `ChatterBot` can maintain a convincing conversion as the dialog engine for `Salvius`, a mechanical robot built from salvaged parts by Gunther Cox.¹⁹

Footnote 19 ChatterBot by Gunther Cox and others at github.com/gunthercox/ChatterBot

`will` is an open source Python chatbot framework by Steven Skoczen with a completely

different approach.²⁰ `will` can only be trained to respond to statements by programming it with regular expressions. This is the labor-intensive and data-light approach to NLP. This grammar-based approach is especially effective for question answering systems and task-execution assistant bots, like Lex, Siri, and Google Now. These kinds of systems overcome the "brittleness" of regular expressions by employing "fuzzy regular expressions"²¹ and other techniques for finding approximate grammar matches. Fuzzy regular expressions find the closest grammar matches among a list of possible grammar rules (regular expressions) instead of exact matches by ignoring some maximum number of insertion, deletion, and substitution errors. However, expanding the breadth and complexity of behaviors for a grammar-based chatbot requires a lot of human development work. Even the most advanced grammar-based chatbots, built and maintained by some of the largest corporations on the planet (Google, Amazon, Apple, Microsoft), remain in the middle of the pack for depth and breadth of chatbot IQ.

Footnote 20 `Will`, a chatbot for HipChat by Steven Skoczen and the HipChat community
github.com/skoczen/will

Footnote 21 The Python `regex` package is backward compatible with `re` and adds fuzziness among other features. It will replace the `re` in the future: pypi.python.org/pypi/regex. Similarly TRE `agrep` (approximate grep) is an alternative to the UNIX command-line application `grep`: github.com/laurikari/tre/

A lot of powerful things can be done with shallow NLP. And little, if any, human supervision (labeling or curating of text) is required. Often a machine can be left to learn perpetually from its environment (the stream of words it can pull from Twitter or some other source).²²

Footnote 22 Restricted Boltzmann machines are often the model of choice in recent research into this sort of unsupervised feature extraction or embedding of character sequences. Neural nets are more commonly used for token or word sequence embeddings and language models. We visit these and other unsupervised models in a chapter on natural language embedding.

1.9 Natural language IQ

Like human brainpower, the power of an NLP pipeline cannot be easily gauged with a single IQ score without considering multiple "smarts" dimensions. A common way to measure the capability of a robotic system is along the dimensions of complexity of behavior and degree of human supervision required. But for a natural language processing pipeline, the goal is to build systems that fully automate the processing of natural language, eliminating all human supervision (once the model is trained and deployed). So a better pair of IQ dimensions should capture the breadth and depth of the complexity of the natural language pipeline.

A consumer product chatbot or virtual assistant like Alexa or Allo is usually designed to have extremely broad knowledge and capabilities. However, the logic used to respond to requests tends to be shallow, often consisting of a set of trigger phrases that all produce the same response with a single if-then decision branch. Alexa (and the underlying Lex engine) behave like a single layer, flat tree of (if, elif, elif, ...) statements. On the other hand, the Google Translate pipeline (or any similar machine translation system) relies on a deep tree of feature extractors, decision trees, and deep knowledge graphs connecting bits of knowledge about the world. Sometimes these feature extractors, decision trees, and knowledge graphs are explicitly programmed into the system, as in figure 1.5. Another approach rapidly overtaking this "hand-coded" pipeline is the deep learning data-driven approach, where these layers tend to be an emergent, self-organizing property of large neural networks.

You will use both approaches as we incrementally build an NLP pipeline for a focused but deep chatbot dialog engine. This will give you the skills you need to accomplish natural language processing in your vertical or application area. Along the way you'll probably get ideas about how to expand the breadth of things this NLP pipeline can do. Figure 1.6 puts our chatbot in its place among the natural language processing systems that are already out there. Imagine the chatbots you have interacted with. Where do you think they might fit on a plot like this? Have you attempted to gauge their intelligence by probing them with difficult questions or something like an IQ test? You'll get a chance to do exactly that in later chapters, to help you decide how your chatbot stacks up against some of the others in this diagram.

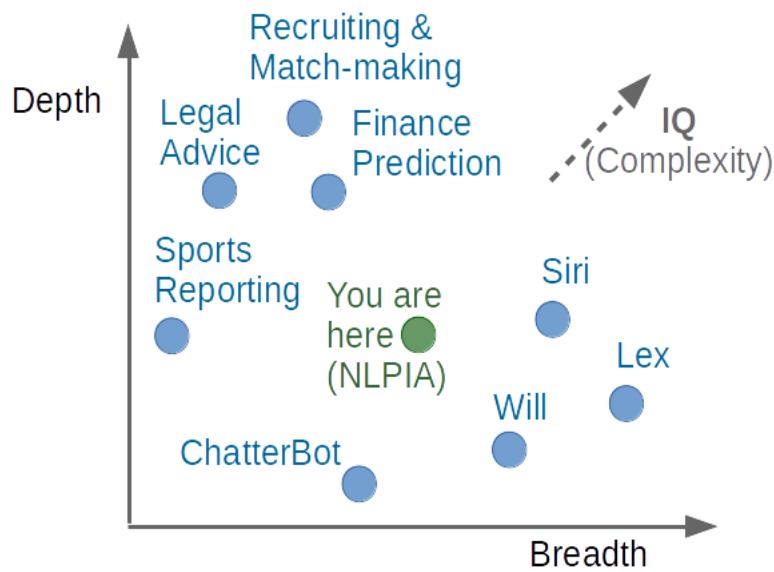


Figure 1.6 Existing natural language processing systems

As you progress through this book, you'll be building the elements of a chatbot. Chatbots

require all the tools of NLP to work well:

- Feature extraction to produce a vector space model
- Information extraction to be able to answer factual questions
- Semantic search to help our chatbot learn from previous human dialog
- Natural language generation to compose new, meaningful statements

Machine learning gives us a way to trick machines into behaving as if we'd spent a lifetime programming them with hundreds of complex regular expressions. We can teach a machine to respond to patterns similar to the patterns defined in regular expressions by merely providing it examples of statements and the responses we want to see from the chatbot. And the "models" of language, the FSMs, produced by machine learning, are much better. They are less picky about mispellings and typos.

And machine learning NLP pipelines are easier to "program." We don't have to anticipate every possible use of symbols in our language. We just have to feed the training pipeline with examples of the phrases that match and example phrases that don't match. As long we label them during training, so that the chatbot knows which is which, it will learn to discriminate between them. And there are even machine learning approaches that require little if any "labeled" data.

The rest of this book is about using machine learning to save us from having to anticipate all the ways people can say things in natural language. And each chapter incrementally improves on the basic NLP pipeline for the chatbot introduced in this chapter. In the next chapter, we detect these same greetings and many, many more using machine learning.

1.10 Summary

- Good NLP may help save the world.
- The meaning and intent of words can be deciphered by machines.
- A smart NLP pipeline will be able to deal with ambiguity.
- We can teach machines common sense knowledge without spending a lifetime training them.
- Chatbots can be thought of as semantic search engines.
- Regular expressions are useful for more than just search.

In this chapter we've given you some exciting reasons to learn about natural language processing. You want to help save the world, don't you? And we've attempted to pique your interest with some practical NLP applications that are revolutionizing the way we communicate, learn, do business, and even think. Building an effective chatbot requires an understanding of the important and useful NLP tools and techniques. And the chatbot

depth and breadth of capabilities can be built up incrementally. It won't be long before you're able to build a systems that approaches human-like conversational behavior, as long as you stay within it's "domain" of knowledge. And you should be able to see in upcoming chapters how to train a chatbot with any domain knowledge that interests you—from finance and sports to psychology and literature. If you can find a *corpus* of writing about it, then you can train a machine to understand it. As you are learning the tools of Natural Language Processing you'll be building an NLP pipeline that can help you accomplish your goals in business and in life.

Build your vocabulary (word tokenization)



This chapter covers

- Tokenizing your text into words and n -grams (tokens)
- Building a vector representation of a statement
- Dealing with text contractions and abbreviations
- Tokenizing social media texts, such as tweets from Twitter
- Compressing your token vocabulary with stemming and lemmatization
- Filtering out words with negligible information content (stop words)
- Handling capitalized words appropriately
- Building a sentiment analyzer from scores for tokens

So you're ready to save the world with the power of natural language processing (NLP)? Well the first thing you need is a powerful vocabulary. This chapter will help you split a document, any string, into discrete tokens of meaning. Our tokens are limited to words, punctuation marks, and numbers, but the techniques we use are easily extended to any other units of meaning contained in a sequence of characters, like ASCII emoticons, Unicode emojis, mathematical symbols, and so on.

Retrieving tokens from a document will require some string manipulation beyond just the `str.split()` method employed in chapter 1. You'll need to separate punctuation from words, like quotes at the beginning and end of a statement. And you'll need to split contractions like "we'll" into the words that were combined to form them. Once you've identified the tokens in a document that you'd like to include in your vocabulary, you'll return to the regular expression toolbox to try to combine words with similar meaning in

a process called *stemming*. Then you'll assemble a vector representation of your documents called a bag of words, and you'll try to use this vector to see if it can help you improve upon the greeting recognizer sketched out at the end of chapter 1.

Think for a moment about what a word or token represents to you. Does it represent a single concept, or some blurry cloud of concepts? Could you be sure you could always recognize a word? Are natural language words like programming language keywords that have a precise definitions and set of grammatical usage rules? Could you write software that could recognize a word? Is "ice cream" one word or two to you? Don't both words have entries in your mental dictionary that is separate from the compound word "ice cream"? What about the contraction "don't"? Should that string of characters be split into one or two "packets of meaning"?

And words could be divided even further into smaller packets of meaning. Words have meaningful parts, and even letters themselves carry sentiment and meaning.²³

Footnote 23 Morphemes are parts of words that contain meaning in and of themselves. Geoffrey Hinton and other deep learning deep thinkers have demonstrated that even the smallest part of a word, letters or characters, carry sentiment and meaning.

What about invisible or implied words? Can you think of additional words that are implied by the single-word command "Don't!"? If you can force yourself to think like a machine and then switch back to thinking like a human, you might realize that there are three invisible words in that command. The single statement "Don't!" means "Don't you do that!" or "You, do not do that!" That's three hidden packets of meaning for a total of five tokens you'd like your machine to know about. But don't worry about invisible words for now. All you need for this chapter is a tokenizer that can recognize words that are spelled out. You'll worry about implied words and connotation and even meaning itself in chapter 4 and beyond.

In this chapter, we show you straightforward algorithms for separating a string into words. You'll also extract pairs, triplets, quadruplets, and even quintuplets of tokens. These are called *n*-grams_. Pairs of words are 2-grams; triplets are 3-grams; quadruplets are 4-grams;, and so on. Using *n*-grams enables your machine to know about "ice cream" as well as the "ice" and "cream" that makes it up. Another 2-gram that you'd like to keep together is "Mr. Smith". Your tokens and your vector representation of a document will have a place for "Mr. Smith" along with "Mr." and "Smith," too.

For now, all possible pairs (and short *n*-grams) of words will be included in your vocabulary. But in chapter 3, you'll learn how to estimate the importance of words based

on their document frequency, or how often they occur. That way you can filter out pairs and triplets of words that rarely occur together. You'll find that the approaches we show are not perfect. Feature extraction can rarely retain all the information content of the input data in any machine learning pipeline.

In natural language processing, composing a numerical vector from text is a particularly "lossy" feature extraction process. Nonetheless the bag-of-words (BOW) vectors retain enough of the information content of the text to produce useful and interesting machine learning models. The techniques for sentiment analyzers at the end of this chapter are the exact same techniques Google used to save email from a flood of spam that almost made it useless.

2.1 Challenges (a preview of stemming)

As an example of why feature extraction from text is hard, consider *stemming*—trying to identify words with similar meaning based on the characters they contain. Very smart people spent their careers developing algorithms for grouping similar words together based only on their spelling. Imagine how difficult that is. Imagine trying to remove verb endings like "ing" from "ending" so you'd have a stem called "end" to represent both words. And you'd like to stem the word "running" to "run," so those two words are treated the same. And that's tricky, because you have to remove not only the "ing" but also the extra "n". But you want the word "sing" to stay whole. You wouldn't want to remove the "ing" ending from "sing" or you'd end up with a single-letter "s".

Or imagine trying to discriminate between a pluralizing "s" at the end of a word like "words" and a normal "s" at the end of words like "bus" and "lens". Do isolated individual letters in a word or parts of a word provide any information at all about that word's meaning? Can the letters be misleading? Yes and yes. We show you how to make your NLP pipeline a bit smarter by dealing with these word spelling challenges using conventional stemming approaches. Later, in chapter 5, we show you statistical approaches that only require you to collect text in your preferred natural language. From that corpus, the statistics of word usage will reveal "semantic stems", without any hard-coded regular expressions or stemming rules.

2.2 Building your vocabulary with a tokenizer

In NLP, tokenization is a particular kind of document segmentation. Segmentation breaks up text into smaller chunks or segments, with more focused information content. Segmentation can include breaking a document into paragraphs, paragraphs into sentences, sentences into phrases, or phrases into tokens (usually words) and punctuation. In this chapter, we focus on segmenting text into *tokens*, which is called tokenization.

You may have heard of tokenizers before, if you took a computer science class where you learned about how compilers work. A tokenizer used for compiling computer languages is often called *scanner* or *lexer*. The vocabulary (the set of all the valid tokens) for a computer language is often called a *lexicon*, and that term is still used in academic articles about NLP. If the tokenizer is incorporated into the computer language compiler's parser, the parser is often called a scannerless parser. And because tokens are the end of the line for the context-free grammars (CFG) used to parse computer languages, they are often referred to as terminals in that grammar. We talk about "formal" grammar (as opposed to natural language grammar) later when we talk about regular expressions and finite state machines.

For the fundamental building blocks of NLP, there are equivalents in a computer language compiler:

- *tokenizer*—scanner, lexer, lexical analyzer
- *vocabulary*—lexicon
- *parser*—compiler
- *token, term, word, or n-gram*—token, symbol, or terminal symbol

Tokenization is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline. A tokenizer breaks unstructured data, natural language text, into chunks of information which can be counted as discrete elements. These counts of token occurrences in a document can be used directly as a vector representing that document. This immediately turns an unstructured string (text document) into a numerical data structure suitable for machine learning. These counts can be used directly by a computer to trigger useful actions and responses. Or they might also be used in a machine learning pipeline as features that trigger more complex decisions or behavior. The most common use for bag-of-words vectors created this way is for document retrieval, or search.

The simplest way to tokenize a sentence is to use white space within a string as the "delimiter" of words. In Python, this can be accomplished with the standard library method `split`, which is available on all `str` objects as well as on the `str` built-in type

itself.

```
>>> sentence = "Thomas Jefferson began building Monticello at the age of 26."
>>> sentence.split()
['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26.']
>>> str.split(sentence)
['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26.']
```

Thomas | Jefferson | began | building | Monticello | at | the | age | of | 26.

Figure 2.1 Tokenized phrase

As you can see, this built-in Python method already does a decent job tokenizing a simple sentence. Its only "mistake" was on the last word, where it included the sentence-ending punctuation with the token "26." For now, let's forge ahead with your imperfect tokenizer. You'll deal with punctuation and other challenges later.

With a bit more Python, you can create a numerical vector representation for each word. These vectors are called *one-hot vectors*, and soon you'll see why. A sequence of these one-hot vectors fully captures the original document text in a sequence of vectors, a table of numbers. That will solve the first problem of NLP, turning words into numbers.

```
>>> import numpy as np
1
>>> token_sequence = str.split(sentence)
2
>>> vocab = sorted(set(token_sequence))
3
>>> ', '.join(vocab)
4
'26., Jefferson, Monticello, Thomas, age, at, began, building, of, the'
5
>>> num_tokens = len(token_sequence)
>>> vocab_size = len(vocab)
>>> onehot_vectors = np.zeros((num_tokens, vocab_size), int)
>>> for i, word in enumerate(token_sequence):
...
    onehot_vectors[i, vocab.index(word)] = 1
>>> ', '.join(vocab)
'26. Jefferson Monticello Thomas age at began building of the'
>>> onehot_vectors
array([[0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
       [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
       [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

- ➊ `str.split()` is your quick-and-dirty tokenizer.
- ➋ Your vocabulary lists all the unique tokens (words) that you want to keep track of.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

- ③ Vocabulary is sorted so numbers come before letters, and capital letters come before lowercase letters.
- ④ The empty table is as wide as your count of unique vocabulary terms and as high as the length of your document, 10 rows by 10 columns.
- ⑤ For each word in the sentence, mark the column for that word in your vocabulary with a 1.

So we can see what happened, let's make that look a little nicer using Pandas—a Python package that wraps numpy arrays to create `Series`. Pandas is particularly handy with 2D arrays, arrays of arrays, or tables. It can help keep track of labels for each column, which string token it represents. It can also keep track of labels for each row in an index, if you want. But for now you'll use the default index of integers for the rows in your table of word vectors for this sentence about Thomas Jefferson.

```
>>> import pandas as pd
>>> pd.DataFrame(onehot_vectors, columns=vocab)
   26. Jefferson Monticello Thomas age at began building of the
0      0          0       0   1   0   0     0        0   0   0
1      0          1       0       0   0   0     0        0   0   0
2      0          0       0       0   0   0     1        0   0   0
3      0          0       0       0   0   0     0        1   0   0
4      0          0       1       0   0   0     0        0   0   0
5      0          0       0       0   0   1     0        0   0   0
6      0          0       0       0   0   0     0        0   0   1
7      0          0       0       1   0   0     0        0   0   0
8      0          0       0       0   0   0     0        0   1   0
9      1          0       0       0   0   0     0        0   0   0
```

In this representation of your one-sentence document, each row is a vector for a single word. The sentence has 10 words, all unique, and it doesn't reuse any words. The table has the 10 columns (words in your vocabulary) and 10 rows (words in the document). A "1" in a column indicates a vocabulary word that was present at that position in the document. So if you wanted to know what the third word in a document was, you'd go to the third row in the table. And you'd look up at the column heading for the "1" value in the third row (the row labeled 2, because the row numbers start at 0). At the top of that column, the seventh column in the table, you can find the natural language representation of that word, "began".

Each row of the table is a binary vector, and you can see why it's called a one-hot vector. All the other positions (columns) in a row are 0. Only one column, or position in the vector is "hot" ("1"). A one (1) means on, or hot. A zero (0) mean off, or absent. And you can use the vector [0, 0, 0, 0, 0, 0, 1, 0, 0, 0] to represent the word "began" in your NLP pipeline.

One nice feature of this representation of words and documents is that no information is lost.²⁴ As long as you keep track of which words are indicated by which column, you can reconstruct the original document from this table of one-hot vectors. And this reconstruction process is 100% accurate even though your tokenizer was only 90% accurate at generating the tokens you thought would be useful. As a result, one-hot word vectors like this are typically used in neural nets, sequence-to-sequence language models, and generative language models. They're a good choice for any model or NLP pipeline that needs to retain all the meaning inherent in the original text.

Footnote 24 Except for the distinction between various white spaces that were "split" with your tokenizer. If you wanted to get the original document back, there's no way to tell whether a space or a newline or a tab should be inserted at each position between words. But the information content of whitespace is low, negligible in most English documents.

This one-hot vector table is like a recording of the original text. If you squint hard enough you might be able to imagine that the matrix of ones and zeros above is a player piano paper roll.²⁵ Or maybe it's the bumps on the metal drum of a music box.²⁶ The vocabulary key at the top tells the machine which "note" or word to play for each row in the sequence of words or piano music. Unlike a player-piano, your mechanical word recorder and player is only allowed to use one "finger" at a time. It can only play one "note" or word at a time. It's one-hot. And each note or word is played for the same amount of "time" with a consistent pace. There's no variation in the spacing of the words.

Footnote 25 en.wikipedia.org/wiki/Player_piano

Footnote 26 en.wikipedia.org/wiki/Music_box

But this is just one way of thinking of one-hot word vectors. You can come up with whatever mental model makes sense for you. The important thing is that you've turned a sentence of natural language words into a sequence of numbers, or vectors. Now you can have the computer read and do math on the vectors just like any other vector or list of numbers. This allows your vectors to be input into any natural language processing pipeline that requires this kind of vector.

You could also play a sequence of one-hot encoded vectors back if you want to generate text for a chat bot, just like a player piano might play a song for a less artificial audience. Now all you need to do is figure out how to build a player piano that can "understand" and combine those word vectors in new ways. Ultimately, you'd like your chatbot or NLP pipeline to play us, or say something, you haven't heard before. We get to that in chapters 9 and 10 when we talk about LSTM models, and similar neural networks.

This representation of a sentence in one-hot word vectors retains all the detail, grammar, and order of the original sentence. And you've successfully turned words into numbers that a computer can "understand." They are also a particular kind of number that computers like a lot: binary numbers. But this is a big table for a short sentence. If you think about it, you've expanded the file size that would be required to store your document. For a long document this might not be practical. Your vocabulary size (the length of the vectors) would get huge. The English language contains at least 20,000 common words, millions if you include names and other proper nouns. And your one-hot vector representation requires a new table (matrix) for every document you want to process. This is almost like a raw "image" of your document. If you've done any image processing, you know that you need to do dimension reduction if you want to extract useful information from the data.

Let's run through the math to give you an appreciation for just how big and unwieldy these "player piano paper rolls" are. In most cases, the vocabulary of tokens you'll use in an NLP pipeline will be much more than 10,000 or 20,000 tokens. Sometimes it can be hundreds of thousands or even millions of tokens. Let's assume you have a million tokens in your NLP pipeline vocabulary. And let's say you have a meager 3000 books with 3500 sentences each and 15 words per sentence—reasonable averages for short books. That's a whole lot of big tables (matrices).

```
>>> 3000 * 3500 * 15
157500000 # rows
>>> _ * 1000000
1575000000000 # bytes, if you use one byte for each cell in your table
>>> _ / 1e9
157500 # gigabytes
>>> _ / 1000
157.5 # terabytes
```

You're talking more than a million million bits, even if you use a single bit for each cell in your matrix. At one bit per cell, you'd need more than 10 terabytes of storage for a small bookshelf of books processed this way. Fortunately you don't ever use this for storing documents. You only use it temporarily, in RAM, while you're processing documents one word at a time.

So storing all those zeros, and trying to remember the order of the words in all your documents doesn't make much sense. It's not practical. And what you really want to do is compress the meaning of a document down to its essence. You'd like to compress your

document down to a single vector rather than a big table. And you're willing to give up perfect "recall". You just want to capture most of the meaning (information) in a document, not all of it.

What if you split your documents into much shorter chunks of meaning, say sentences. And what if you assumed that most of the meaning of a sentence can be gleaned from just the words themselves. Let's assume you can ignore the order and grammar of the words, and jumble them all up together into a "bag", one bag for each sentence or short document. That turns out to be a reasonable assumption. Even for documents several pages long, a bag-of-words vector is still useful for summarizing the essence of a document. You can see that for your sentence about Jefferson, even after you sorted all the words lexically, a human can still guess what the sentence was about. So can a machine. You can use this new bag-of-words vector approach to compress the information content for each document into a data structure that's easier to work with.

If you summed all these one-hot vectors together, rather than "replaying" them one at a time, you'd get a bag-of-words vector. This is also called a word frequency vector, because it only counts the "frequency" words, not their order. You could use this single vector to represent the whole document or sentence in a single, reasonable-length vector. It would only be as long as your vocabulary size (the number of unique tokens you want to keep track of).

Alternatively, if you're doing basic keyword search, you could *OR* the one-hot word vectors into a binary bag-of-words vector. And you could ignore a lot of words that wouldn't be interesting as search terms or keywords. This would be fine for a search engine index or the first filter for an information retrieval system. Search indexes only need to know the presence or absence of each word in each document to help you find those documents later.

Just like laying your arm on the piano, hitting all the notes (words) at once, it doesn't make for a pleasant, meaningful experience. Nonetheless this approach turns out to be critical to helping a machine "understand" a whole group of words as a unit. And if you limit your tokens to the 10,000 most important words, you can compress your numerical representation of your imaginary 3500 sentence book down to 10 kilobytes, or about 30 megabytes for your imaginary 3000-book corpus. One-hot vector sequences would require hundreds of gigabytes.

Fortunately, the words in your vocabulary are sparsely utilized in any given text. And for most bag-of-words applications, we keep the documents short, sometimes just a sentence

will do. So rather than hitting all the notes on a piano at once, your BOW vector is more like a broad and pleasant piano chord, a combination of notes (words) that work well together and contain meaning. Your chatbot can handle these chords even if there's a lot of "dissonance" from words in the same statement that aren't normally used together. Even dissonance (odd word usage) is useful information about a statement that a machine learning pipeline can make use of.

Here's how you can put the tokens into a binary vector indicating the presence or absence of a particular word in a particular sentence. This vector representation of a set of sentences could be "indexed" to indicate which words were used in which document. This index is equivalent to the index you find at the end of many textbooks, except that instead of keeping track of which page a word occurs on, you can keep track of the sentence (or the associated vector) where it occurred. Whereas a textbook index generally only cares about important words relevant to the subject of the book, you keep track of every single word (at least for now).

Here's what your single text document, the sentence about Thomas Jefferson, looks like as a binary bag-of-words vector.

```
>>> sentence_bow = {}
>>> for token in sentence.split():
...     sentence_bow[token] = 1
>>> sorted(sentence_bow.items())
[('26.', 1),
 ('Jefferson', 1),
 ('Monticello', 1),
 ('Thomas', 1),
 ('age', 1),
 ('at', 1),
 ('began', 1),
 ('building', 1),
 ('of', 1),
 ('the', 1)]
```

One thing you might notice is that Python's `sorted()` puts decimal numbers before characters, and capitalized words before lowercase words. This is the ordering of characters in the ASCII and Unicode character sets. Capital letters come before lowercase letters in the ASCII table. The order of your vocabulary is unimportant. As long as you are consistent across all the documents you tokenize this way, a machine learning pipeline will work equally well with any vocabulary order.

And you might also notice that using a `dict` (or any paired mapping of words to their 0/1 values) to store a binary vector shouldn't waste much space. Using a dictionary to represent your vector ensures that it only has to store a 1 when any one of the thousands,

or even millions, of possible words in your dictionary appear in a particular document. You can see how it would be much less efficient to represent a bag of words as a continuous list of 0's and 1's with an assigned location in a "dense" vector for each of the words in a vocabulary of say 100,000 words. This dense binary vector representation of your "Thomas Jefferson" sentence would require 100 kB of storage. Because a dictionary "ignores" the absent words, the words labeled with a 0, the dictionary representation only requires a few bytes for each word in your 10-word sentence. And this dictionary could be made even more efficient if you represented each word as an integer pointer to each word's location within your lexicon—the list of words that makes up your vocabulary for a particular application.

So let's use an even more efficient form of a dictionary, a Pandas Series. And you'll wrap that up in a Pandas DataFrame so you can add more sentences to your binary vector "corpus" of texts about Thomas Jefferson. All this hand waving about gaps in the vectors and sparse versus dense bags of words should become clear as you add more sentences and their corresponding bag-of-words vectors to your DataFrame (table of vectors corresponding to texts in a corpus).

```
>>> import pandas as pd
>>> df = pd.DataFrame(pd.Series(dict([(token, 1) for token in sentence.split()])),
    columns=['sent']).T
>>> df
   26. Jefferson Monticello Thomas age at began building of the
sent      1          1          1      1  1    1      1      1  1

```

Let's add a few more texts to your corpus to see how a DataFrame stacks up. A DataFrame indexes both the columns (documents) and rows (words) so it can be an "inverse index" for document retrieval, in case you want to find a Trivial Pursuit answer in a hurry.

```
>>> sentences = "Construction was done mostly by local masons and carpenters.\n" \
...     "He moved into the South Pavilion in 1770.\n" \
...     "Turning Monticello into a neoclassical masterpiece was
Jefferson's obsession."
>>> corpus = {}
>>> corpus['sent0'] = dict((tok.strip('.'), 1) for tok in sentence.split())
>>> for i, sent in enumerate(sentences.split('\n')):
...     corpus['sent{}'.format(i + 1)] = dict((tok, 1) for tok in sent.split())
>>> df = pd.DataFrame.from_records(corpus).fillna(0).astype(int).T
>>> df[df.columns[:7]] # show just the first 7 tokens (columns)
   1770  26. Construction He Jefferson Jefferson's Monticello
sent0      0      1      0      0      1      0      1
sent1      0      0      1      0      0      0      0
sent2      1      0      0      1      0      0      0
sent3      0      0      0      0      0      1      1
```

With a quick scan, you can see little overlap in word usage for these sentences. Among

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

the first seven words in your vocabulary, only the word "Monticello" appears in more than one sentence. Now you need to be able to compute this overlap within your pipeline whenever you want compare documents or search for similar documents. One way to check for the similarities between sentences is to count the number of overlapping tokens using a dot product.

```
>>> df = df.T
>>> df.sent0.dot(df.sent1)
0
>>> df.sent0.dot(df.sent2)
1
>>> df.sent0.dot(df.sent3)
1
```

From this you can tell that one word was used in both `sent0` and `sent2`. Likewise one of the words in your vocabulary was used in both `sent0` and `sent3`. This overlap of words is a measure of their similarity. Interestingly, that oddball sentence, `sent1`, was the only sentence that did not mention Jefferson or Monticello directly, but used a completely different set of words to convey information about other anonymous people. Here's one way to find the word that is shared by `sent0` and `sent3`, the word that gave you that last dot product of 1.

```
>>> [(k, v) for (k, v) in (df.sent0 & df.sent3).items() if v]
[('Monticello', 1)]
```

This is your first vector space model (VSM) of natural language documents (sentences). Not only are dot products possible, but other vector operations are defined for these bag-of-word vectors: addition, subtraction, OR, AND, and so on. You can even compute things such as Euclidean distance or the angle between these vectors. This representation of a document as a binary vector has a lot of power. It was a mainstay for document retrieval and search for many years. All modern CPUs have hardwired memory addressing instructions that can efficiently hash, index, and search a large set of binary vectors like this. Though these instructions were built for another purpose (indexing memory locations to retrieve data from RAM), they are equally efficient at binary vector operations for search and retrieval of text.

2.2.1 A token improvement

In some situations, other characters besides spaces are used to separate words in a sentence. And you still have that pesky period at the end of your "26." token. You need your tokenizer to split a sentence not just on white space, but also on punctuation such as commas, periods, quotes, semicolons, and even hyphens (dashes). In some cases you want these punctuation marks to be treated like words, as independent tokens. In other cases you may want to ignore them.

In the preceding example, the last token in the sentence was corrupted by a period at the end of "26." The trailing period can be misleading for the subsequent sections of an NLP pipeline, like stemming, where you would like to group similar words together using rules that rely on consistent word spellings.

```
>>> import re
>>> sentence = "Thomas Jefferson began building Monticello at the age of 26."
>>> tokens = re.split(r"([-\\s.,;!?]+)", sentence)
>>> list(filter(lambda x: x if x not in '- \t\n.,;!?' else None, tokens))
['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26']
```

This last filter operation using a lambda function could also be accomplished with a list comprehension.

```
>>> [x for x in tokens if x != '' and x not in '- \t\n.,;!?']
['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26']
```

This regular expression splits the sentence on white space or punctuation that occurs at least once (note the "+" in the regular expression). The square brackets ([and]) are used to indicate a character class. This is equivalent to listing each character separated by a pipe character (|) to indicate "OR". This regex splits the sentence on any occurrences of these characters. We promised we'd use more regular expressions. Hopefully they're starting to make a little more sense than they did when we first used them.

TIP**When to compile your regex patterns²⁷**

The regular expression module in Python allows you to precompile regular expressions, which you then can reuse across your code base. For example, you might have a regex that extracts phone numbers. You could use `re.compile()` to precompile the expression and pass it along as an argument to a function or class doing tokenization. This is rarely a speed advantage, because Python caches the compiled objects for the last `MAXCACHE=100` regular expressions. But if you have more than 100 different regular expressions at work, or you want to call methods of the regular expression rather than the corresponding `re` functions, `re.compile` can be useful.

```
>>> pattern = re.compile(r"([-\\s.,;!?]+)")
>>> tokens = pattern.split(sentence)
>>> tokens[-10:] # just the last 10 tokens
['the', ' ', 'age', ' ', 'of', ' ', '26', '.', '']
```

This simple regular expression is helping to split off the period from the end of the token "26". However, you have a new problem. You need to filter the whitespace and punctuation characters you do not want to include in your vocabulary.

```
>>> sentence = "Thomas Jefferson began building Monticello at the age of 26."
>>> tokens = pattern.split(sentence)
>>> [x for x in tokens if x != '' and x not in '- \t\n.,;!?']
['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26']
```

Thomas | Jefferson | began | building | Monticello | at | the | age | of | 26 | .

Figure 2.2 Tokenized phrase

So the built-in Python `re` package seems to do just fine on this example sentence, as long as you are careful to filter out undesirable tokens. There's really no reason to look elsewhere for regular expression packages, except...

TIP**When to use the new `regex` module in Python**

There's a new regular expression package called `regex` that will eventually replace the `re` package. It's completely backward compatible and can be installed with `pip` from pypi. Its useful new features include support for

- Overlapping match sets
- Multithreading
- Feature-complete support for Unicode
- Approximate regular expression matches (similar to TRE's `agrep` on UNIX systems)
- Larger default MAXCACHE (500 regexes)

Even though `regex` will eventually replace the `re` package and is completely backward compatible with `re`, for now you must install it as an additional package using a package manager such as pip.

```
$ pip install regex
```

You can find more information about the `regex` module on the PyPI website (pypi.python.org/pypi/regex).

As you can imagine, tokenizers can easily become complex. In one case, you might want to split based on periods, but only if the period is not followed by a number, in order to avoid splitting decimals. In another case, you might not want to split after a period that is part of "smiley" emoticon symbol, such as in a Twitter message.

Several Python libraries implement tokenizers, each with its own advantages and disadvantages.

- spaCy—Accurate, flexible, fast, Python
- Stanford CoreNLP—More accurate, less flexible, fast, depends on Java 8
- NLTK—Standard used by many NLP contests and comparisons, popular, Python

NLTK and Stanford CoreNLP have been around the longest and are the most widely used for comparison of NLP algorithms in academic papers. Even though the Stanford CoreNLP has a Python API, it relies on the Java 8 CoreNLP backend, which must be installed and configured separately. So you use the Natural Language Toolkit (NLTK) tokenizer here to get you up and running quickly; it will help you duplicate the results you see in academic papers and blog posts.

You can use the NLTK function `RegexpTokenizer` to replicate your simple tokenizer

example like this.

```
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer = RegexpTokenizer(r'\w+|[0-9.]+|\S+')
>>> tokenizer.tokenize(sentence)
['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of',
'26', '.']
```

This regular expression is a bit better than the one you used originally, because it ignores whitespace tokens. It also separates sentence-ending trailing punctuation from tokens that do not contain any other punctuation characters.

An even better tokenizer is the Treebank Word Tokenizer from the NLTK package. It incorporates a variety of common rules for English word tokenization. For example, it separates phrase-terminating punctuation (?!.;,) from adjacent tokens and retains decimal numbers containing a period as a single token. In addition it contains rules for English contractions. For example "don't" is tokenized as ["do", "n't"]. This tokenization will help with subsequent steps in the NLP pipeline, such as the stemming. You can find all rules for the Treebank Tokenizer at www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.treebank.

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> sentence = "Monticello wasn't designated as UNESCO World Heritage Site until 1987."
>>> tokenizer = TreebankWordTokenizer()
>>> tokenizer.tokenize(sentence)
['Monticello', 'was', "n't", 'designated', 'as', 'UNESCO', 'World', 'Heritage', 'Site',
'until', '1987', '.']
```

Monticello | was | n't | designated | as | UNESCO | World | Heritage | Site | until | 1987 | .

Figure 2.3 Tokenized phrase

CONTRACTIONS

You might wonder why you would split the contraction `wasn't` into `was` and `n't`. For some applications, like grammar-based NLP models that use syntax trees, it's important to separate the words `was` and `not` to allow the syntax tree parser to have a consistent, predictable set of tokens with known grammar rules as its input. There are a variety of standard and nonstandard ways to contract words, by reducing contractions to their constituent words, a dependency tree parser or syntax parser only need to be programmed to anticipate the various spellings of individual words rather than all possible contractions.

TIP

How to tokenize informal text from social networks such as Twitter or Facebook

The NLTK library includes a tokenizer that was built to deal with short, informal, emoticon-laced texts from social networks where grammar and spelling conventions vary widely.

You can tokenize Twitter messages with the function `casual_tokenize`. The function allows you to strip usernames or reduce the number of repeated characters to a maximum of three of the same character.

```
>>> from nltk.tokenize.casual import casual_tokenize
>>> message = "RT @TJMonticello Best day everrrrrr at Monticello.
Awesommmmmeeeeeee day :*)"
>>> casual_tokenize(message)
['RT', '@TJMonticello', 'Best', 'day', 'everrrrrr', 'at', 'Monticello', '.', 'Awesommmmmeeeeeee', 'day', ':*)']
>>> casual_tokenize(message, reduce_len=True, strip_handles=True)
['RT', 'Best', 'day', 'everrr', 'at', 'Monticello', '.', 'Awesommeee', 'day', ':*)']
```

2.2.2 Extending your vocabulary with *n*-grams

Let's revisit that "ice cream" problem from the beginning of the chapter. Remember we talked about trying to keep "ice" and "cream" together.

I scream, you scream, we all scream for ice cream.

But I don't know many people that scream for "cream". And nobody screams for "ice", unless they're about to slip and fall on it. So you need a way for your word-vectors to keep "ice" and "cream" together.

WE ALL GRAM FOR *N*-GRAMS

An *n*-gram is a sequence containing up to *n* elements that have been extracted from a sequence of those elements, usually a string. In general the "elements" of an *n*-gram can be characters, syllables, words, or even symbols like "A", "D", and "G" used to represent the chemical amino acid markers in a DNA or RNA sequence.²⁸

Footnote 28 Linguistic and NLP techniques are often used to glean information from DNA and RNA, this site provides a list of amino acid symbols that helps turn amino acid language into a human-readable language: en.wikipedia.org/wiki/Amino_acid#Table_of_standard_amino_acidAbbreviations_and_properties.

In this book, we're only interested in *n*-grams of words, not characters.²⁹ So in this book, when we say 2-gram, we mean a pair of words, like "ice cream". When we say 3-gram, we mean a triplet of words like "beyond the pale" or "Johann Sebastian Bach" or "riddle

me this". *n*-grams don't have to mean something special together, like compound words. They have to be frequent enough together to catch the attention of your token counters.

Footnote 29 You may have learned about trigram indexes in your database class or the documentation for PostgreSQL (`postgres`). But these are triplets of characters. They help you quickly retrieve fuzzy matches for strings in a massive database of strings using the `%` and `~*` SQL full text search queries.

Why bother with *n*-grams? As you saw earlier, when a sequence of tokens is vectorized into a bag-of-words vector, it loses a lot of the meaning inherent in the order of those words. By extending your concept of a token to include multiword tokens, *n*-grams, your NLP pipeline can retain much of the meaning inherent in the order of words in your statements. For example, the meaning-inverting word "not" will remain attached to its neighboring words, where it belongs. Without *n*-gram tokenization, it would be free floating. Its meaning would be associated with the entire sentence or document rather than its neighboring words. The 2-gram "was not" retains much more of the meaning of the individual words "not" and "was" than those 1-grams alone in a bag-of-words vector. A bit of the context of a word is retained when you tie it to its neighbor(s) in your pipeline.

In the next chapter, we show you how to recognize which of these *n*-grams contain the most information relative to the others, which you can use to reduce the number of tokens (*n*-grams) your NLP pipeline has to keep track of. Otherwise it would have to store and maintain a list of every single word sequence it came across. This prioritization of *n*-grams will help it recognize "Thomas Jefferson" and "ice cream", without paying particular attention to "Thomas Smith" or "ice shattered". In chapter 4, we associate word pairs, and even longer sequences, with their actual meaning, independent of the meaning of their individual words. But for now, you need your tokenizer to generate these sequences, these *n*-grams.

Let's use your original sentence about Thomas Jefferson to show what a 2-gram tokenizer should output, so you know what you're trying to build.

```
>>> tokenize_2grams("Thomas Jefferson began building Monticello at the age of 26.")
['Thomas Jefferson',
 'Jefferson began',
 'began building',
 'building Monticello',
 'Monticello at',
 'at the',
 'the age',
 'age of',
 'of 26']
```

I bet you can see how this sequence of 2-grams retains a bit more information than if you'd just tokenized the sentence into words. The later stages of your NLP pipeline will only have access to whatever tokens your tokenizer generates. So you need to let those later stages know that "Thomas" wasn't about "Isaiah Thomas" or the "Thomas & Friends" cartoon. *n*-grams are one of the ways to maintain context information as data passes through your pipeline.

Here's the original 1-gram tokenizer.

```
>>> sentence = "Thomas Jefferson began building Monticello at the age of 26."
>>> pattern = re.compile(r"([-\\s.,;!?]+)")
>>> tokens = pattern.split(sentence)
>>> tokens = [x for x in tokens if x != '' and x not in '- \t\n.,;!?']
['Thomas', 'Jefferson', 'began', 'building', 'Monticello', 'at', 'the', 'age', 'of', '26']
```

And this is the *n*-gram tokenizer from nltk in action.

```
>>> from nltk.util import ngrams
>>> list(ngrams(tokens, 2))
[('Thomas', 'Jefferson'), ('Jefferson', 'began'), ('began', 'building'), ('building', 'Monticello'), ('Monticello', 'at'), ('at', 'the'), ('the', 'age'), ('age', 'of'), ('of', '26'), ('26', '.')]
>>> list(ngrams(tokens, 3))
[('Thomas', 'Jefferson', 'began'), ('Jefferson', 'began', 'building'), ('began', 'building', 'Monticello'), ('building', 'Monticello', 'at'), ('Monticello', 'at', 'the'), ('at', 'the', 'age'), ('the', 'age', 'of'), ('age', 'of', '26'), ('of', '26', '.')]
```

TIP

In order to be more memory efficient, the `ngrams` function of the NLTK library returns a Python generator. Python generators are "smart" functions that behave like iterators, yielding only one element at a time instead of returning the entire sequence at once. This is useful within `for` loops, where the generator will load each individual item instead of loading the whole item list into memory. However, if you want to inspect all the returned *n*-grams at once, convert the generator to a list as you did in the earlier example. Keep in mind that you should only do this in an interactive session, not within a long-running task tokenizing large texts.

The *n*-grams are provided in the previous listing as tuples, but they can easily be joined together if you'd like all the tokens in your pipeline to be strings. This will allow the later stages of the pipeline to expect a consistent datatype as input, string sequences.

```
>>> two_grams = list(ngrams(tokens, 2))
>>> [" ".join(x) for x in two_grams]
['Thomas Jefferson', 'Jefferson began', 'began building', 'building Monticello',
'Monticello at', 'at the', 'the age', 'age of', 'of 26']
```

You might be able to sense a problem here. Looking at your earlier example, you can imagine that the token "Thomas Jefferson" will occur across quite a few documents. However the 2-grams "of 26" or even "Jefferson began" will likely be extremely rare. If tokens or n -grams are extremely rare, they do not carry any correlation with other words that you can use to help identify topics or themes that connect documents or classes of documents. So rare n -grams will not be helpful for classification problems. You can imagine that most 2-grams are pretty rare—even more true for 3- and 4-grams.

Because word combinations are rarer than individual words, your vocabulary size is exponentially approaching the number of n -grams in all the documents in your corpus. If your feature vector dimensionality exceeds the length of all your documents, your feature extraction step is counterproductive. It will be virtually impossible to avoid overfitting a machine learning model to your vectors; your vectors have more dimensions than there are documents in your corpus. In chapter 3, you'll use document frequency statistics to identify n -grams so rare that they are not useful for machine learning. Typically, n -grams are filtered out that occur too infrequently (for example, in three or fewer different documents). This scenario is represented by the "rare token" filter in the coin-sorting machine of chapter 1.

Now consider the opposite problem. Consider the 2-gram "at the" in the previous phrase. That's probably not a rare combination of words. In fact it might be so common, spread among most of your documents, that it loses its utility for discriminating between the meanings of your documents. It has little predictive power. Just like words and other tokens, n -grams are usually filtered out if they occur too often. For example if a token or n -gram occurs in more than 25% of all the documents in your corpus, you usually ignore it. This is equivalent to the "stop words" filter in the coin-sorting machine of chapter 1. These filters are as useful for n -grams as they are for individual tokens. In fact, they are even more useful.

STOP WORDS

Stop words are common words in any language that occur with a high frequency but carry much less substantive information about the meaning of a phrase. Examples of some common stop words include³⁰

Footnote 30 A more comprehensive list of stop words for various languages can be found here:
[raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/stopwords.zip](https://githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/stopwords.zip).

- a, an
- the, this

- and, or
- of, on

Historically stop words have been excluded from NLP pipelines in order to reduce the computational effort to extract information from a text. Even though the words themselves carry little information, the stop words can provide important relational information as part of an n -gram. Consider these two examples:

- Mark reported to the CEO
- Suzanne reported as the CEO to the board

In your NLP pipeline, you might create 4-grams such as `reported to the CEO` and `reported as the CEO`. If you remove the stop words from the 4-grams, both examples would be reduced to `reported CEO`, and you would lack the information about the professional hierarchy. In the first example, Mark could have been an assistant to the CEO, whereas in the second example Suzanne was the CEO reporting to the board. Unfortunately, retaining the stop words within your pipeline creates another problem: It increases the length of the n -grams required to make use of these connections formed by the otherwise meaningless stop words. This issue forces us to retain at least 4-grams if you want to avoid the ambiguity of the human resources example.

Designing a filter for stop words depends on your particular application. Vocabulary size will drive the computational complexity and memory requirements of all subsequent steps in the NLP pipeline. But stop words are only small portion of your total vocabulary size. A typical stop word list has only 100 or so frequent and unimportant words listed in it. But a vocabulary size of 20,000 words would be required to keep track of 95% of the words seen in a large corpus of tweets, blog posts, and news articles.³¹ And that's just for 1-grams or single-word tokens. A 2-gram vocabulary designed to catch 95% of the 2-grams in a large English corpus will generally have more than 1 million unique 2-gram tokens in it.

Footnote 31 rstudio-pubs-static.s3.amazonaws.com/41251_4c55dff8747c4850a7fb26fb9a969c8f.html

You may be worried that vocabulary size drives the required size of any training set you must acquire to avoid overfitting to any particular word or combination of words. And you know that the size of your training set drives the amount of processing required to process it all. However, getting rid of 100 stop words out of 20,000 is not going to significantly speed up your work. And for a 2-gram vocabulary, the savings you'd achieve by removing stop words is minuscule. In addition, for 2-grams you lose a lot more information when you get rid of stop words arbitrarily, without checking for the

frequency of the 2-grams that use those stop words in your text. For example, you might miss mentions of "The Shining" as a unique title and instead treat texts about that violent, disturbing movie the same as you treat documents that mention "Shining Light" or "shoe shining".

So if you have sufficient memory and processing bandwidth to run all the NLP steps in your pipeline on the larger vocabulary, you probably don't want to worry about ignoring a few unimportant words here and there. And if you're worried about overfitting a small training set with a large vocabulary, there are better ways to select your vocabulary or reduce your dimensionality than ignoring stop words. This will enable document frequency filters (discussed in chapter 3) to directly select the words and n -grams with the most information content within your particular domain.

If you do decide to arbitrarily filter out a set of stop words during tokenization, a Python list comprehension is sufficient. Here you take a few stop words and ignore them when you iterate through your token list.

```
>>> stop_words = ['a', 'an', 'the', 'on', 'of', 'off', 'this', 'is']
>>> tokens = ['the', 'house', 'is', 'on', 'fire']
>>> tokens_without_stopwords = [x for x in tokens if x not in stop_words]
>>> print(tokens_without_stopwords)
['house', 'fire']
```

You can see that some words carry a lot more meaning than others. And you can lose more than half the words in some sentences without significantly affecting their meaning. You can often get your point across without articles, prepositions, or even forms of the verb "to be". Imagine someone doing sign language or in a hurry to write a note to themselves. Which words would they chose to always skip? That's how stop words are chosen.

To get a complete list of "canonical" stop words, NLTK is probably the most generally applicable list.

```
>>> import nltk
>>> nltk.download('stopwords')
>>> stopwords = nltk.corpus.stopwords.words('english')
>>> len(stopwords)
153
>>> stopwords[:7]
['i', 'me', 'my', 'myself', 'we', 'our', 'ours']
>>> [sw for sw in stopwords if len(sw) == 1]
['i', 'a', 's', 't', 'd', 'm', 'o', 'y']
```

A document that dwells on the first person is pretty boring, and more importantly for

you, has low information content. The NLTK package includes pronouns (not just first person ones) in its list of stop words. And these one-letter stop words are even more curious, but they make sense if you've used the NLTK tokenizer and Porter stemmer a lot. These single-letter tokens pop up a lot when contractions are split and stemmed.

2.2.3 Normalizing your vocabulary

So you've seen how important vocabulary size is to the performance of an NLP pipeline. Another vocabulary reduction technique is to normalize your vocabulary so that tokens that mean similar things are combined into a single, normalized form. Doing so reduces the number of tokens you need to retain in your vocabulary and also improve the association of meaning across those different "spellings" of a token or *n*-gram in your corpus. And as we mentioned before, reducing your vocabulary can reduce the likelihood of overfitting.

CASE NORMALIZATION

Normalizing word capitalization is one way to reduce your vocabulary by consolidating words that are intended to mean the same thing under a single token. However, some information is often communicated by capitalization of a word—for example, 'doctor' and 'Doctor' often have different meanings. Often, capitalization is used to indicate that a word is a proper noun, the name of a person, place, or thing. You'll want to be able to recognize proper nouns if named entity recognition is important to your pipeline. However, if tokens are not case normalized, your vocabulary may typically be twice as large, consume twice as much memory and processing time, and might increase the amount of training data you need to have labeled for your machine learning pipeline to converge to an accurate, general solution. Just as in any other machine learning pipeline, your labeled data set used for training must be "representative" of the space of all possible feature vectors your model must deal with. For 100000-D bag-of-words vectors, you usually must have 100000 labeled examples, and sometimes even more than that, to train a supervised machine learning pipeline without overfitting. So cutting your vocabulary size by half can sometimes be worth the loss of information content.

In Python, you can easily case normalize your tokens with a list comprehension.

```
>>> tokens = ['House', 'Visitor', 'Center']
>>> normalized_tokens = [x.lower() for x in tokens]
>>> print(normalized_tokens)
['house', 'visitor', 'center']
```

And if you are certain that you want to normalize the case for an entire document, you

can `lower()` the text string before tokenization.

Words can become "denormalized" when they are capitalized because of their presence at the beginning of a sentence, or when they're written in all caps for emphasis. Undoing this denormalization is called *case normalization*.

With case normalization, you are attempting to return these tokens to their "normal" state before grammar rules and their position in a sentence affected their capitalization. The simplest and most common way to normalize the case of a text string is to lowercase all the characters with a function like Python's built-in `str.lower()`. Unfortunately this approach will also "normalize" away a lot of meaningful capitalization as well as the less meaningful first-word-in-sentence capitalization you want to normalize away. A better approach for case normalization is to lowercase only the first word of a sentence and allow all other words to retain their capitalization so that a "Ward Smith" is not confused with "word smith" in your tokenization process. This will continue to introduce capitalization errors for the rare proper nouns at the beginning of a sentence. To avoid this complexity, and potential loss of information, many NLP pipelines do not normalize for case at all. The benefit of reducing one's vocabulary size by about half is outweighed by the loss of information for proper nouns and other capitalized words.

In addition to reducing overfitting for a machine learning pipeline, case normalization is also beneficial for a search engine application. For search, normalization increases the number of matches found for a particular query. For search, without normalization, a query might return a different set of documents if you searched for the word "Age" than if you searched for "age". "Age" would likely occur in phrases like "New Age" or "Age of Reason". In contrast, "age" would be more likely occur in phrases like "at the age of" in your sentence about Thomas Jefferson. By normalizing the vocabulary in your search index (as well as the query), you can ensure that both kinds of documents about "age" are returned regardless of the capitalization in the query from the user. However, this additional recall comes at the cost of precision, returning many documents that the user may not be interested in. Because of this issue, modern search engines allow users to turn off normalization with each query, typically by quoting those words for which they want only exact matches returned. If you are building such a search engine pipeline, in order to accommodate both types of queries you will have to build two indexes for your documents: one with case-normalized *n*-grams, and another with the original capitalization.

STEMMING

Another common vocabulary normalization technique is to eliminate the small meaning differences of pluralization or possessive endings of words, or even various verb forms. This normalization, identifying a common stem among various forms of a word, is called stemming. For example, the words housing and houses share the same stem, house. Stemming removes suffixes from words, in an attempt to combine words with similar meanings together under their common stem. A stem is not required to be a properly spelled word, but merely a token, or label, representing several possible spellings of a word.

A human can easily see that "house" and "houses" are the singular and plural forms of the same noun. However, you need some way to provide this information to the machine. One of its main benefits is in the compression of the number of words whose meaning your software or language model needs to keep track of. It reduces the size of your vocabulary while limiting the loss of information and meaning, as much as possible. In machine learning this is referred to as dimension reduction. It helps generalize your language model, enabling the model to behave identically for all the words included in a stem. So, as long as your application doesn't require your machine to distinguish between "house" and "houses", this stem will reduce your programming or data set size by half or even more, depending on the aggressiveness of the stemmer you chose.

Stemming is important for "search." It allows you to search for "developing houses in Portland" and get web pages that use both the word "house" and "houses" and even the word "housing" because these words are often stemmed to the "hous" token. Likewise you might receive pages with the words "developer" and "development" rather than "developing" because all these words typically reduce to the stem "develop". As you can see, this is a "broadening" of your search, ensuring that you are less likely to miss a relevant document or web page. But in some applications this "false-positive rate" (proportion of the pages returned that you don't find useful) can be a problem. So most search engines allow you to turn off stemming and even case normalization by putting quotes around a word or phrase to indicate that you only want pages containing the exact spelling of the phrase "'Portland Housing Development software'", which will be a different sort of page than one that talks about a "'Portland software developer's house'." And there are times when you want to search for "Dr. House's calls" and not "dr house call", which would likely be the output of many stemmers applied to that query.

Here's a concise stemmer implementation in pure Python that can handle trailing S's.

```
>>> def stem(phrase):
...     return ' '.join([re.findall('^(.*ss|.*?)(s)?$', word)[0][0].strip("") for word
...                     in phrase.lower().split()])
>>> stem('houses')
'house'
>>> stem("Doctor House's calls")
'doctor house call'
```

The stemmer function follows a few simple rules all within that one short regular expression:

- If a word ends with more than one s, the stem is the word and the suffix is a blank string.
- If a word ends with a single s, the stem is the word without the s and the suffix is the s.
- If a word does not end on an s, the stem is the word and no suffix is returned.

The strip method ensures that some possessive words can be stemmed along with plurals.

This function works well for regular cases, but is unable to address more complex cases. For example, the rules would fail with words like `dishes` or `heroes`. For more complex cases like these, the NLTK package provides other stemmers.

It also doesn't handle the "housing" example from your "Portland Housing" search.

Two of the most popular stemming algorithms are the Porter and Snowball stemmers. The former is named after the computer scientist Martin Porter, the algorithm³² follows more complex rules to accommodate more complex cases of English grammar.

Footnote 32 Original publication: www.cs.odu.edu/~jbollen/IR04/readings/readings5.pdf

```
>>> from nltk.stem.porter import PorterStemmer
>>> stemmer = PorterStemmer()
>>> ' '.join([stemmer.stem(w).strip("") for w in "dish washer's washed dishes".split()])
'dish washer wash dish'
```

Notice that the Porter stemmer, like the regular expression stemmer, retains the trailing apostrophe (unless you explicitly strip it) so that possessive words will be distinguishable from nonpossessive words. Possessive words are often proper nouns, so this feature can be important for applications where you want to treat names differently than other nouns.

LEMMATIZATION

If you have access to information about connections between the meanings of various words, you might be able to associate several words together even if their spelling is quite different. This more extensive normalization down to the semantic root of a word—its lemma—is called lemmatization.

In chapter 12, we show how you can use lemmatization to reduce the complexity of the logic required to respond to a statement with a chatbot. Any NLP pipeline that wants to "react" the same for multiple different spellings of the same basic root word can benefit from a lemmatizer. It reduces the number of words you have to respond to, the dimensionality of your language model. Using it can make your model more general, but it can also make your model less precise, because it will treat all spelling variations of a given root word the same. For example "chat", "chatter", "chatty", "chatting", and perhaps even "chatbot" would all be treated the same in an NLP pipeline with lemmatization, even though they have different meanings.

As you work through this section, think about words where lemmatization would drastically alter the meaning of a word, perhaps even inverting its meaning and producing the opposite of the intended response from your pipeline. This scenario is called *spoofing* —when you try to elicit the wrong response from a machine learning pipeline by cleverly constructing a difficult input.

Lemmatization is a more accurate way to normalize a word than stemming or case normalization because it takes into account a word's meaning. Lemmatization identifies words that mean similar things and groups them together so that they can all be treated as the same token or symbol by subsequent stages of the pipeline. Accurate lemmatization of a word requires identification of the word's part of speech (POS) because the POS affects its meaning. The POS tag for a word indicates its role in the grammar of a phrase or sentence. For example, the noun POS is for words that refer to things within the phrase. An adjective is for a word that modifies or describes a noun. A verb refers to an action. The POS of a word in isolation cannot be determined. The context of a word must be known for its POS to be identified.

Can you think of ways you can use the part of speech to identify a better "root" of a word than stemming could? Consider the word `better`. Stemmers would strip the "er" ending from "better" and return the stem "bett" or "bet". However, this would lump the word "better" with words like "betting", "bets", and "Bet's", rather than more similar words like "betterment", "best", or even "good" and "goods".

How can you identify word lemmas in Python? Again, the NLTK package provides functions for this.

```
>>> nltk.download('wordnet')
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize("better")
'better'
```

```
>>> lemmatizer.lemmatize("better", pos="a")
'good'
>>> lemmatizer.lemmatize("good", pos="a")
'good'
>>> lemmatizer.lemmatize("goods", pos="a")
'goods'
>>> lemmatizer.lemmatize("goods", pos="n")
'good'
>>> lemmatizer.lemmatize("goodness", pos="n")
'goodness'
>>> lemmatizer.lemmatize("best", pos="a")
'best'
```

You might be surprised that the first attempt to lemmatize the word "better" didn't change it at all. This is because the part of speech of a word can have a big affect on its meaning. If a POS is not specified for a word, then the NLTK lemmatizer assumes it is a noun. Once you specify the correct POS, 'a' for adjective, the lemmatizer returns the correct lemma. Unfortunately, the NLTK lemmatizer is restricted to the connections within the Princeton WordNet graph of word meanings. So the word "best" does not lemmatize to the same root as "better". This graph is also missing the connection between "goodness" and "good". A Porter stemmer, on the other hand, would make this connection by blindly stripping off the "ness" ending of all words.

```
>>> stemmer.stem('goodness')
'good'
```

USE CASES

When should you use a lemmatizer or a stemmer? Stemmers are generally faster to compute and require less-complex code and data sets. But stemmers will make more errors and stem a far greater number of words, reducing the information content or meaning of your text much more than a lemmatizer would. Both stemmers and lemmatizers will reduce your vocabulary size and increase the ambiguity of the text. But lemmatizers do a better job retaining as much of the information content as possible based on how the word was used within the text and its intended meaning. Therefore, some NLP packages, such as spaCy, don't provide stemming functions and only offer lemmatization methods.

If your application involves search, stemming and lemmatization will improve the recall of your searches by associating more documents with the same query words. However, stemming and lemmatization will reduce the accuracy of your search results by returning many more documents not relevant to the word's original meaning. Because search results can be ranked according to relevance (potentially computed using both stemmed and unstemmed versions of words as well as additional metadata that may resolve

ambiguity), search engines and indexers typically use lemmatization to increase the likelihood that the search results include the documents a user is looking for.

For a search-based chatbot, however, accuracy is more important. As a result, a chatbot should first search for the closest match using unstemmed, unnormalized words before falling back to stemmed or filtered token matches to find similar statements if not enough candidate responses can be found for the initial "exact" search.

2.3 Sentiment

Whether you use raw single-word tokens, *n*-grams, stems, or lemmas in your NLP pipeline, each of those tokens contains some information. An important part of this information is the word's sentiment—the overall feeling or emotion that word invokes. This *sentiment analysis*—measuring the sentiment of phrases or chunks of text—is a common application of NLP. In many companies it's the main thing an NLP engineer is asked to do.

Companies like to know what users think of their products. So they often will provide some way for you to give feedback. A star rating on Amazon or Rotten Tomatoes is one way to get quantitative data about how people feel about products they've purchased. But a more natural way is to use natural language comments. Giving your user a blank slate (an empty text box) to fill up with comments about your product can produce more detailed feedback.

In the past you'd have to read all that feedback. Only a human can understand something like emotion and sentiment in natural language text, right? However, if you had to read thousands of reviews you'd see how tedious and error-prone a human reader can be. Humans are remarkably bad at reading feedback, especially criticism or negative feedback. And customers aren't generally very good at communicating feedback in a way that can get past your natural human triggers and filters.

But machines don't have those biases and emotional triggers. And humans aren't the only things that can process natural language text and extract information, even meaning from it. An NLP pipeline can process a large quantity of user feedback quickly and objectively, with less chance for bias. And an NLP pipeline can output a numerical rating of the positivity or negativity or any other emotional quality of the text.

Another common application of sentiment analysis is junk mail and troll message filtering. You'd like your chatbot to be able to measure the sentiment in the chat messages it processes so it can respond appropriately. And even more importantly, you

want your chatbot to measure its own sentiment of the statements it's about to send out, which you can use to steer your bot to be kind and pro-social with the statements it makes. The simplest way to do this might be to do what Moms told us to do: If you can't say something nice, don't say anything at all. So you need your bot to measure the niceness of everything you're about to say and use that to decide whether to respond.

What kind of pipeline would you create to measure the sentiment of a block of text and produce this sentiment positivity number? Say you just want to measure the positivity or favorability of a text—how much someone likes a product or service that they are writing about. Say you want your NLP pipeline and sentiment analysis algorithm to output a single floating point number between -1 and +1. Your algorithm would output +1 for text with positive sentiment like "Absolutely perfect! Love it! :-) :-)". And your algorithm should output -1 for text with negative sentiment like "Horrible! Completely useless. :(.". Your NLP pipeline could use values near 0, like say +0.1, for a statement like "It was OK. Some good and some bad things".

There are two approaches to sentiment analysis:

- A rule-based algorithm composed by a human
- A *machine learning* model learned from data by a machine

The first approach to sentiment analysis uses human-designed rules, sometimes called heuristics, to measure sentiment. A common rule-based approach to sentiment analysis is to find keywords in the text and map each one to numerical scores or weights in a dictionary or "mapping"—a Python `dict`, for example. Now that you know how to do tokenization, you can use stems, lemmas, or *n*-gram tokens in your dictionary, rather than just words. The "rule" in your algorithm would be to add up these scores for each keyword in a document that you can find in your dictionary of sentiment scores. Of course you need to hand-compose this dictionary of keywords and their sentiment scores before you can run this algorithm on a body of text. We show you how to do this using the VADER algorithm (in `sklearn`) in the upcoming listing.

The second approach, machine learning, relies on a labeled set of statements or documents to train a machine learning model to create those rules. A machine learning sentiment model is trained to process input text and output a numerical value for the sentiment you are trying to measure, like positivity or spamminess or trolliness. For the machine learning approach, you need a lot of data, text labeled with the "right" sentiment score. Twitter feeds are often used for this approach because the hash tags, such as `#awesome` or `#happy` or `#sarcasm`, can often be used to create a "self-labeled" data set. Your company may have product reviews with five-star ratings that you could associate

with reviewer comments. You can use the star ratings as a numerical score for the positivity of each text. We show you shortly how to process a dataset like this and train a token-based machine learning algorithm called *Naive Bayes* to measure the positivity of the sentiment in a set of reviews after you're done with VADER.

2.3.1 VADER—A rule-based sentiment analyzer

Hutto and Gilbert at GA Tech came up with one of the first successful rule-based sentiment analysis algorithms. They called their algorithm VADER, for Valence Aware Dictionary for sEntiment Reasoning.³³ Many NLP packages implement some form of this algorithm. The NLTK package has an implementation of the VADER algorithm in `nltk.sentiment.vader`. Hutto himself maintains the Python package `vaderSentiment`. You'll go straight to the source and use `vaderSentiment` here.

Footnote 33 "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text" by Hutto and Gilbert comp.social.gatech.edu/papers/icwsm14.vader.hutto.pdf

You'll need to `pip install vaderSentiment` to run the following example.³⁴ You have not included it in the `nlpia` package.

Footnote 34 You can find more detailed installation instructions with the package source code on github: github.com/cjhutto/vaderSentiment.

```
>>> from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
>>> sa = SentimentIntensityAnalyzer()
>>> sa.lexicon
①
{
    ...
    ':(' : -1.9,
    ':)': 2.0,
    ...
    'pls': 0.3,
    'plz': 0.3,
    ...
    'great': 3.1,
    ...
}
>>> [(tok, score) for tok, score in sa.lexicon.items() if " " in tok]
④
[("({ }{ )", 1.6),
 ("can't stand", -2.0),
 ('fed up', -1.8),
 ('screwed up', -1.5)]
>>> sa.polarity_scores(text="Python is very readable and it's great for NLP.")
⑤
{'compound': 0.6249, 'neg': 0.0, 'neu': 0.661, 'pos': 0.339}
>>> sa.polarity_scores(text="Python is not a bad choice for most applications.")
⑥
{'compound': 0.431, 'neg': 0.0, 'neu': 0.711, 'pos': 0.289}
```

- ① `SentimentIntensityAnalyzer.lexicon` contains that dictionary of tokens and their scores that we talked about.

② ©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

- ➅ A tokenizer better be good at dealing with punctuation and emoticons (emojis) for VADER to work well. After all, emoticons are designed to convey a lot of sentiment (emotion).
- ➆ If you use a stemmer (or lemmatizer) in your pipeline, you'd need to apply that stemmer to the VADER lexicon, too, combining the scores for all the words that go together in a single stem or lemma.
- ➇ Out of 7500 tokens defined in VADER, only 3 contain spaces, and only 2 of those are actually n-grams; the other is an emoticon for "kiss".
- ➈ The VADER algorithm considers the intensity of emotion for three different polarity sentiments separately (positive, negative, and neutral) and then combines them together into a compound positivity sentiment.
- ➉ Notice that VADER handles negation pretty well—"great" has a slightly more positive sentiment than "not bad". VADER's built-in tokenizer ignores any words that aren't in its lexicon, and it doesn't consider n-grams at all.

Let's see how well this rule-based approach does for the example statements we mentioned earlier.

```
>>> corpus = ["Absolutely perfect! Love it! :-) :-) :-)",  
...           "Horrible! Completely useless. :(,  
...           "It was OK. Some good and some bad things."]  
>>> for doc in corpus:  
...       scores = sa.polarity_scores(doc)  
...       print('{:+.2f}: {}'.format(scores['compound'], doc))  
+0.9428: Absolutely perfect! Love it! :-) :-) :-)  
-0.8768: Horrible! Completely useless. :(  
+0.3254: It was OK. Some good and some bad things.
```

This looks a lot like what you wanted. So the only drawback is that VADER doesn't look at all the words in a document, only about 7,500. What if you want all the words to help add to the sentiment score? And what if you don't want to have to code your own understanding of the words in a dictionary of thousands of words or add a bunch of custom words to the dictionary in `SentimentIntensityAnalyzer.lexicon`? The rule-based approach might be impossible if you don't understand the language because you wouldn't know what scores to put in the dictionary (lexicon)!

That's what machine learning sentiment analyzers are for.

2.3.2 Naive Bayes—a machine learning sentiment analyzer

A Naive Bayes model tries to find keywords in a set of documents that are predictive of the output variable—sentiment score in your case. The nice thing about a Naive Bayes model is that the internal coefficients will map words or tokens to scores just like VADER does. Only this time you won’t have to be limited to just what an individual human decided those scores should be. The machine will find the "best" scores for any problem.

For any machine learning algorithm, you first need to find a dataset. You need a bunch of text documents that have labels for their positive emotional content (positivity sentiment). Hutto compiled four different sentiment datasets for us when he and his collaborators built VADER. You’ll load them from the `nlpia` package.³⁵

Footnote 35 If you haven’t already installed `nlpia`, check out the installation instructions at github.com/totalgood/nlpia.

```
>>> from nlpia.data.loaders import get_data
>>> movies = get_data('hutto_movies')
>>> movies.head().round(2)
   sentiment                                text
id
1      2.27  The Rock is destined to be the 21st Century...
2      3.53  The gorgeously elaborate continuation of '...'
3     -0.60                  Effective but too tepid ...
4      1.47  If you sometimes like to go to the movies t...
5      1.73  Emerges as something rare, an issue movie t...
>>> movies.describe().round(2)
   sentiment
count    10605.00
mean      0.00
min      -3.88
max       3.94
```

①

- ① It looks like movies were rated on a scale from -4 to +4.

Now let’s tokenize all those movie review texts to create a bag of words for each one. You’ll put them all into a Pandas DataFrame like you did earlier in this chapter.

```
>>> import pandas as pd
>>> pd.set_option('display.width', 140)
>>> from nltk.tokenize import casual_tokenize
>>> bags_of_words = []
>>> from collections import Counter
>>> for text in movies.text:
...     bags_of_words.append(Counter(casual_tokenize(text)))
>>> df_bows = pd.DataFrame.from_records(bags_of_words)
>>> df_bows = df_bows.fillna(0).astype(int)
```

①

②

③

④

⑤

```
>>> df_bows.shape
(10605, 20756)
>>> df_bows.head()
#   !   "   #   $   %   &   '   (   )   *   ...   zips   zombie   zombies   zone   zoning   zzzzzzzz   %   élan
# 0   0   0   0   0   0   0   4   0   0   0   ...   0   0   0   0   0   0   0   0   0   0   0
# 1   0   0   0   0   0   0   4   0   0   0   ...   0   0   0   0   0   0   0   0   0   0   0
# 2   0   0   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0   0   0   0   0
# 3   0   0   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0   0   0   0   0
# 4   0   0   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0   0   0   0   0
>>> df_bows.head()[list(bags_of_words[0].keys())]
#   The Rock is destined to be the 21st   ...   Schwarzenegger   ,   Jean   Claud   Van   ...
# 0   1   1   1   1   2   1   1   1   ...   1   1   1   1   1   1   1   ...
# 1   2   0   1   0   0   0   1   0   ...   0   0   0   0   0   0   0   ...
# 2   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0   ...
# 3   0   0   1   0   4   0   1   0   ...   0   1   0   0   0   0   0   ...
# 4   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0   ...
```

- ➊ This line just helps display wide DataFrames in the console so they look prettier here.
- ➋ NLTK's casual_tokenize can handle emoticons, unusual punctuation, and slang better than Treebank Word Tokenizer or the other tokenizers in this chapter
- ➌ The Python built-in Counter takes a list of objects and counts them up, returning a dictionary where the keys are the objects (tokens in your case) and the values are the integer counts of those objects.
- ➍ The from_records() DataFrame constructor takes a sequence of dictionaries. It creates columns for all the keys and the values are added to the table in the appropriate columns, filling missing values with NaN.
- ➎ Numpy and Pandas can only represent NaNs in float objects, so once you fill all the NaNs with zeros you can convert the DataFrame to integers, which are much more compact (in memory and to display).
- ➏ A bag-of-words table can grow quite large quickly, especially when you don't use case normalization, stop word filters, stemming, and lemmatization we discussed earlier in this chapter. Try inserting some of these dimension reducers here and see how they affect your pipeline.

Now you have all the data that a Naive Bayes model needs to find the keywords that predict sentiment from natural language text.

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> nb = MultinomialNB()
>>> nb = nb.fit(df_bows, movies.sentiment > 0) ➊
>>> movies['predicted_sentiment'] = nb.predict(df_bows) * 8 - 4 ➋
>>> movies['error'] = (movies.predicted_sentiment - movies.sentiment).abs()
>>> movies.error.mean().round(1)
2.4 ➌
>>> movies[['sentiment predicted_sentiment sentiment_ispositive predicted_ispositive'].split()
...      ].head(8)
sentiment predicted_sentiment sentiment_ispositive predicted_ispositive
id
1    2.266667          4              1              1
2    3.533333          4              1              1
3   -0.600000         -4              0              0
```

```

4    1.466667      4          1          1
5    1.733333      4          1          1
6    2.533333      4          1          1
7    2.466667      4          1          1
8    1.266667     -4          1          0
>>> hist -o -p
>>> (movies.predicted_ispositive == movies.sentiment_ispositive).sum() / len(movies)
0.9344648750589345

```

4

- ➊ Naive Bayes models are classifiers, so you need to convert your output variable (sentiment float) to a discrete label (integer, string, or bool).
- ➋ Convert your discrete classification variable back to a real value between -4 and +4 so you can compare it to the "ground truth" sentiment.
- ➌ The average absolute value of the error between your prediction (mean absolute precision or MAP) is 2.4.
- ➍ You got the "thumbs up" rating correct 93% of the time.

This is a pretty good start at building a sentiment analyzer with only a few lines of code (and a lot of data). You didn't have to compile a list of 7500 words and their sentiment like VADER did. You just gave it a bunch of text and labels for that text. That's the power of machine learning and NLP!

How well do you think it will work on a completely different set of sentiment scores like for product reviews instead of movie reviews?

If you want to build a real sentiment analyzer like this, remember to split your training data (and leave out a test set—see Appendix D for more on test/train splits). You forced your classifier to rate all the text as thumbs up or thumbs down, so a random guess would have had a MAP error of about 4. So you're about twice as good as a random guess.

```

>>> products = get_data('hutto_products')
>>> for text in products.text:
...     bags_of_words.append(Counter(casual_tokenize(text)))
>>> df_product_bows = pd.DataFrame.from_records(bags_of_words)
>>> df_product_bows = df_product_bows.fillna(0).astype(int)
>>> df_all_bows = df_bows.append(df_product_bows)
>>> df_bows2.columns
# Index(['!', '"', '#', '#38', '$', '%', '&', '!', '(', '(8',
# ...
#       'zoomed', 'zooming', 'zooms', 'zx', 'zzzzzzzz', '~', '%', 'élan', '-', ''']
>>> df_product_bows = df_all_bows.iloc[len(movies):][df_bows.columns]
>>> df_product_bows.shape
(3546, 20756)
>>> df_bows.shape
(10605, 20756)
>>> products['sentiment_ispositive'] = (products.sentiment > 0).astype(int)
>>> products['predicted_ispositive'] = nb.predict(df_product_bows.values).astype(int)
>>> products.head()
#   id  sentiment           text  sentiment_ispositive
# 0  1_1    -0.90  troubleshooting ad-2500 and ad-2600 no picture...          0

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

```
# 1 1_2      -0.15  repost from january 13, 2004 with a better fit...          0
# 2 1_3      -0.20  does your apex dvd player only play dvd audio ...        0
# 3 1_4      -0.10  or does it play audio and video but scrolling ...        0
# 4 1_5      -0.50  before you try to return the player or waste h...        0
>>> (products.predicted_ispositive == products.sentiment_ispositive).sum() / len(products)
0.5572476029328821
```

- ➊ Your new bags of words have some tokens that weren't in the original bags of words DataFrame (23302 columns now instead of 20756 before).
- ➋ You need to make sure your new product DataFrame of bags of words has the exact same columns (tokens) in the exact same order as the original one used to train your Naive Bayes model.
- ➌ This is the original movie bags of words.

So your Naive Bayes model does a poor job of predicting whether a product review is positive (thumbs up). One reason for this subpar performance is that your vocabulary from the `casual_tokenize` product texts has 2546 tokens that weren't in the movie reviews. That's about 10% of the tokens in your original movie review tokenization, which means that all those words won't have any weights or scores in your Naive Bayes model. Also the Naive Bayes model doesn't deal with negation as well as VADER does. You'd need to incorporate *n*-grams into your tokenizer to connect negation words (such as "not" or "never") to the positive words they might be used to qualify.

We leave it to you to continue the NLP action by improving on this machine learning model. And you can check your progress relative to VADER at each step of the way to see if you think machine learning is a better approach than hard-coding algorithms for NLP.

2.3.3 Summary

- You implemented tokenization and configured a tokenizer for your application.
- Several techniques can improve the accuracy of tokenization and minimizing information loss.
- *n*-gram tokenization helps retain some of the "word order" information in a document.
- Normalization, stemming, and lemmatization consolidate words into groups that improve the "recall" for search engines but reduce precision.
- Stop words can contain useful information and discarding them is not always helpful .

Math with Words (TF-IDF Vectors)

3

In this chapter

- Counting words and *Term Frequencies* to analyze meaning
- Predicting word occurrence probabilities with *Zipf's Law*
- Vector representation of words and how to start using them
- Finding relevant documents from a corpus using *Inverse Document Frequencies*
- Estimating the similarity of pairs of documents with *Cosine Similarity* and *Okapi BM25*

Having collected and counted words (tokens), and bucketed them into "stems" or "lemmas", it's time to do something interesting with them. Detecting words is useful for simple tasks, like getting statistics about word usage or doing keyword search. But we'd like to know which words are more important to a particular document and across the corpus as a whole. Then we can use that "importance" value to find relevant documents in a corpus based on keyword importance within each document. That will make a spam detector a little less likely to get tripped up by a single curse word or a few slightly-spammy words within an email. And we'd like to measure how positive and prosocial a tweet is when we have a broad range of words with various degrees of "positivity" scores or labels. If we have an idea about the frequency those words appear in a document *in relation to* the rest of the documents, we can use that further refine the "positivity" of the document. In this chapter you'll learn about a more nuanced, less binary measure of words and their usage within a document. This approach has been the mainstay for generating features from natural language for commercial search engines and spam filters for decades.

The next step in our adventure is to turn the words of Chapter 2 into continuous numbers

rather than just integers representing word counts or binary "bit vectors" that detect the presence or absence of particular words. With representations of words in a continuous space, we can operate on their representation with more exciting math. Our goal is to find numerical representation of words that somehow capture the "importance" or "information content" of the words they represent. You'll have to wait until Chapter 4 to see how to turn this information content into numbers that represent the actual *meaning* of words.

In this chapter, we will look at three increasingly powerful ways to represent words and their importance in a document:

- Bags of Words: vectors of word counts or frequencies
- Bags of N-Grams: counts of word pairs, triplets, etc.
- TF-IDF Vectors: word scores that better represent their importance

DEFINITION

TF-IDF stands for Term Frequency times Inverse Document Frequency. "Term Frequency" is just the counts of each word in a document that you learned about in previous chapters. And "Inverse Document Frequency" means that we will divide each of those word counts by the number of documents that the word occurs in.

Each of these techniques can be applied separately or as part of an NLP pipeline. These are all going to be statistical models in that they are *frequency* based. Later in the book, we will see various ways to peer ever deeper into word relationships and the patterns and non-linearities that make up that space.

However, these "shallow" NLP machines are very powerful and useful for many practical applications like spam filtering and sentiment analysis.

3.1 Bag of Words

In the previous chapter we created our first vector space model of a text using "one-hot encoding" of each word and then combining all those vectors with a binary OR (or clipped sum) to create a vector representation of a text. And this binary bag of words vector makes a great index for document retrieval when loaded into a data structure like a Pandas DataFrame.

We then looked at an even more useful vector representation is one that counts the number of occurrences, or "frequency", of each word in the given text. As a first approximation, we assume that the more times a word occurs, the more meaning it must contribute to that document. Say a document that refers to "wings" and "rudder"

frequently may be more relevant to a problem involving jet airplanes or air travel, than say a document that refers frequently to "cat"s and "gravity". Or if we have classified some words as expressing positive emotions—words like "good", "best", "joy", and "fantastic"—then the more likely a document that contains those words is to have positive "sentiment". Though one can imagine how an algorithm that relied on these simple rules might be mistaken or led astray.

Let's look at a example where counting occurrences of words is useful:

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> sentence = "The faster Harry got to the store, the faster Harry, the faster,
   would get home."
>>> tokenizer = TreebankWordTokenizer()
>>> tokens = tokenizer.tokenize(sentence.lower())
>>> tokens
['the', 'faster', 'harry', 'got', 'to', 'the', 'store', ',', 'the', 'faster', 'harry',
 ',', 'the', 'faster', ',', 'would', 'get', 'home', '.']
```

Now with our simple list we want to get unique words from the document and their counts. A Python dictionary will serve this purposed nicely, and we want to count them as well, so from the previous chapters we can use Counter.

```
>>> from collections import Counter
>>> bag_of_words = Counter(tokens)
>>> bag_of_words
Counter({' ': 3, '.': 1, 'faster': 3, 'get': 1, 'got': 1, 'harry': 2, 'home': 1,
      'store': 1, 'the': 4, 'to': 1, 'would': 1})
```

As with any good Python dictionary, the order of our keys got shuffled. The new order is optimized for storage, update, and retrieval, not consistent display. Any information inherent in the order of words has been discarded.

NOTE

In Python2 the words in our `Counter` dictionary happen to be listed in descending order by count, but that is coincidence, do not count on `Counter` to always return that way). In Python3 the dictionary may be displayed in the lexical order of the keys. But never rely on the order of the elements in a Python dictionary.

For short documents like this one, the unordered bag of words still contains a lot of information about the original intent of the sentence. And the information in a bag of words is sufficient to do some powerful things like detect spam, compute sentiment (positivity, happiness, etc), and even detect subtle intent, like sarcasm. So it may be a bag, but it's full of meaning, information. So let's get these words ranked, sorted in some

order that's easier to think about. The Counter object has a handy method, `most_common` for just this purpose.

```
>>> bag_of_words.most_common(4) ①
[('the', 4), ('.', 3), ('faster', 3), ('harry', 2)]
```

- ① By default, `most_common()` lists all tokens from most frequent to least, but we've limited the list to the top 4 here

Specifically, the number of times a word occurs in a given document is called the *term frequency*, commonly abbreviated TF. In some examples you may see the count of word occurrences normalized (divided) by the number of terms in the document.

So Our top four terms or tokens are 'the', '.', 'harry', and 'faster'. But the word 'the' and the punctuation '.' aren't very informative about the intent of this document. And these uninformative tokens are likely to appear a lot during our hurried adventure. So for this example, we will ignore them along with a list of standard English stop words and punctuation. This will not always be the case, but for now it helps simplify the example. That leaves us with 'harry' and 'faster' among the top tokens in our TF vector (bag of words):

Let's calculate the term frequency of 'harry' from the Counter object ("bag_of_words") we defined above:

```
>>> times_harry_appears = bag_of_words['harry']
>>> num_unique_words = len(bag_of_words) ①
>>> tf = times_harry_appears / num_unique_words
>>> round(tf, 4)
0.1818
```

- ① The number of unique tokens from our original source.

Let's pause for a second and look a little deeper at *term frequency*, as it is a phrase (and calculation) we will use often throughout this book. It is basically the word count tempered by how long the document is. But why "temper" it all? Let's say you find the word "dog" 3 times in document A and 100 times in document B. Clearly "dog" is way more important to document B. But wait. Let's say you find out document A is a 30 word email to a veterinarian and document B is *War & Peace* (approx 580,000 words!). Our first analysis was straight up backwards. But if we take the document length into account:

$$TF('dog', document_A) = 3/30 = .1$$

Figure 3.1 TF of "dog" in documentA = 3/30 = .1

$$TF('dog', document_B) = 100/580000 = 0.00017$$

Figure 3.2 TF of "dog" in documentB = 100/580000 = .00017

Now we have something you can see describes "something" about the two documents and their relationship to the word "dog" and each other. So instead of raw word counts to describe our documents in a corpus, we can use *Term Frequencies*.

Similarly we could calculate each word and get the relative "importance" to the document of that term. Our protagonist, Harry, and his need for speed are clearly central to the story of this document, we've made some great progress in turning text into numbers, beyond just the presence or absence of a given word. Now this is a clearly contrived example, but one can quickly see how meaningful results could come from this approach. Let's look at a bigger piece of text. Take these first few paragraphs from the Wikipedia article on Kites.

A kite is traditionally a tethered heavier-than-air craft with wing surfaces that react against the air to create lift and drag. A kite consists of wings, tethers, and anchors. Kites often have a bridle to guide the face of the kite at the correct angle so the wind can lift it. A kite's wing also may be so designed so a bridle is not needed; when kiting a sailplane for launch, the tether meets the wing at a single point. A kite may have fixed or moving anchors. Untraditionally in technical kiting, a kite consists of tether-set-coupled wing sets; even in technical kiting, though, a wing in the system is still often called the kite.

The lift that sustains the kite in flight is generated when air flows around the kite's surface, producing low pressure above and high pressure below the wings. The interaction with the wind also generates horizontal drag along the direction of the wind. The resultant force vector from the lift and drag force components is opposed by the tension of one or more of the lines or tethers to which the kite is attached. The anchor point of the kite line may be static or moving (e.g., the towing of a kite by a running person, boat, free-falling anchors as in paragliders and fugitive parakites or vehicle).

The same principles of fluid flow apply in liquids and kites are also used under water.

A hybrid tethered craft comprising both a lighter-than-air balloon as well as a kite lifting surface is called a kytoon.

Kites have a long and varied history and many different types are flown individually and at festivals worldwide. Kites may be flown for recreation, art or other practical uses.

Sport kites can be flown in aerial ballet, sometimes as part of a competition. Power kites are multi-line steerable kites designed to generate large forces which can be used to power activities such as kite surfing, kite landboarding, kite fishing, kite buggying and a new trend snow kiting. Even Man-lifting kites have been made.

-- Wikipedia Kites (<https://en.wikipedia.org/wiki/Kite>)

Then we will assign the above text to a variable:

```
>>> from collections import Counter
>>> from nltk.tokenize import TreebankWordTokenizer
>>> tokenizer = TreebankWordTokenizer()
>>> from nlpia.data.loaders import kite_text
1
>>> tokens = tokenizer.tokenize(kite_text.lower())
>>> token_counts = Counter(tokens)
>>> token_counts
Counter({'the': 26, 'a': 20, 'kite': 16, ',': 15, ...})
```

- ① kite_text = "A kite is traditionally ..." as above

NOTE

Interestingly the Treebank tokenizer returns 'kite.' (with a period) as a token. Each tokenizer (such as a RegexpTokenizer) treats punctuation differently, and you'll get similar but different results. They each have their advantages and we encourage to experiment. Just a nice reminder that NLP is hard.

Okay, back to the example. So that is a lot of stopwords. If we are just looking at raw word count, and we are, this article isn't going to tell us a great deal about *the*, *a*, *and*, and *of*. So let's ditch them for now:

```
>>> import nltk
>>> nltk.download('stopwords', quiet=True)
True
>>> stopwords = nltk.corpus.stopwords.words('english')
>>> tokens = [x for x in tokens if x not in stopwords]
>>> kite_counts = Counter(tokens)
>>> kite_counts
Counter({'kite': 16, ',': 15, 'kites': 8, 'wing': 5, 'lift': 4...})
```

Haha! By looking purely at the number of time words occur in this document we are learning something about it. The terms *kite(s)*, *wing*, *lift* are all very important. And, if we didn't actually know what this document was about, we just happened across this document in our vast database of Google-like knowledge, we might "programmatically" be able to infer it has something to do with "flight" or "lift" or, in fact, "kites".

Across multiple documents in a corpus things get a little more interesting. A set of documents may *all* be about, say, kite flying. You would imagine all of the documents may refer to string and wind quite often, and the term frequencies `TF("string")` and `TF("wind")` would therefore rank very highly in all of the documents. Now let's look at a way to more gracefully represent these numbers for mathematical intents.

3.2 Vectorizing

We've transformed our text into numbers on a basic level. But we've still just stored them in a dictionary, so we've taken one step out of the text-based world and into the realm of mathematics, let's go ahead and jump all the way. Instead of describing a document in terms of a frequency dictionary, let's make a vector of those word counts. In Python, this will just be a list. But in general it is an ordered collection or array. We can do this quickly with:

```
document_vector = []
doc_length = len(tokens)
for key, value in kite_counts.most_common():
    document_vector.append(value / doc_length)

print(document_vector)

[0.07207207207207, 0.06756756756756757, 0.036036036036036036, 0.02252252252252252,
 ..(+ others)..., 0.0045045045045045, 0.0045045045045045]
```

This list, or *vector*, is something we can do math on directly.

TIP

There are many ways to speed up processing of these data structures³⁶, but for now as we are playing with the nuts and bolts, but very soon we'll want to speed things up.

Footnote 36 www.numpy.org/

Math isn't very interesting with just one element. We need some more elements. Since that first vector represents a document, we can grab a couple more documents and make vectors for each of them as well. But we want to make them uniform, because if we are going to do math on them it is important that they represent a position in a common space, relative to something specific. There are 2 steps to this: first, regularizing the size of the counts by calculating *Term Frequency* instead of raw count in the document (as we did in the last section) and second, making all of the vectors of standard length or *dimension*.

Specifically we want each position in the vector to represent the same word in each

document's vector. But you may notice that your email to your vet is not going to contain many of the words that are in *War & Peace*, or maybe it will, who knows. But it is fine (and as it happens, necessary) if our vectors contain values of 0 in various positions. So we will find every unique word in each document and then find every unique word in the union of those two sets. This collections of words in our vocabulary is often called a *lexicon*. The lexicon is same concept referenced in the earlier chapters, just in terms of our special corpus. Let's look at what that would look like with something shorter than *War & Peace*. Let's check in on Harry. We had one "document" already, let's round out the corpus with a couple more.

```
>>> docs = ["The faster Harry got to the store,
           the faster and faster Harry would get home."]
>>> docs = docs + ["Harry is hairy and faster than Jill."]
>>> docs = docs + ["Jill is not as hairy as Harry."]
```

TIP

If you're playing along with us, rather than typing these out just import them from the `nlpia` package: `from nlpia.data.loaders import harry_docs as docs`

First, lets look at our *lexicon* for this *corpus* containing 3 *documents*.

```
>>> doc_tokens = []
>>> for doc in docs:
...     doc_tokens += [sorted(tokenizer.tokenize(doc.lower()))]
>>> len(doc_tokens[0])
17
>>> all_doc_tokens = sum(doc_tokens, [])
>>> len(all_doc_tokens)
33
>>> lexicon = sorted(set(all_doc_tokens))
>>> len(lexicon)
18
>>> lexicon
[',', '.', 'and', 'as', 'faster', 'get', 'got', 'hairy', 'harry', 'home', 'is', 'jill',
 'not', 'store', 'than', 'the', 'to', 'would']
```

Each of our 3 document vectors will need to have 18 values, even if the document for that vector does not contain all of the 18 words in our lexicon. Each token is assigned a "slot" in our vectors corresponding to its position in our lexicon. Some of those token counts in the vector will be zeros and that's what we want.

```
>>> from collections import OrderedDict
>>> zero_vector = OrderedDict((token, 0) for token in lexicon)
>>> zero_vector
OrderedDict([(' ', 0), ('.', 0), ('and', 0), ('as', 0), ('faster', 0), ('get', 0), ...
```

Now we'll make copies of that base vector, update the values of the vector for each document and store them in an array.

```
>>> import copy
>>> doc_vectors = []
>>> for doc in docs:
...     vec = copy.copy(zero_vector) ①
...     tokens = tokenizer.tokenize(doc.lower())
...     token_counts = Counter(tokens)
...     for key, value in token_counts.items():
...         vec[key] = value / len(lexicon)
...     doc_vectors.append(vec)
```

- ① `copy.copy()` Creates an independent copy, a separate instance of our zero vector, rather than reusing a reference (pointer) to the original object's memory location. Otherwise we just be overwriting the same `zero_vector` with new values in each loop. Then we wouldn't have a fresh zero on each pass of the loop.

Now we have 3 vectors, one for each document. So what? What can we do with them? Our document word-count vectors can do all the cool stuff that any vector can do, so let's learn a bit more about vectors and vector spaces first.³⁷

Footnote 37 If you would like more details about linear algebra and vectors take a look at Appendix C

3.2.1 Vector Spaces

Vectors are the primary building blocks of linear algebra, or vector algebra. They are simply an ordered list of numbers, or coordinates, in a vector space. They describe a location or position in that space. Or they can be used to identify a particular direction and magnitude or distance in that space. A *space* is the collection of all possible vectors that could appear in that space. So a vector with 2 values would lie in a 2 dimensional vector space. A vector with 3 values in 3D vector space, etc.

A piece of graph paper, or a grid of pixels in an image, are both nice 2D vector spaces. You can see how the order of these coordinates matter. If you reverse the x and y coordinates for locations on your graph paper, without reversing all your vector calculations, all your answers for linear algebra problems would be flipped. Graph paper and images are examples of rectilinear, or Euclidean spaces, because the x and y coordinates are perpendicular to each other. The vectors we'll talk about in this chapter all all rectilinear, Euclidean spaces.

What about latitude and longitude on an map or globe? That map or globe is definitely a 2D vector space because it's an ordered list of 2 numbers, latitude and longitude. But each of the latitude-longitude pairs describes a point on an approximately spherical,

bumpy surface—the surface of the Earth. And latitude and longitude coordinates are not exactly perpendicular. So a latitude-longitude vector space is not rectilinear. That means we have to be careful when we calculate things like distance or closeness (similarity) between two points represented by a pair of 2D latitude-longitude vectors, or vectors in any non-Euclidean space. Think about how you'd calculated the distance between the latitude and longitude coordinates of Portland, OR and New York, NY.³⁸ to get the math right]

Footnote 38 You'd need to use a package like `geopy.readthedocs.io[GeoPy]`

Here's one way many people might draw the 2D vectors $(5, 5)$, $(3, 2)$, and $(-1, 1)$. The head of vector (represented by the pointy tip of an arrow) is used to identify a location in a vector space. So the vector heads in this diagram will be at those three pairs of coordinates. The tail of a position vector (represented by the "rear" of the arrow) is always at the origin, or $(0, 0)$.

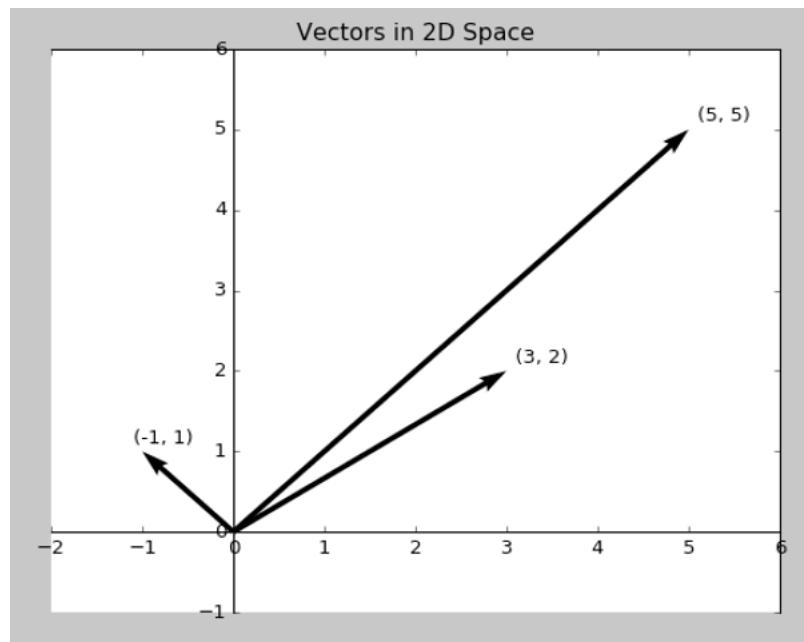


Figure 3.3 2D Vectors

What about 3D vector spaces? Positions and velocities in the 3D physical world we live in can be represented by x, y, and z coordinates in a 3D vector. Or the curvilinear space formed by all the latitude-longitude-altitude triplets describing locations near the surface of the Earth.

But we aren't limited to normal 3D space. We can have 5 dimensions; 10 dimensions; 5,000; whatever. The linear algebra all works out the same. We just might need more computing power as the dimensionality grows. And we'll run into some

curse-of-dimensionality issues, but we'll wait to deal with that until the last chapter, Chapter 13.³⁹

Footnote 39 The "curse of dimensionality" is that vectors will get exponentially farther and farther away from one another, in Euclidean distance, as the dimensionality increases. A lot of simple operations become impractical above 10 or 20 dimensions, like sorting a large list of vectors based on their distance from a "query" or "reference" vector (approximate nearest neighbor search). To dig deeper, check out Wikipedia's ["Curse of Dimensionality" article](#), [explore hyperspace with one of this book's authors at bit.ly/explorehyperspace](#), play with the Python [annoy package](#) or [search Google Scholar](#) for "high dimensional approximate nearest neighbors"

For our natural language document vector space, for TF (and TF-IDF to come), the dimensionality of our vector space will be the count the number of distinct words that appear in the entire corpus. We'll call this capital letter "K". This number of distinct words is also the vocabulary size of our corpus, so in an academic paper it'll usually be called " $|V|$ " We can then describe each document, within this K-dimensional vector space by a K-dimensional vector. $K = 18$ in our 3 document corpus about Harry and Jill. Since we as humans can't easily visualize spaces of more than 3 dimensions, let's set aside most of those dimensions and just look at 2 for a moment, so we can have a visual representation of the vectors on this flat page that you're reading.

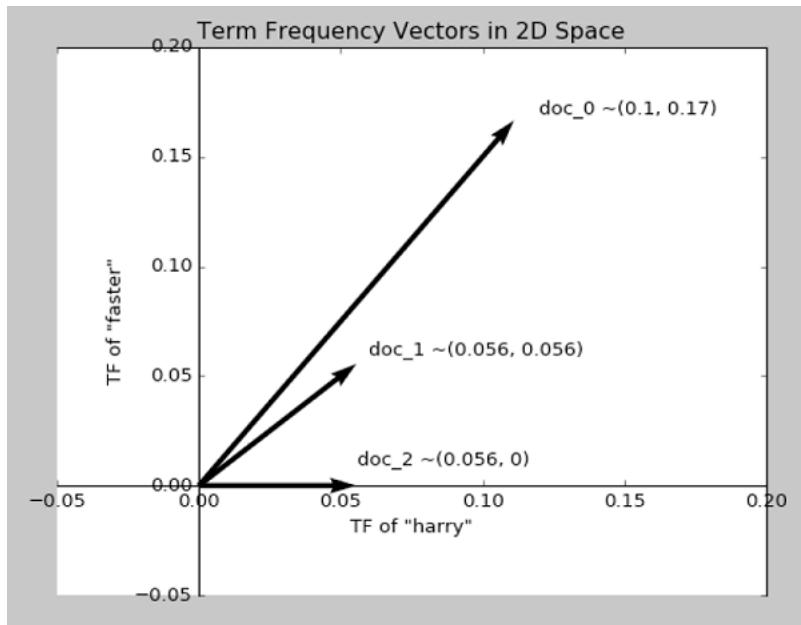


Figure 3.4 2D Term Frequency Vectors

K-dimensional vectors will work the same way, just in ways we can't easily visualize. Now that we have a representation of each our document and know they share a common space, we have a path to comparing them. We could just measure the Euclidean distance between the vectors by subtracting them and computing the length of that distance

between them. This is called the 2-norm distance. It's the distance a "crow" would have to fly (in a straight line) to get from a location identified by the tip (head) of one vector and the location of the tip of the other vector. Check out Appendix C on Linear Algebra to see why this is a bad idea for word count (term frequency) vectors.

Two vectors are "similar" if they share similar direction. They might have similar magnitude (length), which would mean that the word count (term frequency) vectors are for documents of about the same length. But do we care about document length in our similarity estimate for vector representations of words in documents? Probably not. We'd like our estimate of document similarity to find use the same words about the same amount of times in similar proportions. This would give us confidence that the documents they represent are probably talking about similar things.

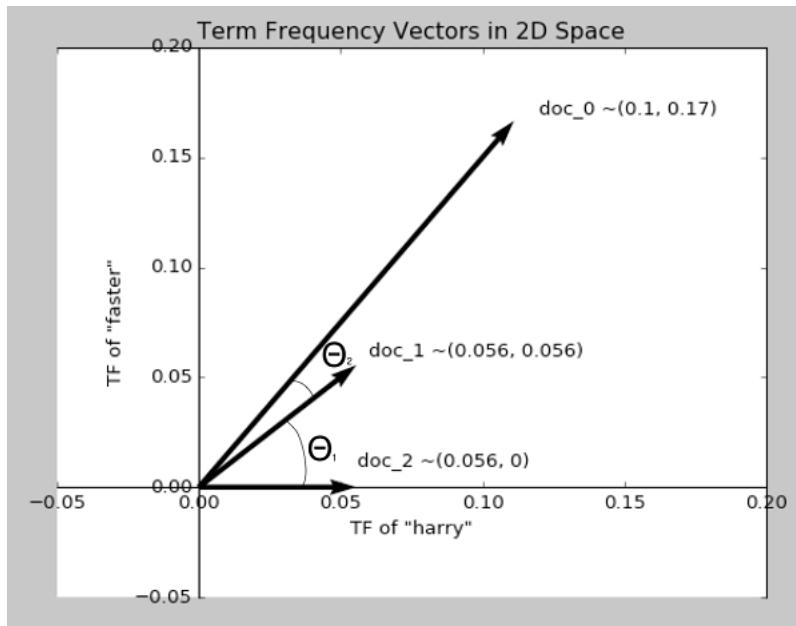


Figure 3.5 2D Thetas

Cosine Similarity is merely the cosine of the angle between two vectors (theta), which can be calculated from the Euclidian Dot Product by:

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}| |\mathbf{B}| * \cos \Theta$$

Figure 3.6 Dot Product: $\mathbf{A} \cdot \mathbf{B} = \text{norm}(\mathbf{A}) * \text{norm}(\mathbf{B}) * \cos(\theta)$

From which we can then derive:

$$\cos \Theta = \frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{A}| |\mathbf{B}|}$$

Figure 3.7 Cosine Similarity: $\cos(\theta) = \mathbf{A} \cdot \mathbf{B} / \text{norm}(\mathbf{A}) * \text{norm}(\mathbf{B})$

Or in Python:

```
>>> import math
>>> def cosine_sim(vec1, vec2):
...     """ Let's convert our dictionaries to lists for easier matching. """
...     vec1 = [val for val in vec1.values()]
...     vec2 = [val for val in vec2.values()]

...
...     dot_prod = 0
...     for i, v in enumerate(vec1):
...         dot_prod += v * vec2[i]

...
...     mag_1 = math.sqrt(sum([x**2 for x in vec1]))
...     mag_2 = math.sqrt(sum([x**2 for x in vec2]))
...
...     return dot_prod / (mag_1 * mag_2)
```

So we just need to take the dot product of two of our vectors in question—multiply the elements of each vector pairwise and then sum those products up. We then divide by the norm (magnitude or length) of each vector. The norm of a vector, is the same as its Euclidean distance from the head to the tail of the vector—the square root of the sum of the squares of its elements. This *normalized dot product*, like the output of the cosine function, will be a value between -1 and 1. It is the cosine of the angle between these 2 vectors. This is the same as the portion of the longer vector that is covered by the shorter vector’s perpendicular projection onto the longer one. It gives us a value for how much the vectors point in the same direction.

A cosine similarity of *1* represents identical normalized vectors that point in exactly the same direction along all dimensions. The vectors may have different lengths or magnitudes, but they point in the same direction. Remember we divided the dot product by the norm of each vector, and this can happen before or after the dot product. So the vectors are normalized so they both have a length of 1 as we do the dot product. So the closer a cosine similarity value is to 1, the closer the two vectors are in angle. For NLP document vectors that have a cosine similarity close to 1, we know that the documents are using similar words in similar proportion. So the documents whose document vectors are close to each other a likely talking about the same thing.

A cosine similarity of *0* represents two vectors that share no components. They are orthogonal, perpendicular in all dimensions. For NLP TF vectors this can only happen if the two documents share no words in common. Since these documents use completely different words, they must be talking about completely different things. This doesn’t necessarily mean they have different meanings or topics, just that they use completely different words.

A cosine similarity of -1 represents two vectors that are anti-similar, completely opposite. They point in opposite directions. This can never happen for simple word count (term frequency) vectors or even normalized TF vectors (which we'll talk about later). Counts of words can never be negative. So word count (term frequency) vectors will always be in the same "quadrant" of the vector space. None of the term frequency vectors can "sneak around behind" into one of the quadrants behind the tail of the other vectors. None of our term frequency vectors can have components (word frequencies) that are the negative of another term frequency vector, because term frequencies just can't be negative.

You will not see any negative cosine similarity values for pairs of vectors for natural language documents in this chapter. However, in the next chapter we'll develop a concept of words and topics that are "opposite" to each other. And this will show up as documents, words, and topics that have cosine similarities of less than zero, or even -1 .

TIP

Opposites Attract

There's an interesting consequence of the way we calculated cosine similarity. If 2 vectors or documents have a cosine similarity of -1 (are opposites) to a 3rd vector, then they must be perfectly similar to each other. They must be exactly the same vectors. However the documents those vectors represent may not be exactly the same. Not only might the word order be shuffled, but one may be much longer than the other, if it uses the same words in the same proportion.

We will see later that we will come up with vectors that more accurately model a document. But, for now, this gives an introduction to the tools we need.

3.3 Zipf's Law

So, now on to our main topic Sociology. Okay, not really, but we will make a quick detour into the world of counting people and words and a seemingly universal rule that governs the counting of most things. It turns out, that in language, like most things involving living organisms, patterns abound.

In the early twentieth century, the French stenographer Jean-Baptiste Estoup noticed a pattern in the frequencies of words that he painstakingly counted by hand across many documents (thank goodness for computers and Python). Later in the 30's, the American linguist George Kingsley Zipf sought to formalize Estoup's observation, and this relationship eventually came to bear Zipf's name.

Zipf's law states that given some corpus of natural language utterances, the frequency of any word is inversely proportional to its rank in the frequency table.

-- Wikipedia Zipf's Law https://en.wikipedia.org/wiki/Zipf%27s_law

Specifically, *inverse proportionality* refers to a situation where an item in a ranked list will appear with a frequency tied explicitly to its rank in the list. The first item in the ranked list, will appear twice as much as the second, and three times as much as third, for example. So, one of the quick things you can do with any corpus or document is plot the frequencies of word usages relative to their rank (in frequency). If you see any outliers that don't fall along a straight line in a log-log plot, then it may be worth investigating.

As an example of how far Zipf's Law stretches beyond the world of words, below is a graph charting the relationship between the population of US cities and the rank of that population. It's amazing that something so simple could hold true across a vast array of applications. Nobel Laureate Paul Krugman, speaking about economic models, put it very succinctly:

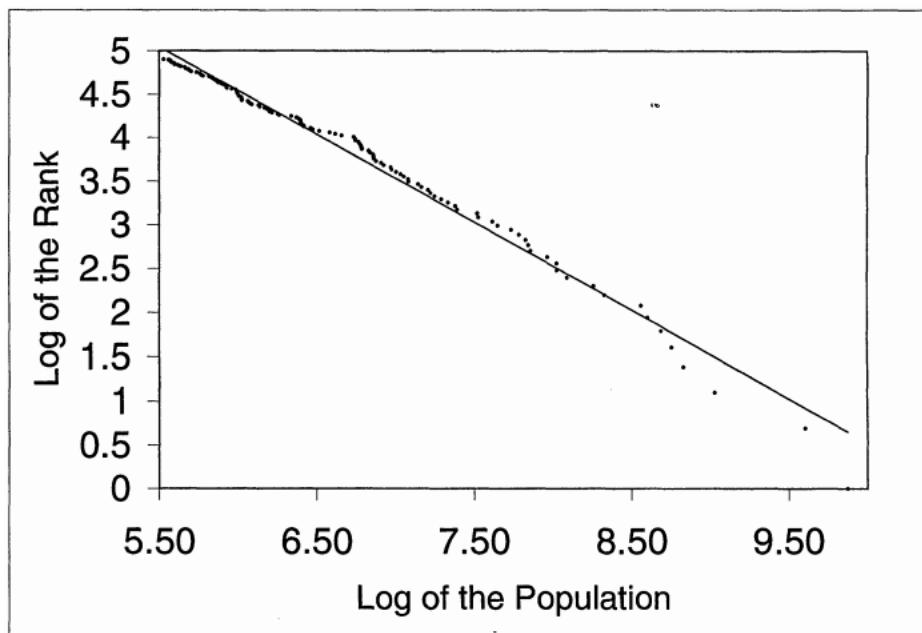


FIGURE I
Log Size versus Log Rank of the 135 largest U. S. Metropolitan Areas in 1991
Source: Statistical Abstract of the United States [1993].

Figure 3.8 City Population Distributions

The usual complaint about economic theory is that our models are oversimplified — that they offer excessively neat views of complex, messy reality. [With Zipf's law] the reverse is true: we have complex, messy models, yet reality is startlingly neat and simple.

-- Paul Krugman The Self-Organizing Economy

As with cities, so with words. Let's first download the Brown Corpus from NLTK.

The Brown Corpus was the first million-word electronic corpus of English, created in 1961 at Brown University. This corpus contains text from 500 sources, and the sources have been categorized by genre, such as news, editorial, and so on. 1.1 gives an example of each genre.⁴⁰

Footnote 40 for a complete list, see icame.uib.no/brown/bcm-los.html

-- NLTK Documentation

```
>>> from nltk.corpus import brown
>>> len(brown.words())
1161192
```

- ➊ .words() is a built-in method of the nltk corpus object that returns a list of tokens

So with over 1 million tokens, we have something meaty to look at.

```
>>> from collections import Counter
>>> puncs = [',', '.', '--', '-', '!', '?', ':', ';', '``', "''", '(', ')', '[', ']']
>>> word_list = [x.lower() for x in brown.words() if x not in puncs]
>>> token_counts = Counter(word_list)
>>> token_counts.most_common(20) # builtin method of a Counter Object
[('the', 69971), ('of', 36412), ('and', 28853), ('to', 26158), ('a', 23195),
 ('in', 21337), ('that', 10594), ('is', 10109), ('was', 9815), ('he', 9548),
 ('for', 9489), ('it', 8760), ('with', 7289), ('as', 7253), ('his', 6996), ('on', 6741),
 ('be', 6377), ('at', 5372), ('by', 5306), ('i', 5164)]
```

So if we disregard the punctuation tokens a quick glance shows a (very) rough approximation of what Zipf says we should expect to see. *The* occurs roughly twice as many times as *of*, and roughly three times as often as *and*, the third item on the list.

In short, this amounts to: if you rank the words of a corpus by the number of times they occur and list them in descending order, you will find that, for a sufficiently large sample, the first word in that ranked list is twice as likely to occur in the corpus as the second word in the list is. And it is four times as likely to appear as the fourth word on the list. So given a large corpus, we can use this break down to say statistically how likely a given word is to appear in any given document of that corpus.

3.4 Topic Modeling

Now back to our document vectors. Word counts are useful, but pure word count, even when normalized by the length of the document doesn't tell us much about the importance of that word to that document *relative* to the rest of the documents in the corpus. If we could suss out that information, we could start to describe documents within the corpus. Say we have a corpus of every kite book ever written. *Kite* would almost surely occur many times in every book (document) we counted, but that doesn't provide any new information, it doesn't help distinguish between those documents. Whereas something like *construction* or *aerodynamics* might not be so prevalent across the entire corpus, but for the ones where it frequently occurred, we would know more about those documents' nature. For this we need another tool.

Inverse Document Frequency, or *IDF*, is our window through Zipf in topic analysis. Let's take our term frequency counter from earlier and expand on it. There are two ways we can count tokens and bin them up: per document and across the entire corpus. We are going to be counting just by document.

Let's return to our Kite example from Wikipedia and grab another section (the History section) and say it is the second document in our Kite corpus.

Kites were invented in China, where materials ideal for kite building were readily available: silk fabric for sail material; fine, high-tensile-strength silk for flying line; and resilient bamboo for a strong, lightweight framework.

The kite has been claimed as the invention of the 5th-century BC Chinese philosophers Mozi (also Mo Di) and Lu Ban (also Gongshu Ban). By 549 AD paper kites were certainly being flown, as it was recorded that in that year a paper kite was used as a message for a rescue mission. Ancient and medieval Chinese sources describe kites being used for measuring distances, testing the wind, lifting men, signaling, and communication for military operations. The earliest known Chinese kites were flat (not bowed) and often rectangular. Later, tailless kites incorporated a stabilizing bowline. Kites were decorated with mythological motifs and legendary figures; some were fitted with strings and whistles to make musical sounds while flying. From China, kites were introduced to Cambodia, Thailand, India, Japan, Korea and the western world.

After its introduction into India, the kite further evolved into the fighter kite, known as the patang in India, where thousands are flown every year on festivals such as Makar Sankranti.

Kites were known throughout Polynesia, as far as New Zealand, with the assumption being that the knowledge diffused from China along with the people. Anthropomorphic kites made from cloth and wood were used in religious ceremonies to send prayers to the gods. Polynesian kite traditions are used by anthropologists to get an idea of early "primitive" Asian traditions that are believed to have at one time existed in Asia.

-- Wikipedia Kites: History (<https://en.wikipedia.org/wiki/Kite>)

First let's get the total word count for each document in our corpus: intro_doc and history_doc.

```
>>> from nlpia.data.loaders import kite_text, kite_history
>>> kite_intro = kite_text.lower() ①
>>> intro_tokens = tokenizer.tokenize(kite_intro)
>>> kite_history = kite_history.lower() # "Kites were invented in China, ..."
# Also as above
>>> history_tokens = tokenizer.tokenize(kite_history)
>>> intro_total = len(intro_tokens)
>>> intro_total
363
>>> history_total = len(history_tokens)
>>> history_total
297
```

① "A kite is traditionally ..." "a kite is traditionally ..."

Now with a couple tokenized kite documents in hand, lets look at the *term frequency* of "kite" in each document. We'll store the TF's we find in 2 dictionaries, one for each document.

```
>>> intro_tf = {}
>>> history_tf = {}
>>> intro_counts = Counter(intro_tokens)
>>> intro_tf['kite'] = intro_counts['kite'] / intro_total
>>> history_counts = Counter(history_tokens)
>>> history_tf['kite'] = history_counts['kite'] / history_total
>>> 'Term Frequency of "kite" in intro is: {:.4f}'.format(intro_tf['kite'])
'Term Frequency of "kite" in intro is: 0.0441'
>>> 'Term Frequency of "kite" in history is: {:.4f}'.format(history_tf['kite'])
'Term Frequency of "kite" in history is: 0.0202'
```

Okay we have a number is twice as large as the other. Is the intro section twice as much about kites? No, not really. So let's dig a little deeper. First, let's see how those numbers relate to some other word, say *and*.

```
>>> intro_tf['and'] = intro_counts['and'] / intro_total
>>> history_tf['and'] = history_counts['and'] / history_total
>>> print('Term Frequency of "and" in intro is: {:.4f}'.format(intro_tf['and']))
Term Frequency of "and" in intro is: 0.0275
```

```
>>> print('Term Frequency of "and" in history is: {:.4f}'.format(history_tf['and']))
Term Frequency of "and" in history is: 0.0303
```

Great! We know both of these documents are about *and* just as much as they are about *kite*! Oh, wait. That's not actually helpful, huh? Just as in our first example, where the system seemed to think *the* was the most import word in the document about our fast friend Harry; in this example *and* is considered highly relevant. Even at first glance, we can tell this isn't revelatory.

A good way to think of the *Inverse Document Frequency* of a term is, how strange is it that this token in this document? If a term appears in one document a lot times, but occurs rarely in the rest of the corpus, one could assume it is important to that document specifically. Our first step toward topic analysis!

The *IDF* of a term is merely the ratio of the total number of documents to the number of documents the term appears in. In the case of *and* and *kite* in our current example the answer would be the same for both:

$2 \text{ total documents} / 2 \text{ documents contain } \textit{and} = 2/2 = 1$

$2 \text{ total documents} / 2 \text{ documents contain } \textit{kite} = 2/2 = 1$

Not very interesting. So let's look at another word *China*.

$2 \text{ total documents} / 1 \text{ document contains } \textit{China} = 2/1 = 2$

Okay, that's something different. Let's use this "rarity" measure to weight the Term Frequencies.

```
num_docs_containing_and = 0
for doc in [intro_tokens, history_tokens]:
    if 'and' in doc:
        num_docs_containing_and += 1
```

(similarly for *kite* and *China*)

And let's grab the TF of *China* in the two documents:

```
intro_tf['china'] = intro_counts['china'] / intro_total
history_tf['china'] = history_counts['china'] / history_total
```

And finally, the idf for all 3. We'll store the idf's in dictionaries per document like we did with tf above:

```

num_docs = 2
intro_idf = {}
history_idf = {}
intro_idf['and'] = num_docs / num_docs_containing_and
history_idf['and'] = num_docs / num_docs_containing_and
intro_idf['kite'] = num_docs / num_docs_containing_kite
history_idf['kite'] = num_docs / num_docs_containing_kite
intro_idf['china'] = num_docs / num_docs_containing_china
history_idf['china'] = num_docs / num_docs_containing_china

```

And then for the intro document we find:

```

intro_tfidf = {}

intro_tfidf['and'] = intro_tf['and'] * intro_idf['and']
intro_tfidf['kite'] = intro_tf['kite'] * intro_idf['kite']
intro_tfidf['china'] = intro_tf['china'] * intro_idf['china']

```

And then for the history document:

```

history_tfidf = {}

history_tfidf['and'] = history_tf['and'] * history_idf['and']
history_tfidf['kite'] = history_tf['kite'] * history_idf['kite']
history_tfidf['china'] = history_tf['china'] * history_idf['china']

```

3.4.1 Return of Zipf

We are almost there. Let's say though we have a corpus of 1 million documents (maybe we are baby-Google), and someone searches for the word *cat*, and in our 1 million documents we have exactly 1 document that contains the word *cat*. The raw IDF of this would be:

$$1,000,000 / 1 = 1,000,000$$

Let's imagine we have 10 documents with the word *dog* in them. Our IDF for *dog* would be:

$$1,000,000 / 10 = 100,000$$

That's a big difference. Our friend Zipf would say too big. For reasons relating to his law, we'll give him the benefit of the doubt (and leave it to the mathematically inclined reader to work out the details) and temper this result with a log function. And in so doing we will redefine *IDF* from the raw ratio of number of document:number of documents, to the log of that ratio. As it turns out, the base of log function is not important, merely the smooth tempering effect as the value of the ratio `number_of_docs /`

`number_of_docs_containing_our_word` grows large. For the ease of this example, we'll use base 10 and get:

$$idf = \log(1,000,000/1) = 6$$

Figure 3.9 search: cat

$$idf = \log(1,000,000/10) = 5$$

Figure 3.10 search: dog

So now we are weighting the TF results of each more appropriately to the their occurrences in language, in general

And then finally, for a given term, t , in a given document, d , in a corpus, D , we get:

$$tf(t, d) = count(t)/count(d)$$

Figure 3.11 TT

$$idf(t, D) = \log(\text{num of documents}/\text{num of documents containing } t)$$

Figure 3.12 IDF

$$tfidf(t, d, D) = tf(t, d) * idf(t, D)$$

Figure 3.13 TFIDF

So the more times a word appears in the document the TF (and hence the TF-IDF) will go up. At the same time, as the number of documents that contain that word goes up, the IDF (and hence the TF-IDF) for that word will go down. So now, we have a number. Something our computer can chew on. But what is it exactly? It relates a specific word or token to a specific document in a specific corpus, and then assigns a numeric value to the importance of that word in the given document, given its usage across the entire corpus.

This single number, the TF-IDF is the humble foundation of a simple search engine. As we've stepped from the realm of text firmly into the realm of numbers it's time for some math. The preceding formulas and concepts for computing TF-IDF and Linear Algebra are not necessary for full understanding of the tools used in Natural Language Processing, but a general familiarity with how they work can make their use more intuitive.

3.4.2 Relevance Ranking

As we saw above, we can easily compare 2 vectors and get their similarity, but we have since learned that merely counting words isn't as descriptive as using their *TF-IDF*, so in each document vector let's replace each word's `word_count`, with the word's *TF-IDF*. Now our vectors will more thoroughly reflect the meaning, or *topic*, of the document. So in our Harry example:

```
>>> document_tfidf_vectors = []
>>> for doc in docs:
...     vec = copy.copy(zero_vector) ①
...     tokens = tokenizer.tokenize(doc.lower())
...     token_counts = Counter(tokens)
...
...     for key, value in token_counts.items():
...         docs_containing_key = 0
...         for _doc in docs:
...             if key in _doc:
...                 docs_containing_key += 1
...         tf = value / len(lexicon)
...         if docs_containing_key:
...             idf = len(docs) / docs_containing_key
...         else:
...             idf = 0
...         vec[key] = tf * idf
...     document_tfidf_vectors.append(vec)
```

- ① We need to copy the `zero_vector` to create a new, separate object. Otherwise we'd end up just overwriting the same object/vector each time through the loop.

With this we have K-dimensional vector representation of each document in the corpus. And now onto the hunt! Or search, in our case. Two vectors, in a given vector space can be said to be similar if they have a similar angle. If you imagine each vector starting at the origin and reaching out its prescribed distance and direction:

Two vectors are considered similar if their cosine similarity is high, so we can find 2 similar vectors near each other if they minimize:

$$\cos \Theta = \frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{A}| |\mathbf{B}|}$$

Figure 3.14 Cosine Distance— $\cos(\theta) = \mathbf{A} \cdot \mathbf{B} / (\|\mathbf{A}\| \|\mathbf{B}\|)$

Now we have all we need to do a basic TF-IDF based search. We can treat the search query itself as a document, and therefore get the a TF-IDF based vector representation of it. The last step is then to just find the documents whose vectors have the highest cosine similarities to the query and return those as the search results.

If we take our 3 documents about Harry, and make the query "How long does it take to get to the store?":

```
>>> query = "How long does it take to get to the store?"
>>> query_vec = copy.copy(zero_vector)
>>> query_vec = copy.copy(zero_vector) ①
```

- ① So we are dealing with separate objects, not multiple references to the same object

```
tokens = tokenizer.tokenize(query.lower())
token_counts = Counter(tokens)

for key, value in token_counts.items():
    docs_containing_key = 0
    for _doc in documents:
        if key in _doc.lower():
            docs_containing_key += 1
    if docs_containing_key == 0: ①
        continue
    tf = value / len(tokens)
    idf = len(documents) / docs_containing_key
    query_vec[key] = tf * idf

print(cosine_sim(query_vec, document_tfidf_vectors[0]))
print(cosine_sim(query_vec, document_tfidf_vectors[1]))
print(cosine_sim(query_vec, document_tfidf_vectors[2]))

0.5235048549676834
0.0
0.0
```

- ① We didn't find that token in the lexicon go to next key

So we can safely say document 0 has the most relevance for our query! And with this we can find relevant documents amidst any corpus, be it articles in Wikipedia, books from Gutenberg, or tweets from the wild west that is Twitter. Google look out!

Actually, Google's search engine is safe from competition from us. We have to do an "index scan" of our TF-IDF vectors with each query. That's an $O(N)$ algorithm. Most search engines can respond in constant time ($O(1)$) because they use an *inverted index*.⁴¹ We aren't going to implement an index that can find these matches in constant time here, but if you're interested you might like exploring the state-of-the-art Python implementation in the whoosh⁴² package and its source code.⁴³ Instead of showing you how to build this conventional keyword-based search engine, in Chapter 4 we'll show you the latest semantic indexing approaches that capture the meaning of text.

Footnote 41 en.wikipedia.org/wiki/Inverted_index

Footnote 42 pypi.python.org/pypi/Whoosh

Footnote 43 github.com/Mplsbeb/whoosh

TIP

In the code above we dropped the keys that were not found in the lexicon to avoid a divide-by-zero error. But a better approach is to +1 the denominator of every IDF calculation. This ensures no denominators are zero. In fact this approach is used in a lot of other "counting" problems and is called additive smoothing (Laplace smoothing).⁴⁴ will usually improve the search results for TF-IDF keyword-based searches.

Footnote 44 en.wikipedia.org/wiki/Additive_smoothing

Keyword search is just one tool in our NLP pipeline. We want to build a chatbot. But most chatbots rely heavily on a search engine. And some chatbots rely exclusively on their search engine as their only algorithm for generating responses. We just need to take one additional step to turn our simple search index (TF-IDF) into a chatbot. We need to store our training data in pairs of questions (or statements) and appropriate responses. Then we can use TF-IDF to search for a question (or statement) most like the user input text. Instead of returning the most similar statement in our database, we return the response associated with that statement. Like any tough computer science problem, ours can be solved with one more layer of indirection. And with that, we're chatting!

3.4.3 Tools

Now that was a lot of code for things that have long since been automated. A quick path to the same result can be found using the `scikit-learn` package.⁴⁵ If you haven't already set up your environment using Appendix A so that it includes this package, here's one way to install it.

Footnote 45 scikit-learn.org/

```
pip install scipy
pip install sklearn
```

Here's how you can use `sklearn` to build a TF-IDF matrix. The `sklearn` TF-IDF class is a *model* with `.fit()` and `.transform()` methods that comply with the `sklearn` API for all machine learning models.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

<https://forums.manning.com/forums/natural-language-processing-in-action>
Licensed to Francesco Luzio <f.luzio@converge.it>

```
>>> corpus = docs
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> model = vectorizer.fit_transform(corpus) ❶
>>> print(model.todense().round(2))
[[ 0.16 0.    0.48 0.21 0.21 0.   0.25 0.21 0.   0.   0.   0.21 0.   0.64 0.21 0.21]
 [ 0.37 0.    0.37 0.   0.   0.37 0.29 0.   0.37 0.37 0.   0.   0.49 0.   0.   0.   ]
 [ 0.   0.75 0.   0.   0.29 0.22 0.   0.29 0.29 0.38 0.   0.   0.   0.   0.   ]]
```

- ❶ The TFIDFVectorizer model is a sparse numpy matrix, because a TF-IDF contains mostly zeros since most documents use a small portion of the total words in the vocabulary
- ❷ The .todense() method converts a sparse matrix back into a regular numpy matrix (filling in the gaps with zeros) for our viewing pleasure.

So with `scikit-learn` in 4 lines we created a matrix of our 3 documents and the Inverse Document Frequency for each term in the lexicon. We have a matrix (practically a list of lists in Python) that represents the 3 documents (the 3 rows of the matrix) and the TF-IDF of each term, token, or word in our lexicon make up the columns of the matrix (or again, the indices of each row). They only have 16, as they tokenize differently and drop the punctuation, we had a comma and a period. On large texts this or some other pre-optimized TF-IDF model will save you scads of work.

3.4.4 Alternatives

TF-IDF matrices (term-document matrices) have been the mainstay of information retrieval (search) for decades. As a result, researchers and corporations have spent a lot of time trying to optimize that IDF part to try to improve the relevance of search results. Here's a list of some of the ways you can normalize and smooth your term frequency weights.

| Scheme | Definition |
|------------|--|
| None | $w_{ij} = f_{ij}$ |
| TF-IDF | $w_{ij} = \log(f_{ij}) \times \log(\frac{N}{n_j})$ |
| TF-ICF | $w_{ij} = \log(f_{ij}) \times \log(\frac{N}{f_j})$ |
| Okapi BM25 | $w_{ij} = \frac{f_{ij}}{0.5 + 1.5 \times \frac{f_j}{\sum_j f_j} + f_{ij}} \log \frac{N - n_j + 0.5}{f_{ij} + 0.5}$ |
| ATC | $w_{ij} = \frac{(0.5 + 0.5 \times \frac{f_{ij}}{\max_f}) \log(\frac{N}{n_j})}{\sqrt{\sum_{i=1}^N [(0.5 + 0.5 \times \frac{f_{ij}}{\max_f}) \log(\frac{N}{n_j})]^2}}$ |
| LTU | $w_{ij} = \frac{(\log(f_{ij}) + 1.0) \log(\frac{N}{n_j})}{0.8 + 0.2 \times f_j \times \frac{j}{f_j}}$ |
| MI | $w_{ij} = \log \frac{P(t_{ij} c_j)}{P(t_{ij})P(c_j)}$ |
| PosMI | $\max(0, MI)$ |
| T-Test | $w_{ij} = \frac{P(t_{ij} c_j) - P(t_{ij})P(c_j)}{\sqrt{P(t_{ij})P(c_j)}}$ |
| χ^2 | see (Curran, 2004, p. 83) |
| Lin98a | $w_{ij} = \frac{f_{ij} \times f}{f_i \times f_j}$ |
| Lin98b | $w_{ij} = -1 \times \log \frac{n_j}{N}$ |
| Gref94 | $w_{ij} = \frac{\log f_{ij} + 1}{\log n_j + 1}$ |

Figure 3.15 Alternative TFIDF Normalization Formulae from Word Embeddings Past, Present and Future by Piero Molino at AI with the Best 2017

Search engines (information retrieval systems) match keywords (term) between queries and documents in a corpus. If you're building a search engine and want to provide documents that are likely to match what your users are looking for, then you should spend some time investigating the alternatives described by Piero Molino above.

One such alternative to using straight TF-IDF cosine distance to rank query results is Okapi BM25, or its most recent variant, BM25F.

3.4.5 Okapi BM25

The smart people at London's City University came up with a better way to rank search results. Rather than merely computing the TF-IDF cosine similarity, they normalize and smooth the similarity. They also ignore duplicate terms in the query document, effectively clipping the term frequencies for the query vector at 1. And the dot product for the cosine similarity is not normalized by the TF-IDF vector norms (number of terms in the document and the query), but rather by a nonlinear function of the document length itself.

```
q_idf * dot(q_tf, d_tf[i]) * 1.5 / (dot(q_tf, d_tf[i]) + .25 + .75 * d_num_words[i] / d_num_words.mean())
```

You can optimize your pipeline by choosing the weighting scheme that gives your users the most relevant results. But if your corpus isn't too large you might consider forging ahead with us into even more useful and accurate representations of the meaning of words and documents. In subsequent chapters we are going to show you how to implement a semantic search engine that finds documents that "mean" something similar to the words in your query rather than just documents that use those exact words from your query. Semantic search is much better than anything TF-IDF weighting and stemming and lemmatization can ever hope achieve. The only reason Google and Bing and other web search engines don't use the semantic search approach is that their corpus is too large. Semantic word and topic vectors don't scale to billions of documents, but millions of documents are no problem.

So you only need the most basic TF-IDF vectors to feed into our pipeline to get state-of-the-art performance for semantic search, document classification, dialog systems, and most of the other applications we mentioned in Chapter 1. TFIDFs are just the first stage in our pipeline, the most basic set of features we'll extract from text. In the next chapter we'll compute topic vectors from our TF-IDF vectors. Topic vectors are an even better representation of the meaning of the content of a bag of words than any of these carefully-normalized and smoothed TF-IDF vectors. And things only get better from there as we move on to Word2vec word vectors in Chapter 6 and neural net embeddings of the meaning of words and documents in later chapters.

3.4.6 Summary

In this chapter, we got our feet wet with some basic word arithmetic.

- Counting word occurrences in a document
- Regularizing the counts with reference to the document length

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

- Saw the strengths and weaknesses of using counts for finding meaning in a document
- Probed deeper for meaning by examining the occurrence of word not only in a document, but across the whole corpus
- Represented a word as a topic vector and found similar topics(vectors) in the corpus.

Now that you can convert natural language text to numbers you can begin to manipulate them, compute with them. Numbers firmly in hand, let's now refine those numbers to try to represent the *meaning* or *topic* of natural language text instead of just its words.

Finding Meaning in Word Counts (Semantic Analysis)



In this chapter

- Analyzing the semantics (meaning) of a document
- Creating topic vectors for documents and words
- Estimating the semantic similarity between documents or words
- Implementing semantic search
- Scalable semantic analysis on large corpora
- Using semantic topics as features in your NLP pipeline
- Navigating high-dimensional vector spaces

You've learned quite a few natural language processing tricks. But now may be the first time you'll be able to do a little bit of magic. This will be the first time we talk about a machine beginning able to understand the "meaning" of words.

Our TF-IDF vectors (Term Frequency Inverse Document Frequency vectors) from Chapter 3 helped us estimate the importance of words in a chunk of text. We used TF-IDF vectors and matrices to tell us how important each word is to the overall meaning of a bit of text in a collection of documents.

And these TF-IDF "importance" scores worked not only for words, but also for short sequences of words, *n*-grams. These importance scores for *n*-grams are great for searching text if you know the exact words or *n*-grams you are looking for.

In this chapter we will learn about *semantic* or *topic* vectors.⁴⁶ We're going to use our weighted frequency scores from TF-IDF vectors to compute the topic "scores" that make

up the dimensions of our topic vectors. But we're going to use the correlation of those frequency scores with each other to group words together in topics to define the dimensions of our new _topic vector_s.

Footnote 46 We will generally use the more descriptive term "topic vector" in place of "word vector" that is more common in formal NLP texts like [the NLP bible by Jurafsky and Martin](#). Others, like the authors of [Semantic Vector Encoding and Similarity Search](#) use the term "semantic vector." We're just trying to keep things simple.

These topic vectors will help us do a lot of interesting things. They make it possible to search for documents based on their meaning. This is called *semantic search*. Most of the time semantic search will return search results that are much better than keyword search (TF-IDF search). Sometimes semantic search will return documents that are exactly what the user is searching for, even when they can't think of the right words to search for.

And we can use these semantic vectors to identify the words and *n*-grams that best represent the subject (topic) of a statement, document, or corpus (collection of documents). And with this vector of words and their relative importance you can provide someone with the most meaningful words for a document, a set of keywords that summarizes the meaning of a document.

And we can now compare any two statements or documents and tell how "close" they are in *meaning* to each other.

TIP

The terms "topic", "semantic", and "meaning" have similar meaning and are often used interchangeably when talking about NLP. In this chapter you're learning how to build an NLP pipeline that can figure this kind of synonymy out, all on its own. You pipeline might even be able to find the similarity in meaning of the phrase "figure it out" and the word "compute". Machines, of course, can only "compute" meaning, not "figure out" meaning.

You'll soon see that the linear combinations of words that make up the dimensions of our topic vectors are pretty powerful representations of meaning.

4.1 From Word Counts to Topic Scores

You know how to count the frequency of words. And you know how to score the importance of words in a TF-IDF vector or matrix. But that's not enough. We want to score the meanings, the topics, that words are used for.

4.1.1 TF-IDF Vectors and Lemmatization

TF-IDF vectors count the exact spellings of terms in a document. So texts that restate the same meaning will have completely different TF-IDF vector representations if they spell things differently or use different words. This will mess up search engines and document similarity comparisons that rely on counts of tokens.

In Chapter 2 we normalized word spellings so that words with similar spelling would be lumped together into a single term or word. We used normalization approaches like *stemming* and *lemmatization* to create small collections of words with similar spellings, and often similar meanings. We labeled each of these small collections of words, with their lemma or stem and then we processed these labels (stems or lemmas) instead of the original spellings.

This lemmatization approach kept similarly *spelled* words together in our analysis, but not necessarily words with similar meanings. And it definitely failed to pair up most synonyms. Synonyms rarely have similar spellings. This lemmatization approach can even sometimes erroneously lump together antonyms, words with opposite meaning.

The end result is that two chunks of text that talk about the same thing but use different words will not be very "close" to each other in our lemmatized TF-IDF vector space model. And sometimes two lemmatized TF-IDF vectors that are close to each other aren't very similar in meaning at all. Even a state-of-the-art TF-IDF similarity score from Chapter 3, like Okapi BM25 or cosine similarity, would fail to connect these synonyms or push apart these antonyms. Synonyms with different spellings produce TF-IDF vectors that just aren't close to each other in the vector space.

For example, the cosine similarity between the TF-IDF vector for this chapter in *NLP1A*, the chapter that you're reading right now, may not be at all close to similar-meaning passages in university textbooks about "Latent Semantic Analysis (LSA)." LSA is exactly what this chapter is about. But we use modern and colloquial terms in this chapter. Professors and researchers use more consistent, rigorous language in their textbooks and lectures. Plus, the terminology that professors used a decade ago has likely evolved with the rapid advances of the past few years. For example, terms like "latent semantic *indexing*" were more popular than the term "latent semantic analysis" that researchers now use.⁴⁷

Footnote 47 I love Google NGram Viewer for [visualizing trends like this](#)

4.1.2 Topic Vectors

When we do math on TF-IDF vectors, like addition and subtraction, these sums and differences only tell us about the frequency of word uses in the documents whose vectors we combined or differenced. That math doesn't tell us much about the "meaning" behind those words. You can compute word-to-word TF-IDF vectors (word co-occurrence or correlation vectors) by just multiplying our TF-IDF matrix by itself. But "vector reasoning" with these sparse, high dimensional vectors doesn't work very well. You when you add or subtract these vectors from each other, they don't represent an existing concept or word or topic very well.

So we need a way to extract some additional information, meaning, from word statistics. We need a better estimate of what the words in a document "signify." And we need to know what that combination of words *means* in a particular document. We'd like to represent that meaning with a vector that's like a TF-IDF vector, only more compact and more meaningful.

NLP experimenters before us have found an algorithm for revealing the meaning of word combinations and computing these compact and meaningful vectors. It's called *Latent Semantic Analysis (LSA)*. And when we use this tool, not only can we represent the meaning of words as vectors, but we can use them to represent the meaning of entire documents.

We call these compact meaning vectors "word-topic vectors." We call the document meaning vectors "document-topic vectors." You can call either of these vectors "topic vectors," as long ans you're clear on what the topic vectors are for, words or documents.

These topic vectors can be as compact or as expansive (high-dimensional) as you like. LSA topic vectors can have as few as 1 dimension, or they can have thousands of dimensions.

The topic vectors you'll compute in this chapter can be added and subtracted just like any other vector. Only this time the sums and differences will mean a lot more than they did with TF-IDF vectors (Chapter 3). And the distances between topic vectors will be useful for things like clustering documents or semantic search. Before, we could cluster and search using keywords and TF-IDF vectors. Now we can cluster and search using semantics, meaning!

When we're done, you'll have one document-topic vector for each document in your corpus. And, even more importantly, you won't have to reprocess the entire corpus to

compute a new topic vector for a new document or phrase.

TIP

Some algorithms for creating topic vectors, like Latent Dirichlet Allocation, do require you to reprocess the entire corpus, every time you add a new document.]

You'll have one word-topic vector for each word in your lexicon (vocabulary). So you can compute the topic vector for any new document by just adding up all its word topic vectors.

Coming up with a numerical representation of the semantics (meaning) of words and sentences can be tricky. This is especially true for "fuzzy" languages like English, which has multiple dialects and many different interpretations of the same words. Even formal English text written by your English professor can't avoid the fact that most English words have multiple meanings, a challenge for any new learner, including machine learners. This concept of words with multiple meanings is called *Polysemy*.

- *Polysemy*
The existence of words and phrases with more than one meaning

Here are some ways in which polysemy can affect the semantics of a word or statement. We're just listing them here for you to appreciate the power of Latent Semantic Analysis (LSA). You don't have to worry about these challenges. LSA is going to take care of all this for us:

- *Homonyms*
Words with the same spelling and pronunciation, but different meanings
- *Zeugma*
Use of two meanings of a word simultaneously in the same sentence

And LSA also deals with some of the challenges of polysemy in a voice interface—a chatbot that you can talk to, like Alexa or Siri:

- *Homographs*
Words spelled the same, but with different pronunciations and meanings
- *Homophones*
Words with the same pronunciation, but different spellings and meanings (an NLP challenge with voice interfaces)

Imagine if we had to deal with a statement like the following, if we didn't have tools like

Latent Semantic Analysis to deal with it:

She felt ... less. She felt tamped down. Dim. More faint. Feint. Feigned. Fain.
-- Patrick Rothfuss The Slow Regard of Silent Things

Keeping these challenges in mind, can you imagine how we might squash a TF-IDF vector with one million dimensions (terms) down to a vector with 200 or so dimensions (topics)? This is like identifying the right mix of primary colors to try to reproduce the color of the paint in your apartment so you can cover over those nail holes in your wall.

We'd need to find those word dimensions that "belong" together in a topic and add their TF-IDF values together to create a new number to represent the amount of that topic in a document. We might even weight them for how important they are to the topic, how much we'd like each word to contribute to the "mix." And we could have negative weights for words that reduce the likelihood that the text is about that topic.

4.1.3 Thought Experiment

Let's walk through a thought experiment. Let's assume we have some TF-IDF vector for a particular document and we want to convert that to a topic vector. We can think about how much each word contributes to our topics.

Just for our simple thought experiment, let's say we're processing some sentences about pets in Central Park in New York City. Let's create three topics: one about pets, one about animals, and another about cities. We'll call these topics "petness", "animalness", and "cityness." So our "petness" topic about pets will score words like "cat" and "dog" significantly, but probably ignore words like "NYC" and "apple." Our "cityness" topic will ignore words like "cat" and "dog" but might give a little weight to "apple", just because of the "Big Apple" association.

If we "trained" our topic model like this, without using a computer, just our common sense, we might come up with some weights like this:

```
>>> topic = {}

>>> tfidf = dict(list(zip('cat dog apple lion NYC love'.split(), np.random.rand(6)))) ①

>>> topic['petness']      = (.3 * tfidf['cat'] + .3 * tfidf['dog'] + 0 * tfidf['apple']
                           + 0 * tfidf['lion'] - .2 * tfidf['NYC'] + .2 * tfidf['love'])
>>> topic['animalness'] = (.1 * tfidf['cat'] + .1 * tfidf['dog'] - .1 * tfidf['apple']
                           + .5 * tfidf['lion'] + .1 * tfidf['NYC'] - .1 * tfidf['love'])
>>> topic['cityness']    = (0 * tfidf['cat'] - .1 * tfidf['dog'] + .2 * tfidf['apple']
                           - .1 * tfidf['lion'] + .5 * tfidf['NYC'] + .1 * tfidf['love']) ②
```

- ① This tfidf vector is just a random example, as if it were computed for a single document that contained these words in some random proportion.
- ② "Hand-crafted" weights (.3, .3, 0, 0, -.2, .2) are multiplied by imaginary tfidf values to create topic vectors for our imaginary random document. We'll compute real topic vectors further down.

In this thought experiment we added up the word frequencies that might be indicators of each of our topics. We weighted the word frequencies (TF-IDF values) by how likely the word is associated with a topic. We did the same, but subtracted, for words that might be talking about something that is in some sense the opposite of our topic. This is not a real algorithm walk-through, or example implementation, just a thought experiment. We're just trying to figure out how we can teach a machine to think like we do. We arbitrarily chose to decompose our words and documents into only three topics ("petness", "animalness", and "cityness"). And our vocabulary is very limited, it only has six words in it.

The next step is to think through how a human might actually decide mathematically which topics and words are connected, and what weights those connections should have. Once we decided on three topics to model, we then had to then decide how much to weight each word for those topics. We blended words in proportion to each other to make our topic "color mix." Our topic modeling transformation (color mixing recipe) was a 3×6 matrix of proportions (weights) connecting three topics to six words. We multiplied that matrix by an imaginary 6×1 TF-IDF vector to get a 3×1 topic vector for that document.

In the thought experiment above we made a judgment call that the terms "cat" and "dog" should have similar contributions to the "petness" topic (weight of .3). So the two values in the upper left of the matrix for our TF-IDF-to-topic transformation are both .3. As you think through this thought experiment, can you imagine ways we might "compute" these proportions with software? Remember, you have a bunch of documents that your computer can read, tokenize and count tokens for. You have TF-IDF vectors for as many documents as you like. Keep thinking about how you might use those counts to compute topic weights for a word as you read on.

We decided that the term "NYC" should have a negative weight for the "petness" topic. In some sense city names, and proper names in general, and abbreviations, and acronyms, share very little in common with words about pets. Think about what "sharing in common" means for words. Is there something in a TF-IDF matrix that represents the meaning that words share in common?

We gave the word "love" a positive weight for the "pets" topic. This may be because we often use the word "love" in the same sentence with words about pets. After all, we humans tend to love our pets. We can only hope that our AI overlords will be similarly loving towards us humans.

Notice that we put a small amount of the word "apple" into the topic vector for "city." This could be because we're doing this by hand and we humans know that "NYC" and "Big Apple" are often synonymous. Our semantic analysis algorithm will hopefully be able to calculate this synonymy between "apple" and "NYC" based on how often "apple" and "NYC" occur in the same documents.

As you read the rest of the weighted sums in the example "code" above, try to guess how we came up with these weights for these three topics and six words. How might you change them? What could we use as an objective measure of these proportions (weights)? You may have a different "corpus" in your head than the one we used in our heads. So you may have different a different opinion about these words and the weights we gave them. What could we do something to come to a consensus about our opinions about these six words and three topics?

NOTE

We chose a signed weighting of words to produce our topic vectors. This allows us to use negative weights for words that are the "opposite" of a topic. And since we're doing this manually by hand, we chose to normalize our topic vectors by the easy-to-compute L1-norm (Manhattan, taxicab, or city-block distance). Nonetheless, the real Latent Semantic Analysis (LSA) you'll use later in this chapter normalizes topic vectors by the more useful L2-norm. L2-norm is the conventional Euclidean distance or length that you're familiar with from geometry class. It's the Pythagorean theorem solved for the length of the hypotenuse of a right triangle.

You might have realized in reading these vectors that the relationships between words and topics can be "flipped." Our 3×6 matrix of three topic vectors above can be transposed to produce topic weights for each word in our vocabulary. These vectors of weights would be our word vectors for our six words:

```
>>> word_vector['cat'] = .3*topic['petness'] +.1*topic['animalness'] + 0*topic['cityness']
>>> word_vector['dog'] = .3*topic['petness'] +.1*topic['animalness'] -.1*topic['cityness']
>>> word_vector['apple']= 0*topic['petness'] -.1*topic['animalness'] +.2*topic['cityness']
>>> word_vector['lion'] = 0*topic['petness'] +.5*topic['animalness'] -.1*topic['cityness']
>>> word_vector['NYC'] = -.2*topic['petness'] +.1*topic['animalness'] +.5*topic['cityness']
>>> word_vector['love'] = .2*topic['petness'] -.1*topic['animalness'] +.1*topic['cityness']
```

These six topic vectors, one for each word, represent the meanings of our six words as 3D vectors.

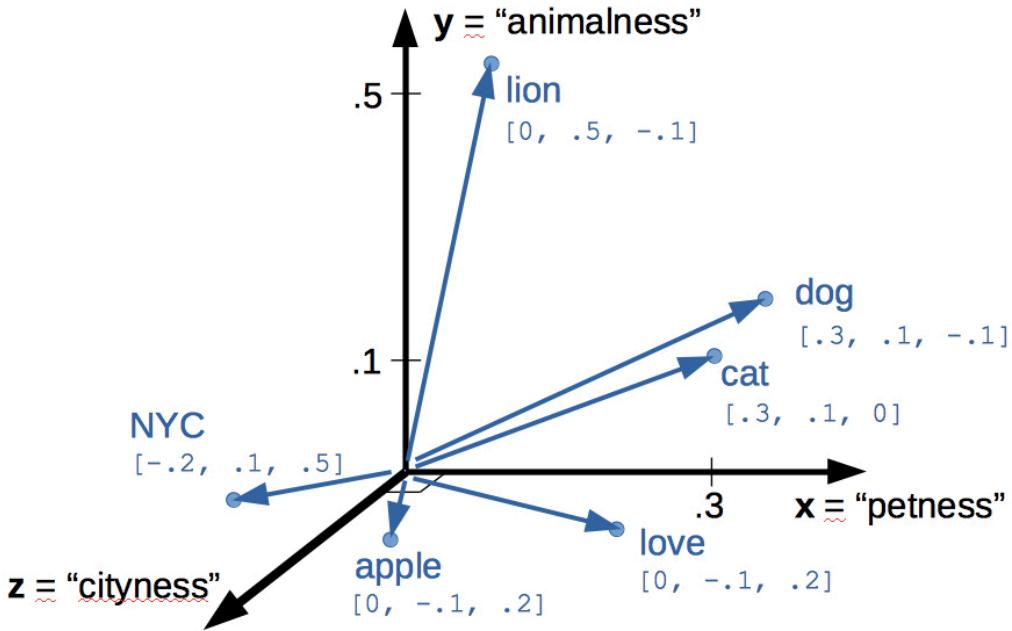


Figure 4.1 3D vectors for a thought experiment about six words about pets and NYC

Earlier, the vectors for each topic, with weights for each word, gave us 6-D vectors representing the linear combination of words in our three topics. In our thought experiment we hand-crafted a three-topic model for a single natural language document! If we just count up occurrences of these six words and multiply them by our weights we get the 3D topic vector for any document. And 3D vectors are fun because they're easy for us humans to visualize. We can plot them and share insights about our corpus or a particular document in graphical form.

And 3D vectors (or any low-dimensional vector space) are great for machine learning classification problems too. An algorithm can slice through the vector space with a plane (or hyperplane) to divide up the space into classes.

The documents in our corpus might use many more words, but this particular topic vector model will only be influenced by the use of these six words. We could, of course, extend this approach to as many words as we had the patience (or an algorithm) for. As long as our model only needed to separate documents according to three different dimensions or

topics, our vocabulary could keep growing as much as we like. In the thought experiment above we compressed six dimensions (TF-IDF normalized frequencies) into three dimensions (topics).

This subjective, labor-intensive approach to semantic analysis relies on human intuition and common sense to break documents down into topics. Common sense is hard to code into an algorithm.⁴⁸ So it isn't repeatable—you'd probably come up with different weights than we did. And obviously this isn't suitable for a machine learning pipeline. Plus it doesn't scale well to more topics and words. A human couldn't allocate enough words to enough topics to precisely capture the meaning in any diverse corpus of documents that we might want our machine to deal with.

Footnote 48 Doug Lenat at Stanford is trying to do just that (code common sense into an algorithm):
www.wired.com/2016/03/doug-lenat-artificial-intelligence-common-sense-engine/

So let's automate this manual procedure. Let's use an algorithm that doesn't rely on common sense to select topic weights for us.⁴⁹

Footnote 49 The wikipedia page for topic models has a video which shows how this might work for many more topics and words. The darkness of the pixels represents the weight or value or score for a topic and a word, like the weights in our manual example above. And the video shows a particular algorithm, called SVD, that reorders the words and topics, to put as much of the "weight" as possible along the diagonal. This helps identify patterns that represent the meanings of both the topics and the words.

upload.wikimedia.org/wikipedia/commons/7/70/Topic_model_scheme.webm#t=00:00:01,00:00:17.600

If you think about it, each of these weighted sums is just a dot product. And three dot products (weighted sums) is just a matrix multiplication, or inner product. You multiply a $3 \times n$ weight matrix with a TF-IDF vector (one value for each word in a document), where n is the number of terms in our vocabulary. The output of this multiplication is a new 3×1 topic vector for that document. What we've done is "transform" a vector from one vector space (TF-IDFs) to another lower dimensional vector space (topic vectors). Our algorithm should create a matrix of n terms by m topics that we can multiply by a vector of the word frequencies in a document to get our new topic vector for that document.

NOTE

In mathematics, the size of a vocabulary (the set of all possible words in a language) is usually written as $|V|$. And the variable V alone is used to represent the set of possible words in your vocabulary. So if you're writing an academic paper about NLP, you'll want to use $|v|$ wherever I've used n to describe the size of a vocabulary.

4.1.4 An Algorithm for Scoring Topics

We still need an algorithmic way to determine these topic vectors. We need a transformation from TF-IDF vectors into topic vectors. A machine can't tell which words belong together or what any of them signify, can it? J. R. Firth, a 20th century British linguist, studied the ways we can estimate what a word⁵⁰ signifies. In 1957 he gave us a clue about how to compute the topics for words. Firth wrote:

Footnote 50 or *morpheme*—the smallest meaningful parts of words: en.wikipedia.org/wiki/Morpheme

You shall know a word by the company it keeps.

-- J. R. Firth 1957

So how do we tell the "company" of a word? Well, the most straightforward approach would be to count co-occurrences in the same document. And we've got exactly what we need for that in our Bag-of-Word and TF-IDF vectors from Chapter 3. This "counting co-occurrences" approach led to the development of several algorithms for creating vectors to represent the statistics of word usage within documents or sentences.

The most common technique for creating these vectors to represent the "meaning" is called Latent Semantic Analysis (LSA). Latent Semantic Analysis is an algorithm to analyze your TF-IDF matrix (table of TF-IDF vectors) to gather up words into topics. It will work on Bag-of-Words vectors too. But TF-IDF vectors will give us slightly better results.

LSA also optimizes these topics to maintain diversity in the topic dimensions. This is so when you use these new topics instead of the original words, you still capture much of the meaning (semantics) of the documents. The number of topics you need for your model to capture the meaning of your documents is far less than the number of words in the vocabulary of your TF-IDF vectors. So LSA is often referred to as a dimension reduction technique. LSA reduces the number of dimensions you need to capture the meaning of your documents.

Have you ever used a dimension reduction technique for a large matrix of numbers? What about pixels? If you've done machine learning on images or other high dimensional data you may have run across a technique called Principal Component Analysis (PCA). As it turns out, PCA is exactly the same math as LSA. It's just that PCA is what we say when we're reducing the dimensionality of images or other tables of numbers, rather than bag-of-words vectors or TF-IDF vectors.

Only recently did researchers discover that you could use PCA for semantic analysis of words. That's when they gave this particular application its own name, Latent Semantic Analysis (LSA). So, even though you'll see the `scikit-learn` PCA model used below, the output of this fit and transform process is a vector representing the semantics of a document. It's still LSA.

And here's one more synonym for LSA that you may run across. In the field of information retrieval, where the focus is on creating indexes for full text search, Latent Semantic Analysis is often referred to as Latent Semantic Indexing (LSI). But this term has fallen out of favor. It doesn't really produce an index at all. In fact, the topic vectors it produces are usually too high dimensional to ever be indexed perfectly. So we'll use the term "LSA" from here on out.

TIP

Indexing is what databases do to be able to retrieve a particular row in a table quickly based on some partial information you provide it about that row. The index at the back of a text book works like this. If you are looking for a particular page, you can look up words in the index that should be on the page. Then you can go straight to the page or pages that contain all of the words you're looking for.

LSA "COUSINS"

There are two algorithms that are similar to LSA, with similar NLP applications, so we'll mention them here.

- Linear Discriminant Analysis (LDA)
- Latent Dirichlet Allocation (LDiA)⁵¹

Footnote 51 Since "LDA" is already "taken" for Linear Discriminant Analysis we use LDiA for Latent Dirichlet Allocation. Perhaps the instructors for this Stanford online CS class on LSA and LDiA would approve: cs.stanford.edu/~ppasupat/a9online/1140.html#latent-dirichlet-allocation-lda-

Linear Discriminant Analysis (LDA) breaks down a document into only one topic. Latent Dirichlet Allocation (LDiA) is more like LSA because it can break down documents into as many topics as you like.

TIP

Because it's one dimensional, Linear Discriminant Analysis (LDA) doesn't require SVD. You can just compute the centroid (average or mean) of all your TF-IDF vectors for each side of a binary class, like spam and non-spam. Your dimension then becomes the line between those two centroids. The further a TF-IDF vector is along that line (the dot product of the TF-IDF vector with that line) tells you how close you are to one class or another.

Here's an example of this simple LDA approach to topic analysis first, to get you warmed up before we tackle LSA and LDIA.

4.1.5 An LDA Classifier

Linear Discriminant Analysis (LDA) is one of the most straightforward and fastest dimension reduction and classifier model you'll find. LDA is a supervised algorithm, so you need labels for your messages.

For this example we'll show you simplified version of LDA from what you'll find in `scikit-learn`. We only need to do three things, so we'll just do them all directly in python.

1. compute the average position (centroid) of all the TF-IDF vectors within the class (e.g. spam SMS messages)
2. compute the average position (centroid) of all the TF-IDF vectors not in the class (e.g. non-spam SMS messages)
3. compute the vector difference between the centroids (the line that connects them)

All you need to "train" an LDA model is to find the vector (line) between the two centroids for your binary class. To do *inference* or prediction with that model we just need to find out if a new TF-IDF vector is closer to the in-class (spam) centroid than it is to the out-of-class (non-spam) centroid along that line. First let's "train" an LDA model to classify SMS messages as spam or non-spam:

Listing 4.1 The SMS spam dataset

```
>>> import pandas as pd
>>> from nlpia.data.loaders import get_data
>>> pd.options.display.width = 120
1
>>> sms = get_data('sms-spam')
2
>>> index = ['sms{}{}'.format(i, ''*j) for (i,j) in zip(range(len(sms)), sms.spam)]
>>> sms = pd.DataFrame(sms.values, columns=sms.columns, index=index)
>>> sms['spam'] = sms.spam.astype(int)

>>> len(sms)
4837
```

```
>>> sms.spam.sum()
638
>>> sms.head(6)
   spam                                text
sms0    0  Go until jurong point, crazy.. Available only ...
sms1    0          Ok lar... Joking wif u oni...
sms2!   1  Free entry in 2 a wkly comp to win FA Cup fina...
sms3    0  U dun say so early hor... U c already then say...
sms4    0  Nah I don't think he goes to usf, he lives aro...
sms5!   1  FreeMsg Hey there darling it's been 3 week's n...
```

- ➊ This helps display the wide column of SMS text within a Pandas DataFrame printout.
- ➋ We've flagged spam messages by appending an exclamation point, "!", to their label.

So we have 4,837 SMS messages and 638 of the messages are labeled with the binary class label "spam."

Now let's do our tokenization and TF-IDF vector transformation on all these SMS messages.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from nltk.tokenize.casual import casual_tokenize
>>> tfidf_model = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf_model.fit_transform(raw_documents=sms.text).toarray()
>>> tfidf_docs.shape
(4837, 9232)
>>> sms.spam.sum()
638
```

The `nltk.casual_tokenizer` gave us 9,232 words in our vocabulary. We have almost twice as many words as we have messages. And we have almost ten times as many words as *spam* messages. So our model will not have a lot of information about the words that will indicate whether a message is spam or not. A Naive Bayes usually will not work well when your vocabulary is much larger than the number of labeled examples in your dataset. That's where the semantic analysis techniques of this chapter can help.

Let's start with the simplest semantic analysis technique, Linear Discriminant Analysis (LDA). We could use the LDA model in `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`. Be we only need compute the centroids of our binary class (spam and non-spam) in order to "train" this model, so let's do that directly.

```
>>> mask = sms.spam.astype(bool)           ➊
>>> spam_centroid = tfidf_docs[mask].mean(axis=0)  ➋
>>> ham_centroid = tfidf_docs[~mask].mean(axis=0)
```

```
>>> spam_centroid.round(2)
array([0.06, 0. , 0. , ..., 0. , 0. , 0. ])
>>> ham_centroid.round(2)
array([0.02, 0.01, 0. , ..., 0. , 0. , 0. ])
```

- ➊ We can use this mask to select only the spam rows from a numpy.array or pandas.DataFrame
- ➋ Because our TF-IDF vectors are row vectors we need to make sure numpy computes the mean for each column independently using axis=0.

Now we can subtract one centroid from the other to get the line between them.

```
>>> spamminess_score = tfidf_docs.dot(spam_centroid - ham_centroid) ➊
>>> spamminess_score.round(2)
array([-0.01, -0.02,  0.04, ..., -0.01, -0. ,  0. ])
```

- ➊ The dot product computes the "shaddow" or projection of each vector on the line between the centroids.

This raw `spamminess_score` is the distance along the line from the ham centroid to the spam centroid. We calculated that score by projecting each TF-IDF vector onto that line between the centroids using the dot product. And we did those 4,837 dot products all at once in a "vectorized" numpy operation. This can speed things up 100 times compared to a python loop.

Here's a view of the TF-IDF vectors in 3D and where these centroids are for our SMS messages:

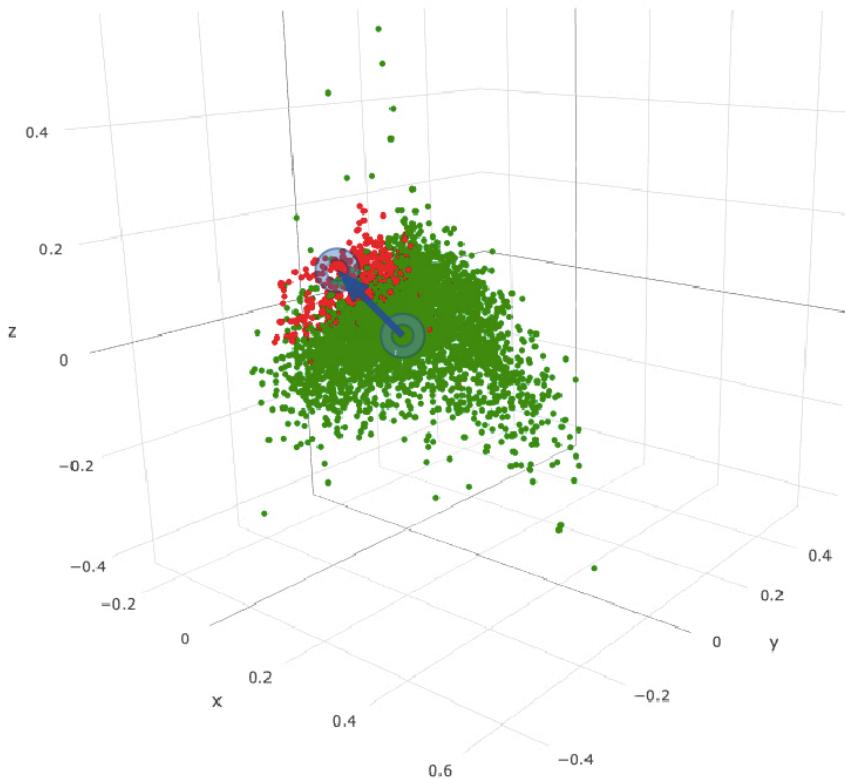


Figure 4.2 3D scatter plot (point cloud) of our TF-IDF vectors

The blue arrow from the green non-spam centroid to the red spam centroid is the line that defines our trained model. You can see how some of the green dots are on the back side of the arrow, so you could get a negative spamminess score when you project them onto this line between the centroids.

Ideally, we'd like our score to range between 0 and 1, like a probability. The `sklearn MinMaxScaler` can do that for us:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> sms['lda_score'] = MinMaxScaler().fit_transform(spamminess_score.reshape(-1,1))
>>> sms['lda_predict'] = (sms.lda_score > .5).astype(int)
>>> sms[['spam lda_predict lda_score']].round(2).head(6)
   spam  lda_predict  lda_score
sms0      0          0     0.23
sms1      0          0     0.18
sms2!     1          1     0.72
sms3      0          0     0.18
sms4      0          0     0.29
sms5!     1          1     0.55
```

That looks pretty good. All of the first 6 messages were classified correctly when we set the threshold at 50%. Let's see how it did on the rest of the training set.

```
>>> (1. - (sms.spam - sms.lda_predict).abs().sum() / len(sms)).round(3)
0.977
```

Wow! 97.7% of the messages were classified correctly with this very simple model. We're not likely to achieve this in the real world, since we haven't separated our test set from our training set for cross validation. But LDA is such a simple model, with very few parameters, so it should generalize pretty well, as long as our SMS messages are representative of the messages we intend to filter. Try it on your own examples to find out.

This is the power of semantic analysis approaches. Unlike Naive Bayes or logistic regression models, semantic analysis doesn't rely on individual words. Semantic analysis gathers up words with similar semantics, in our case all the words with spammy connotation.

However, remember that this training set has a limited vocabulary and some non-English words in it. So your test messages need to use similar words if you want them to be classified correctly.

Lets see what the training set confusion matrix looks like. This will show us the SMS messages that it labeled as spam that were actually not (false positives) and the ones that were labeled as ham that were actually spam (false negatives):

```
>>> from pugnlp.stats import Confusion
>>> Confusion(sms['spam lda_predict'].split())
lda_predict      0      1
spam
0            4135    64
1            45    593
```

That looks nice. We could adjust the 0.5 threshold on our score if the false positives (64) or false negatives (45) were out of balance. Now you're ready to learn about models that can compute *multidimensional* semantic vectors instead just 1D semantic scores. So far, the only thing our 1D vectors "understand" is the spamminess of words and documents. We'd like them to learn a lot more of the nuance of words and give us a multidimensional vector that captures that meaning.

Before we dive into SVD, the math behind multidimensional LSA, we should mention some other approaches first.

THE OTHER COUSIN

LSA (Latent Semantic Analysis) has another "cousin." And it has an abbreviation very similar to LDA (Linear Discriminant Analysis). "LDiA" stands for Latent Dirichlet Allocation.⁵² LDiA can also be used to generate vectors that capture the semantics of a word or document.

Footnote 52 We've chosen the nonstandard LDiA acronym to distinguish it from the acronym "LDA." LDA usually means Linear Discriminant Analysis, but not always. At least in this book, you won't have to guess what we mean by that algorithm. LDA will always mean Linear Discriminant Analysis. LDiA will always mean Latent Dirichlet Allocation.

Latent Dirichlet Allocation (LDiA) takes the math of Latent Semantic Analysis (LSA) in a different direction. LDiA uses a nonlinear statistical algorithm to group words together. Because LDiA must compute the statistics for an entire corpus during training, it cannot be used on new documents without rerunning the training.

This "all or none" challenge of LDiA makes it less practical for many real-world applications. Nonetheless, the statistics of the topics it creates more closely mirror human intuition about words and topics. So the topics often are easier to explain.

And LDiA is useful for some of single-document problems like document summarization. You corpus becomes the document, and your documents become the sentences in that "corpus." This is how `gensim` and other packages use LDiA to identify the most "central" sentences of a document. These sentences can then be strung together to create a machine-generated summary.

For most classification or regression problems you are usually better off using LSA. So we'll explain LSA and its underlying SVD linear algebra first.

LSA AND SVD ENHANCEMENTS

The success of Singular Value Decomposition (SVD) for semantic analysis and dimension reduction has motivated researchers to extend and enhance it. These enhancements are mostly intended for non-NLP problems, but we mention them here in case you run across them. They're sometimes used for behavior-based recommendation engines along side NLP content-based recommendation engines. And there's no reason why they couldn't be used for NLP semantic analysis.

- Quadratic Discriminant Analysis (QDA)
- Random Projection
- Nonnegative Matrix Factorization (NMF)

QDA is an alternative to Linear Discriminant Analysis (LDA). QDA creates quadratic polynomial transformations, rather than the linear transformations. These transformations define a vector space that can be used to discriminate between classes. And the boundary between classes in a QDA vector space is quadratic, curved, like a bowl or sphere or halfpipe.

Random Projection is a matrix decomposition and transformation approach similar to SVD, but the algorithm is stochastic. So you get a different answer each time you run it. But the stochastic nature makes it easier to run it on parallel machines. And in some cases (for some of those random runs) you can get transformations that are better than what comes out of SVD (and LSA). But Random Projection is rarely used for NLP problems. And there aren't widely used implementations of it in NLP packages like Spacy or NLTK. We'll leave it to you to explore this one further, if you think it might apply to your problem.

In most cases you're better off sticking with LSA which uses the tried and true SVD algorithm under the hood.⁵³

Footnote 53 SVD has traditionally been used to compute the "pseudo-inverse" of nonsquare matrices, and you can imagine how many applications there are for matrix inversion.

4.2 Latent Semantic Analysis (LSA)

Latent Semantic Analysis (LSA) is based on the oldest and most commonly-used technique for dimension reduction, Singular Value Decomposition (SVD). SVD was in widespread use long before the term "machine learning" even existed.⁵⁴ SVD decomposes a matrix into three square matrices, one of which is diagonal.

Footnote 54 Google NGram Viewer is a great way to learn about the history of words and concepts:
https://books.google.com/ngrams/graph?content=Machine+Learning%2CSVD%2CNatural+Language+Processing&year_start=1900&year_end=2000&corpus=50000

One application of SVD is matrix inversion. A matrix can be inverted by decomposing it into three simpler square matrices, transposing matrices, and then multiplying them back together. You can imagine all the applications for an algorithm that gives you a shortcut for inverting a large, complicated matrix. SVD is useful for mechanical engineering statics problems like truss structure stress and strain analysis. It's also useful for circuit analysis in electrical engineering. And it's even used in Data Science for behavior-based recommendation engines that run along side content-based NLP recommendation engines.

Using SVD, LSA can break down our TF-IDF term-document matrix into 3 simpler

matrices. And they can be multiplied back together to produce the original matrix, without any changes. This is like factorization of a large integer. Big whoop. But these three simpler matrices from SVD reveal properties about the original TF-IDF matrix that we can exploit to simplify it. We can truncate those matrices (ignore some rows and columns) before multiplying them back together. This reduces the number of dimensions we have to deal with in our vector space model.

These truncated matrices don't give the exact same TF-IDF matrix we started with. They give us a better one. Our new representation of the documents contains the essence, the "latent semantics" of those documents. That's why SVD is used in other fields for things like compression. It captures the essence of a dataset and ignores the noise. A JPEG image is ten times smaller than the original bitmap, but it still contains all the information of the original image.

When we use SVD this way in natural language processing, we call it Latent Semantic Analysis (LSA). LSA uncovers the semantics, or meaning, of words that is hidden and waiting to be uncovered.

Latent Semantic Analysis (LSA) is a mathematical technique for finding the "best" way to linearly transform (rotate and stretch) any set of NLP vectors, like our TF-IDF vectors or Bag-of-Words vectors. And the "best" way for many applications is to line up the axes (dimensions) in our new vectors with the greatest "spread" or variance in the word frequencies.⁵⁵ We can then eliminate those dimensions in the new vector space that don't contribute much to the variance in the vectors from document to document.

Footnote 55 There are some great visualizations and explanations in Chapter 16 of Jurafsky and Martin's [NLP textbook](#)

Using SVD to this way is called *Truncated Singular Value Decomposition* (Truncated SVD). In the image processing and image compression world, you might have heard of this as *Principle Component Analysis* (PCA). And we'll show you some tricks that helps improve the accuracy of LSA vectors. These tricks are also useful when you are doing PCA for machine learning and feature engineering problems in other areas.

If you've taken Linear Algebra you probably learned the algebra behind LSA called "Singular Value Decomposition" (SVD). And if you've done machine learning on images or other high-dimensional data, like time series, you've probably done something called "Principal Component Analysis" (PCA) to those high dimensional vectors. LSA on the natural language documents is equivalent to PCA on TF-IDF vectors.

LSA uses Singular Value Decomposition (SVD) to find the combinations of words that are responsible, together, for the biggest variation in the data. We can rotate our TF-IDF vectors so that the new dimensions (basis vectors) of our rotated vectors all align with these maximum variance directions. The "basis vectors" are the axes of our new vector space and are analogous to our topic vectors in the three 6-D topic vectors from our thought-experiment at the beginning of this chapter. Each of our dimensions (axes) becomes a combination of word frequencies rather than a single word frequency. So we think of them as the weighted combinations of words that make up various "topics" used throughout our corpus.

The machine doesn't really "understand" what the combinations of words means, just that they go together. When it sees words like "dog", "cat", and "love" together a lot, it puts them together in a topic. It doesn't know that such a topic is likely about "pets." It might include a lot of words like "domesticated" and "feral" in that same topic, words that mean the opposite of each other. If they occur together a lot in the same documents, LSA will give them high scores for the same topics together. It's up to us humans to look at what words have a high weight in each topic and give them a name if we want to.

But we don't have to give the topics a name to make use of them. Just as we didn't analyze all the 1000's of dimensions in our stemmed bag-of-words vectors or TF-IDF vectors from previous chapters, we don't have to know what all our topics "mean." We can still do vector math with these new topic vectors, just like we did with TF-IDF vectors. We can add and subtract them and estimate the similarity between documents based on their topic vectors instead of just their word counts.

LSA gives us another bit of useful information. Like the "IDF" part of TF-IDF, it tells us which dimensions in our vector are import to the semantics (meaning) of our documents. We can discard those dimensions (topics) that have the least amount of variance between documents. These low-variance topics are usually distractions, noise, for any machine learning algorithm. If every document has roughly the same amount of some topic and that topic doesn't help us tell the documents apart, then we can get rid of it. And that will help generalize our vector representation so it will work better when we use it with documents our pipeline has not yet seen, even documents from a different context.

This generalization and compression that LSA accomplishes what we attempted in Chapter 2 when we ignored stop words. But the LSA dimension reduction is much better, because it is optimal. It retains as much information as possible, and it doesn't discard any words, it only discards topics, dimensions.

LSA compresses more meaning into fewer dimensions. We only have to retain the high variance dimensions, the major topics that our corpus talks about in a variety of ways (with high variance). And each of these dimensions becomes our "topics" with some weighted combination of all the words captured in each one.

4.2.1 Our Thought Experiment Made Real

Let's use an algorithm to compute some topics like "animalness", "petness", and "cityness" from our thought experiment. We can't tell the LSA algorithm what we want the topics to be about.⁵⁶ But let's just try it and see what happens. For a small corpus of short documents like tweets, chat messages, and lines of poetry, it only takes a few dimensions (topics) to capture the semantics of those documents.

Footnote 56 There is an area of research into something called "learned metrics" which can be used to steer the topics towards what you want them to be about: [Learning Low-Dimensional Metrics](#).

Listing 4.2 Topic-word matrix for LSA on 16 short sentences about cats, dogs and NYC

```
>>> from nlpia.book.examples.ch04_catdog_lsa_3x6x16 import word_topic_vectors
>>> word_topic_vectors.T.round(1)
   cat  dog  apple  lion  nyc  love
top0 -0.6 -0.4    0.5  -0.3  0.4  -0.1
top1 -0.1 -0.3   -0.4  -0.1  0.1   0.8
top2 -0.3  0.8   -0.1  -0.5  0.0   0.1
```

The rows in this topic-word matrix are the "word topic vectors" or just "topic vectors" for each word. This is like the word scores used in the sentiment analysis model in Chapter 2. These will be the vectors we can use to represent the meaning of a word in any machine learning pipeline. So they are also sometimes called word "semantic vectors." And the topic vectors for each word can be added up to compute a topic vector for a document.

Surprisingly SVD created topic vectors analogous to the ones we pulled from our imagination in the thought experiment. The first topic, labeled `topic0`, is a little like our "cityness" topic earlier. The `topic0` weights have larger weights for "apple" and "NYC".

But `topic0` came first in the LSA ordering of topics and last in our imagined topics. LSA sorts the topics in order of importance, how much information or variance they represent for our dataset. The `topic0` dimension is along the axis of highest variance in our dataset. You can see the high variance in the cities when you notice that there are several sentences about "NYC" and "apple"s, and there are several that don't use those words at all.

And `topic1` looks very different from all the thought experiment topics. The LSA algorithm found that "love" was a more important topic than "animalness" for capturing the essence of the documents that we ran it on. The last topic, `topic2`, appears to be about "dog"s, with a little "love" thrown into the mix. The word "cat" is relegated to the "anti-cityness" topic (negative citiness), since cats and cities aren't mentioned together much.

One more short thought experiment should help you appreciate how LSA works, how an algorithm can create topic vectors without knowing what words mean.

MAD LIBS

Can you figure out what the word "awas" means from its context in the statement below?

Awas! Awas! Tom is behind you! Run!

You might not guess that Tom is the alpha Orangutan in Leakey Park, in Borneo. And you might not know that Tom has been "conditioned" to humans but is very territorial, sometimes becoming dangerously aggressive. And your internal natural language processor may not have time to consciously figure out what "awas" means until you have run away to safety.

But once you catch your breath and think about it, you might guess that "awas" means "danger" or "watch out" in Indonesian. Ignoring the real world, and just focusing on the language context, the words, you can often "transfer" a lot of the significance or meaning of words you do know to words that you don't.

Try it sometime, with yourself or with a friend. Like a Mad Libs game,⁵⁷ just replace a word in a sentence with a foreign word, or even a made-up word. Then ask a friend to guess what that word means, or ask them to fill in the blank with an English word. Often your friend's guess won't be too far off from a valid translation of the foreign word, or your intended meaning for the made-up word.

Footnote 57 en.wikipedia.org/wiki/Mad_Libs

Machines, starting with a clean slate, don't have a language to build on. So it takes much more than a single example for them to figure out what the words in it mean. It's like when you look at a sentence full of foreign words. But machines can do it quite well, using LSA, even with just a random sampling of documents containing at least a few mentions of the words you're interested in.

Can you see how shorter documents, like sentences, are better for this than large documents like articles or books? This is because the meaning of a word is usually closely related to the meanings of the words in the sentence that contains it. But this isn't so true about the words that are far apart within a longer document.⁵⁸

Footnote 58 When Tomas Mikolov was thinking about this as came up with `Word2vec` he realized you could tighten up the meaning of word vectors if you tighten up the context even further, limiting the distance between context words to five.

LSA is a way to train a machine to recognize the meaning (semantics) of words and phrases by giving the machine some example usages. Like people, machines can learn better semantics from example usages of words much faster and easier than they can from dictionary definitions. Extracting meaning from example usages requires less logical reasoning than reading all the possible definitions and forms of a word in a dictionary and then encoding that into some logic.

The math we use to uncover the meaning of words in LSA is called Singular Value Decomposition (SVD). Singular Value Decomposition, from your linear algebra class, is what LSA uses to create vectors like those in the word-topic matrices above.⁵⁹

Footnote 59 Check out the examples in `nlpia/book/examples/ch04_*.py` if you want to see the documents and vector math behind this "actualization" of the thought experiment. This really was a thought experiment before SVD was used on real natural language sentences. We were lucky that the topics were at all similar.

Finally some NLP in action, we'll now show you how a machine is able to "play Mad Libs" to understand words.

4.3 Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) is the algorithm behind Latent Semantic Analysis (LSA). Let's start with a corpus of only 11 documents and a vocabulary of 6 words, similar to what we had in mind for our thought experiment.⁶⁰

Footnote 60 We just chose 11 short sentences to keep print version short. You could learn a lot by checking out the ch04 examples in `nlpia` and running SVD on larger and larger corpora

```
>>> from nlpia.book.examples.ch04_catdog_lsa_sorted import lsa_models, prettyify_tdm
>>> bow_svd, tfidf_svd = lsa_models() ①
>>> prettyify_tdm(**bow_svd)
   cat dog apple lion nyc love
0          1      1
1          1      1
2          1      1
3          1      1
                           text
                           NYC is the Big Apple.
                           NYC is known as the Big Apple.
                           I love NYC!
                           I wore a hat to the Big Apple party in NYC.
```

```

4           1       1           Come to NYC. See the Big Apple!
5           1           Manhattan is called the Big Apple.
6   1           1           New York is a big city for a small cat.
7   1           1           The lion, a big cat, is the king of the jungle.
8   1           1           I love my pet cat.
9           1       1           I love New York City (NYC).
10  1   1           Your dog chased my cat.

```

- 1 This performs LSA on the cats_and_dogs corpus using the vocabulary from the thought experiment. We'll soon peak inside this black box.

This is a document-term matrix where each row is a vector of the bag-of-words for a document.

We've limited the vocabulary to match the thought experiment. And we limited the corpus to only a few (eleven) documents that use the six words in our vocabulary. Unfortunately, the sorting algorithm and the limited vocabulary created several identical BOW vectors (NYC, apple). But SVD should be able to "see" that and allocate a topic to that pair of words.

We'll first use SVD on the term-document matrix (the transpose of the document-term matrix above), but it work on TF-IDF matrix or any other vector space model:

```

>>> tdm = bow_svd['tdm']
>>> tdm
      0   1   2   3   4   5   6   7   8   9   10
cat    0   0   0   0   0   0   1   1   1   0   1
dog    0   0   0   0   0   0   0   0   0   0   1
apple   1   1   0   1   1   1   0   0   0   0   0
lion    0   0   0   0   0   0   0   1   0   0   0
nyc     1   1   1   1   1   0   0   0   0   1   0
love    0   0   1   0   0   0   0   0   1   1   0

```

SVD is an algorithm for decomposing any matrix into three "factors", three matrices that can be multiplied together to recreate the original matrix. This is analogous to finding exactly 3 integer factors for a large integer. But our factors aren't scalar integers, they are 2D real matrices with special properties. The three matrix factors we compute with SVD have some useful mathematical properties we can exploit for dimension reduction and LSA. In linear algebra class you may have used SVD to find the inverse of a matrix. Here we'll use it for Latent Semantic Analysis to figure out what our topics (groups of related words) needs to be.

Whether we run SVD on a BOW term-document matrix or a TF-IDF term-document matrix, SVD will find combinations of words that belong together. SVD finds those co-occurring words by calculating the correlation between the columns (terms) of our term-document matrix.⁶¹ SVD simultaneously finds the correlation of term use between

documents and the correlation of documents with each other. With these two pieces of information SVD also computes the linear combinations of terms that have the greatest variation across the corpus. These linear combinations of term frequencies will become our topics. And we'll keep only those topics that retain the most information, the most variance in our corpus. And it gives us the linear transformation, rotation, of our term-document vectors to convert those vectors into shorter topic vectors for each document.

Footnote 61 This is equivalent to the square root of the dot product of two columns (term-document occurrence vectors), but SVD provides us additional information that computing the correlation directly wouldn't provide.

SVD will group terms together that have high correlation with each other (because they occur in the same documents together a lot) and also vary together a lot over the set of documents. We think of these linear combinations of words as "topics". These topics turn our BOW vectors (or TF-IDF vectors) into topic vectors that tell us the topics a document is about. A topic vector is kind-of like a summary, or generalization, of what the document is about.

It's unclear who came up with the idea to apply SVD to word counts to create topic vectors. Several linguists were working on similar approaches simultaneously. They were all finding that the semantic similarity between two natural language expressions (or individual words) is proportional to the similarity between the contexts in which words or expressions are used. These researchers included Harris, Z. S. (1951)⁶², Koll (1979)⁶³, Isbell (1998)⁶⁴, Dumais et al (1988)⁶⁵, Salton and Lesk (1965)⁶⁶, Deerwester (1990)⁶⁷.

Footnote 62 Jurafsky and Schone cite "Methods in structural linguistics" by Harris, Z. S., 1951 in their 2000 paper [Knowledge-Free Induction of Morphology Using Latent Semantic Analysis](#)

Footnote 63 Koll, M. (1979) "[Generalized vector spaces model in information retrieval](#)" An Approach to Concept Based Information Retrieval" by Koll, M. (1979)

Footnote 64 "[Restructuring Sparse High Dimensional Data for Effective Retrieval](#)" by Charles Lee Isbell Jr et all 1998

Footnote 65 "[Using latent semantic analysis to improve access to textual information](#)" by Dumais, Furnas, Landauer, Deerwester, and Harshman in 1988

Footnote 66 Salton, G., (1965) "The SMART automatic document retrieval system"

Footnote 67 Deerwester, S. et al. "Indexing by Latent Semantic Indexing."

Here's what SVD (the heart of LSA) looks like in math notation.

$W_{m \times n}$ $U_{m \times p}$ $S_{p \times p}$ $V_{p \times nT}$

In this formula, m is the number of terms in our vocabulary, n is the number of documents in our corpus, and p is the number of topics in our corpus, and this is the same as the number of words. But wait, weren't we trying to end up with fewer dimensions? We want to eventually end up with fewer topics than words, so we can use those topic vectors (rows of the topic-document matrix) as a reduced-dimension representation of the original TF-IDF vectors. We'll eventually get to that. But at this first stage, we retain all of the dimensions in our matrices.

Here's what those three matrices (U , S , and V) look like.

4.3.1 U —Left Singular Vectors

The U matrix contains the term-topic matrix that tells us about "the company a word keeps".⁶⁸ This is the most important matrix for semantic analysis in NLP. The U matrix is called the "left singular vectors" because it contains row vectors that should be multiplied by a matrix of column vectors from the left.⁶⁹ U is the cross-correlation between words and topics based on word co-occurrence in the same document. It's a square matrix until we start truncating it (deleting columns). It has the same number of rows and columns as we have words in our vocabulary (m), six. We still have six topics (p) because we haven't truncated this matrix... yet.

Footnote 68 If you try to duplicate these results with the PCA model in `sklearn` you'll notice that it gets this term-topic matrix from the VT matrix because the input data set is transposed relative to what we did here. `scikit-learn` always arranges data as row vectors so our term-document matrix in `tdm` is transposed into a document-term matrix when you use `PCA.fit()` or any other `sklearn` model training.

Footnote 69 Mathematicians call these vectors "left eigenvectors" or "row eigenvectors.":
en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors#Left_and_right_eigenvectors

Listing 4.3 $Um \times p$

```
>>> import numpy as np
>>> U, s, vt = np.linalg.svd(tdm) ①
>>> import pandas as pd
>>> pd.DataFrame(U, index=tdm.index).round(2)
      0    1    2    3    4    5
cat -0.04  0.83 -0.38 -0.00  0.11 -0.38
dog -0.00  0.21 -0.18 -0.71 -0.39  0.52
apple -0.62 -0.21 -0.51  0.00  0.49  0.27
lion -0.00  0.21 -0.18  0.71 -0.39  0.52
nyc  -0.75 -0.00  0.24 -0.00 -0.52 -0.32
love -0.22  0.42  0.69  0.00  0.41  0.37
```

- ➊ We're reusing the tdm term-document matrix from the code sections above.

Notice that the SVD algorithm is a bread-and-butter `numpy` math operation, not a fancy `scikit-learn` machine learning algorithm.

The U matrix contains all the topic vectors for each word in our corpus as columns. This means it can be used as a transformation to convert a word-document vector (a TF-IDF vector or a BOW vector) into a topic-document vector. We just multiply our topic-word U matrix by any word-document column vector to get a new topic-document vector. This is because the weights or scores in each cell of the U matrix represent how important each word is to each topic. This exactly what we did in the thought experiment that started this whole cats and dogs adventure in NYC.

Even though we have what we need to map word frequencies to topics, we'll explain the remaining factors that SVD gives us and how they are used.

4.3.2 S—*Singular Values*

The Sigma or S matrix contains the topic "singular values" in a square diagonal matrix.⁷⁰ The singular values tell us how much information is captured by each dimension in our new semantic (topic) vector space. A diagonal matrix has nonzero values only along the diagonal from the upper left to the lower right. Everywhere else the S matrix will have zeros. So `numpy` saves space by returning the singular values as an array, but we can easily convert it to a diagonal matrix with the `numpy.diag` function:

Footnote 70 Mathematicians call these eigenvalues.

Listing 4.4 Sp x p

```
>>> s.round(1)
array([3.1, 2.2, 1.8, 1. , 0.8, 0.5])
>>> S = np.zeros((len(U), len(Vt)))
>>> pd.np.fill_diagonal(S, s)
>>> pd.DataFrame(S).round(1)
   0   1   2   3   4   5   6   7   8   9   10
0  3.1  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
1  0.0  2.2  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
2  0.0  0.0  1.8  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
4  0.0  0.0  0.0  0.0  0.8  0.0  0.0  0.0  0.0  0.0  0.0
5  0.0  0.0  0.0  0.0  0.0  0.5  0.0  0.0  0.0  0.0  0.0
```

Like the U matrix, our S matrix for our 6-word, 6-topic corpus has six rows (p). However it has many more columns (n) filled with zeros. It needs a column for every document so we can multiply it by VT , the document-document matrix, that you'll learn about next.

Since we have not yet reduced the dimensionality by truncating this diagonal matrix, we have as many topics (p) as we have terms in our vocabulary (m), six. And our dimensions (topics) are constructed such that the first dimension contains the most information ("explained variance") about our corpus. That way when we want to truncate our topic model, we can start zeroing out the dimensions at the lower right and work our way up and to the left. We can stop zeroing out these singular values when the error in our topic model starts to contribute significantly to the overall NLP pipeline error.

TIP

Here's the trick we mentioned earlier. For NLP, and most other applications, you don't want to retain the variance information in your topic model. The documents you process in the future might not be about the same topics. In most cases you're better off setting the diagonal elements of your S matrix to ones, creating a rectangular identity matrix that just reshapes the VT document-document matrix to be compatible with your U word-topic matrix. That way if you multiply this S matrix by some new set of document vectors you won't skew the topic vectors towards your original topic mix (distribution).

4.3.3 VT—Right Singular Vectors

The VT matrix contains the "right singular vectors" as the columns of the document-document matrix. This gives you the shared meaning between documents, because it measures how often documents use the same topics in our new semantic model of the documents. It has the same number of rows (p) and columns as we have documents in our small corpus, 11.

Listing 4.5 $Vp \times nT$

```
>>> pd.DataFrame(Vt).round(2)
      0    1    2    3    4    5    6    7    8    9    10
0   -0.44 -0.44 -0.31 -0.44 -0.44 -0.20 -0.01 -0.01 -0.08 -0.31 -0.01
1   -0.09 -0.09  0.19 -0.09 -0.09 -0.09  0.37  0.47  0.56  0.19  0.47
2   -0.16 -0.16  0.52 -0.16 -0.16 -0.29 -0.22 -0.32  0.17  0.52 -0.32
3    0.00 -0.00 -0.00  0.00  0.00  0.00 -0.00  0.71  0.00 -0.00 -0.71
4   -0.04 -0.04 -0.14 -0.04 -0.04  0.58  0.13 -0.33  0.62 -0.14 -0.33
5   -0.09 -0.09  0.10 -0.09 -0.09  0.51 -0.73  0.27 -0.01  0.10  0.27
6   -0.57  0.21  0.11  0.33 -0.31  0.34  0.34 -0.00 -0.34  0.23  0.00
7   -0.32  0.47  0.25 -0.63  0.41  0.07  0.07  0.00 -0.07 -0.18  0.00
8   -0.50  0.29 -0.20  0.41  0.16 -0.37 -0.37 -0.00  0.37 -0.17  0.00
9   -0.15 -0.15 -0.59 -0.15  0.42  0.04  0.04 -0.00 -0.04  0.63 -0.00
10  -0.26 -0.62  0.33  0.24  0.54  0.09  0.09 -0.00 -0.09 -0.23 -0.00
```

Like the S matrix, we'll ignore the VT matrix whenever we are transforming new word-document vectors into our topic vector space. We'll only use it to check the accuracy of our topic vectors for recreating the original word-document vectors that we used to "train" it.

4.3.4 SVD Matrix Orientation

If you've done machine learning with natural language documents before you may notice that our term-document matrix is "flipped" (transposed) relative to what you're used to seeing in `scikit-learn` and other packages. In the Naive Bayes sentiment model at the end of Chapter 2, and the TF-IDF vectors of Chapter 3, we created our training set as a document-term matrix. This is the orientation that `scikit-learn` models require. Each row of our training set in the sample-feature matrix for a machine learning sample, a document. And each column represented a word or feature of those documents. But when we do the SVD linear algebra directly, our matrix needs to be transposed into term-document format.⁷¹

Footnote 71 Actually, within the `sklearn.PCA` model they leave the document-term matrix unflipped and just flip the SVD matrix math operations. So the PCA model in `scikit-learn` ignores the U and S matrix and uses only the VT matrix for its transformation of new document-term row vectors into document-topic row vectors.

DEFINITION

Matrices are named and sized by their rows first, then the columns. So a "term-document" matrix is a matrix where the rows are the words, and the columns are the documents. Matrix dimensions (sizes) work the same way. A 2×3 matrix will have 2 rows and 3 columns, which means it has an `np.shape()` of `(2, 3)` and a `len()` of 2.

Don't forget to transpose your term-document or topic-document matrices back to the `scikit-learn` orientation before training a machine learning model. In `scikit-learn` each row in an NLP training set should contain a vector of the features associated with a document (an email, SMS message, sentence, web page, or any other chunk of text). In NLP training sets our vectors are row vectors. In traditional linear algebra operations, vectors are usually thought of as column vectors.

In the next section we'll go through all this with you to train a `scikit-learn` `TruncatedSVD` transformer to transform bag-of-word vectors into topic-document vectors. We'll then transpose those vectors back to create the rows of our training set so we can train a `scikit-learn` (`sklearn`) classifier on those document-topic vectors.

WARNING

If you're using `scikit-learn`, you must transpose the feature-document matrix (usually called `x` in `sklearn`) to create a document-feature matrix to pass into your `.fit()` and `.predict()` methods of a model. Each row in a training set matrix should be a feature vector for a particular sample text, usually a document.⁷²

Footnote 72 `scikit-learn` documentation on LSA:

scikit-learn.org/stable/modules/decomposition.html#lsa

4.3.5 Truncating the Topics

We now have a topic model, a way to transform word frequency vectors into topic weight vectors. But since we have just as many topics as words, our vector space model has just as many dimensions as the original BOW vectors. We've just created some new words and called them "topics" because they each combine words together in various ratios. We haven't reduced the number of dimensions... yet.

We can ignore the S matrix, since the rows and columns of our U matrix are already arranged so that the most important topics (with the largest singular values) are on the left. Another reason we can ignore S is that most of the word-document vectors we'll want to use with this model, like TF-IDF vectors, have already been normalized. Finally, it just produces better topic models if we do it this way.⁷³

Footnote 73 Levy, Goldberg and Dagan, Improving Distributional Similarity with Lesson Learn from Word Embeddings, 2015

So let's start lopping off columns on the right hand side of U . But wait. How many topics will be enough to capture the essence of a document? One way to measure the accuracy of LSA is to see how accurately you can recreate a term-document matrix from a topic-document matrix. Here's a plot of the reconstruction accuracy for the 9-term, 11 document matrix we used earlier to demonstrate SVD.

Listing 4.6 Term-document matrix reconstruction error

```
>>> err = []
>>> for numdim in range(len(s), 0, -1):
...     S[numdim - 1, numdim - 1] = 0
...     reconstructed_tdm = U.dot(S).dot(Vt)
...     err.append(np.sqrt(((reconstructed_tdm - tdm).values.flatten() ** 2).sum()
...                       / np.product(tdm.shape)))
>>> np.array(err).round(2)
array([0.06, 0.12, 0.17, 0.28, 0.39, 0.55])
```

When we reconstruct a term-document matrix for our 11 documents using the the singular vectors, the more we truncate, the more the error grows. Our 3-topic model from earlier would have about 28% error if we used it to reconstruct BOW vectors for each document. Here's a plot of that accuracy drop as we drop more and more dimensions in our topic model.

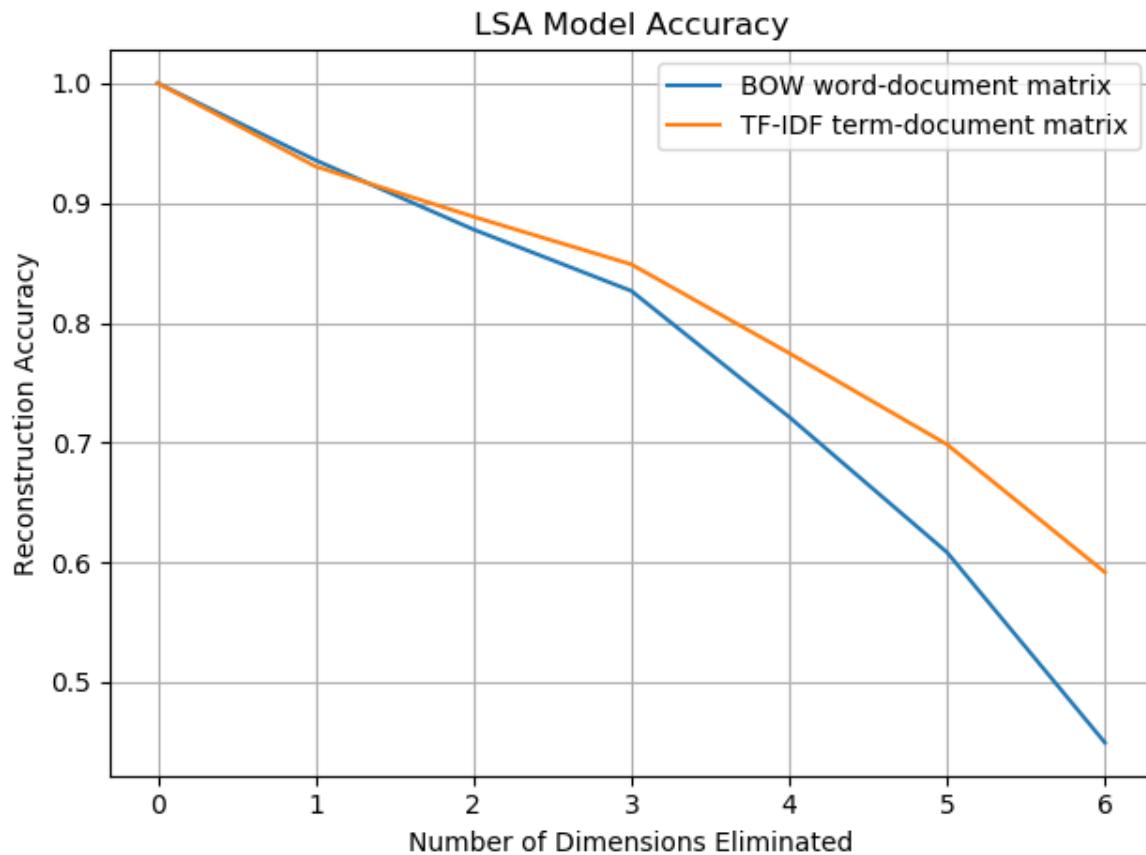


Figure 4.3 Term-document matrix reconstruction accuracy decreases as we ignore more dimensions

As you can see, the accuracy drop is pretty similar, whether you use TF-IDF vectors or BOW vectors for your model. But TF-IDF vectors will perform slightly better if you plan to retain only a few topics in your model.

This is a very simple example, but you can see how you might use a plot like this to decide how many topics (dimensions) you want in your model. In some cases you may find that you get perfect accuracy, after eliminating several of the dimensions in your term-document matrix. Can you guess why?

The SVD algorithm behind LSA "notices" if words are always used together and puts them together in a topic. That's how it can get a few dimensions "for free." So, even if

you don't plan to use a topic model in your pipeline, LSA (SVD) can be a great way to compress your word-document matrices and identify potential compound words or n -grams for you pipeline.

4.4 Principal Component Analysis (PCA)

Principle Component Analysis is another name for SVD when it's used for dimension reduction, like we did to accomplish our Latent Semantic Analysis above. And the `PCA` model in `scikit-learn` has some tweaks to the SVD math that will improve the accuracy of our NLP pipeline.

For one, `sklearn.PCA` automatically "centers" your data by subtracting off the mean word frequencies. Another, more subtle trick, is that PCA uses a function called `flip_sign` to deterministically compute the sign of the singular vectors.⁷⁴

Footnote 74 You can find some experiments with these functions within PCA that we used to understand all these subtleties in `nlpia.book.examples.ch04_sklearn_pca_source`

Finally, the `sklearn` implementation of `PCA` implements an optional "whitening" step. This is similar to our trick of ignoring the singular values when transforming word-document vectors into topic-document vectors. Instead of just setting all the singular values in S to one, whitening divides your data by these variances just like the `sklearn.StandardScaler` transform does. This helps spread out your data and makes any optimization algorithm less likely to get lost in "half pipes" or "rivers" of your data that can arrise when features in your dataset are correlated with each other.⁷⁵

Footnote 75 mccormickml.com/2014/06/03/deep-learning-tutorial-pca-and-whitening/

Before we apply PCA to real-world high-dimensional NLP data, lets take a step back and look at a more visual representation of what PCA and SVD do. This will also help you understand the API for the `scikit-learn` PCA implementation. PCA is useful for a wide range of applications, so this insight will be useful for more than just NLP. We're going to do PCA on a 3D point cloud before we try it out on high-dimensional natural language data.

For most "real" problems you'll want to use the `sklearn.PCA` model for your Latent Semantic Analysis problems. The one exception is if you have more documents than you can hold in RAM. In that case you'll need to use the `IncrementalPCA` model in `sklearn` or some of the scaling techniques we talk about in Chapter 13.

TIP

If you have a huge corpus and you urgently need topic vectors (LSA), skip to Chapter 13 and check out `gensim.models.LsiModel`. If a single machine still isn't enough to get the work done quickly, check out [RocketML's](#) parallelization of the SVD algorithm.

We're going to start with a set of real-world 3D vectors, rather than 10,000+ dimensional document-word vectors. It's a lot easier to visualize things in 3D than it is in 10000D. Since we're only dealing with 3 dimensions, it's straightforward to plot them using the `Axes3D` class in `matplotlib`. See the `nlpia` (github.com/totalgood/nlpia) package for the code to create rotatable 3D plots like this.

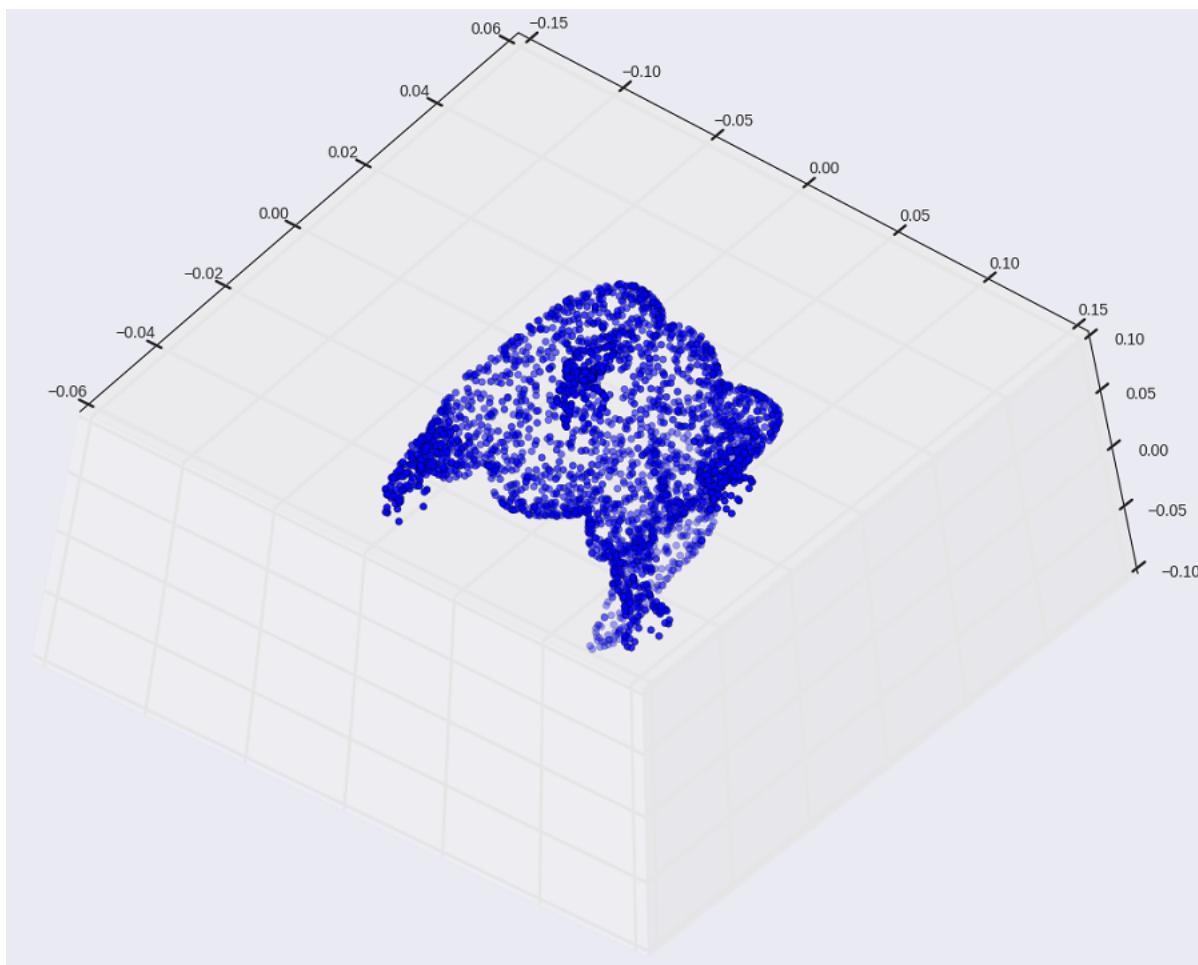


Figure 4.4 Looking up from below the "belly" at the point cloud for a real object

In fact, the point cloud shown above is from the 3D scan of the surface of a real-world object, not the pointy tips of a set of BOW vectors. But this will help you get a feel for how LSA works. And you can see how to manipulate and plot small vectors before we tackle higher-dimensional vectors like document-word vectors.

Can you guess what this 3D object is that created these 3D vectors? You only have a 2D projection printed in this book to go on. Can you think of how you would program a machine to rotate the object around so that you could get a better view? Are there statistics about the data points that you could use to align the X and Y axes with the object optimally? As you rotate the 3D blob in your mind, imagine how the variance along the X, Y, and Z axes might change as you rotate it.

4.4.1 PCA on 3D Vectors

We manually rotated the point cloud into this particular orientation to *minimize* the variance along the axes of the window for the plot. We did that so that you'd have a hard time recognizing what it is. If SVD (LSA) did this to our document-word vectors it would "hide" the information in those vectors. Stacking the points on top of each other in our 2D projection prevents human eyes, or machine learning algorithms, from separating the points into meaningful clusters. But SVD preserves the structure, information content, of our vectors by *maximizing* the variance along the dimensions of our lower dimensional "shadow" of the high-dimensional space. This is what we need for machine learning so that each low dimensional vector captures the "essence" of whatever it represents. SVD *maximizes* the variance along each axis. And variance turns out to be a pretty good indicator of "information" or that "essence" we're looking for.

```
>>> import pandas as pd
>>> pd.set_option('display.max_columns', 6)
1
>>> from sklearn.decomposition import PCA
2
>>> import seaborn
>>> from matplotlib import pyplot as plt
>>> from nlpia.data.loaders import get_data

>>> df = get_data('pointcloud').sample(1000)
3
>>> pca = PCA(n_components=2)
>>> df2d = pd.DataFrame(pca.fit_transform(df), columns=list('xy'))
>>> df2d.plot(kind='scatter', x='x', y='y')
>>> plt.show()
```

- ➊ Ensure that our pd.DataFrame printouts fit within the width of a page
- ➋ Even though it's called PCA in scikit-learn, this is really just SVD
- ➌ We're reducing a 3D point cloud to a 2D "projection" for display in a 2D scatter plot

If you run the script above, the orientation of your 2D projection may randomly "flip" left to right, but it never tips or twists to a new angle. The orientation of the 2D projection is computed so that the maximum variance is always aligned with the x axis, the first axis.

The 2nd largest variance is always aligned with the y axis, the second dimension of our "shadow" or "projection". But the *polarity* (sign) of these axes is arbitrary because the optimization has two remaining degrees of freedom. The optimization is free to flip the polarity of the vectors (points) along the x or y axis, or both.

There's also a `horse_plot.py` script in the `nlpia/data` directory if you'd like to play around with the 3D orientation of the horse. There may indeed be a more optimal transformation of the data that eliminates one dimension without reducing the information content of that data (to your eye). And Picasso's cubist "eye" might come up with a nonlinear transformation that maintains the information content of views from multiple perspectives all at once. And there are "embedding" algorithms to do this like one we'll talk about in Chapter 6.

But don't you think good old linear SVD and PCA does a pretty good job of preserving the "information" in the point cloud vector data? Doesn't our 2D projection of the 3D horse provide a good view of the data? Wouldn't a machine be able to learn something from the statistics of these 2D vectors computed from the 3D vectors of the surface of a horse?

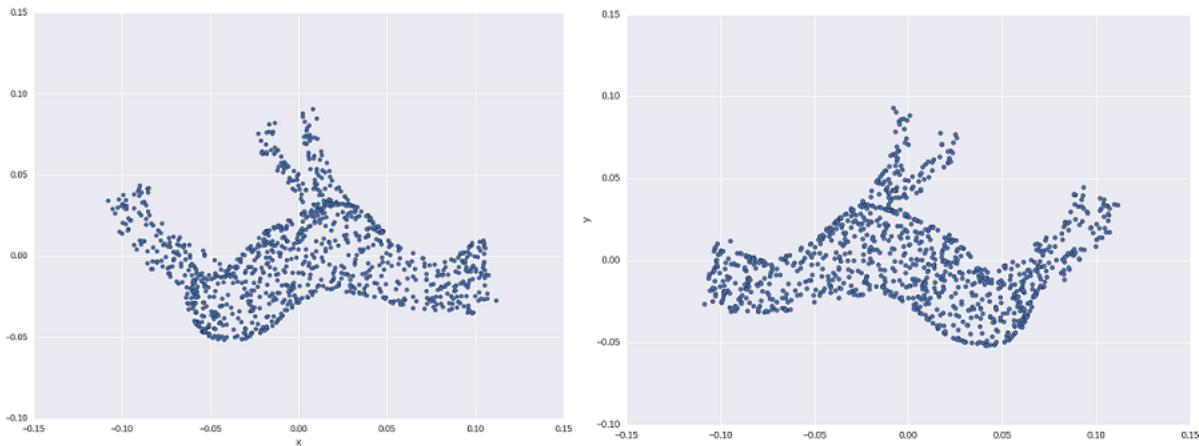


Figure 4.5 Head to Head Horse Point Clouds Upside Down

4.4.2 Stop Horsing Around and Get Back to NLP

Let's see how SVD will do on some natural language documents. Let's find the principal components using SVD the 5000 SMS messages labeled as spam (or not). The vocabulary and variety of topics discussed in this limited set of SMS messages from a university lab should be relatively small. So let's limit the number of topics to 16. We'll use both the `scikit-learn` PCA model as well as the TruncatedSVD model to see if there are any differences.

The TruncatedSVD model is designed to work with sparse matrices. Sparse matrices are matrices that have the same value (usually zero or NaN) in a lot of the cells. NLP bag-of-word and TF-IDF matrices are almost always sparse, because most documents don't contain many of the words in our vocabulary. Most of our word counts are zero (before we add a "ghost" count to them all to smooth our data out).

Sparse matrices are like spreadsheets that are mostly empty, but have a few meaningful values scattered around. The `sklearn PCA` model may provide a faster solution than `TruncatedSVD` by using dense matrices with all those zeros filled in. But `sklearn.PCA` wastes a lot of RAM trying to "remember" all those zeros that are duplicated all over the place. The `TfidfVectorizer` in `scikit-learn` outputs sparse matrices so we just need to convert those to dense matrices before we compare the results to PCA.

First, let's load the SMS messages from a `DataFrame` in the `nlpia` package.

```
>>> import pandas as pd
>>> from nlpia.data.loaders import get_data
>>> pd.options.display.width = 120
1
>>> sms = get_data('sms-spam')
2
>>> index = ['sms{{}}'.format(i, '*'*j) for (i,j) in zip(range(len(sms)), sms.spam)]
>>> sms.index = index
>>> sms.head(6)
   spam                                text
sms0    0  Go until jurong point, crazy.. Available only ...
sms1    0                      Ok lar... Joking wif u oni...
sms2!   1  Free entry in 2 a wkly comp to win FA Cup fina...
sms3    0  U dun say so early hor... U c already then say...
sms4    0  Nah I don't think he goes to usf, he lives aro...
sms5!   1  FreeMsg Hey there darling it's been 3 week's n...
```

Now we can calculate the TF-IDF vectors for each of these messages.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from nltk.tokenize.casual import casual_tokenize
1
>>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray()
>>> len(tfidf.vocabulary_)
9232
2
>>> tfidf_docs = pd.DataFrame(tfidf_docs)
3
>>> tfidf_docs = tfidf_docs - tfidf_docs.mean()
>>> tfidf_docs.shape
(4837, 9232)
>>> sms.spam.sum()
638
```

- ① This centers our vectorized documents (BOW vectors) by subtracting the mean

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

- ② The `.shape` attribute tells us the length or cardinality of each of the dimensions for any numpy array
- ③ The `.sum()` method on a Pandas Series acts just like an Spreadsheet column sum, adding up all the elements of each Series

So we have 4,837 SMS messages with 9,232 different 1-gram tokens from our ``casual_tokenize`r`. Only 638 of these 4,837 messages (13%) are labeled as spam. So we have an unbalanced training set with about 8:1 "ham" (normal SMS messages) to "spam" (unwanted solicitations and advertisements).

We might deal with this "ham" sampling bias by reducing the "reward" for any model that classifies "ham" correctly. But the large vocabulary size, $|V|$, is trickier to deal with. The 9,232 tokens in our vocabulary is greater than the 4,837 messages (samples) we have to go on. So we have many more unique words in our vocabulary (or lexicon) than we have SMS messages. And of those SMS messages only a small portion of them (1/8th) are labeled as spam. That's a recipe for overfitting.⁷⁶ There will be only a few unique words out of our large vocabulary that are labeled as "spammy" words in our dataset.

Footnote 76 en.wikipedia.org/wiki/Overfitting

Overfitting means that we will "key" off of only very few words in our vocabulary. So our spam filter will be dependent on those spammy words being somewhere in the spammy messages it filters out. A spammer could easily get around our filter if they just used synonyms for those spammy words. If our vocabulary doesn't include the spammer's new synonyms, then our filter will mis-classify those cleverly constructed SMS messages as "ham."

And this overfitting problem is an inherent problem in NLP. It's really hard to find a labeled natural language dataset that includes all the ways that people might say something that should be labeled that way. I couldn't find a big database of SMS messages that included all the different ways people say spammy and nonspammy things. And only a few corporations have the resources to create such a dataset. So all the rest of us need to have "countermeasures" for overfitting. We have to use algorithms that "generalize" well on just a few examples.

Dimension reduction is the primary countermeasure for overfitting. By consolidating our dimensions (words) into a smaller number of dimensions (topics), our NLP pipeline will become more "general". Our spam filter will work on a wider range of SMS messages if we reduce our dimensions, our "vocabulary."

That's exactly what Latent Semantic Analysis (LSA) does. LSA will reduce our dimensions and thus reduce overfitting. It will generalize from our small dataset by assuming a linear relationship between word counts. So if the word "half" occurs in spammy messages containing words like "off" a lot (as in "Half off!" LSA will help us make those connections between words and see how strong they are so it will generalize from the phrase "half off" in a spammy message to phrases like "80 % off." And it might even generalize further to the phrase "80 % discount" if the chain of connections in our NLP data includes "discount" associated with the word "off."

TIP

Some think of generalization as the core challenge of machine learning and artificial intelligence. "One-shot Learning" is often used to describe research into models that take this to the extreme, requiring orders of magnitude less data to accomplish the same accuracy as conventional models.

Generalizing our NLP pipeline will help ensure it applies to a broader set of real-world SMS messages instead of just this particular set of messages.

4.4.3 Using PCA for SMS Message Semantic Analysis

Let's try the PCA model from `scikit-learn` first. You've already seen it in action wrangling 3D horses into a 2D pen, now let's wrangle our dataset of 9232-D TF-IDF vectors into 16-D topic vectors.

```
>>> from sklearn.decomposition import PCA

>>> pca = PCA(n_components=16)
>>> pca = pca.fit(tfidf_docs)
>>> pca_topic_vectors = pca.transform(tfidf_docs)
>>> columns = ['topic{}'.format(i) for i in range(pca.n_components)]
>>> pca_topic_vectors = pd.DataFrame(pca_topic_vectors, columns=columns, index=index)
>>> pca_topic_vectors.round(3).head(6)
   topic0  topic1  topic2  ...  topic13  topic14  topic15
sms0    0.201   0.003   0.037  ...    -0.026   -0.019    0.039
sms1    0.404  -0.094  -0.078  ...    -0.036    0.047   -0.036
sms2!   -0.030  -0.048   0.090  ...    -0.017   -0.045    0.057
sms3     0.329  -0.033  -0.035  ...    -0.065    0.022   -0.076
sms4     0.002   0.031   0.038  ...     0.031   -0.081   -0.021
sms5!   -0.016   0.059   0.014  ...     0.077   -0.015    0.021
```

If you're curious about these topics you can find out how much of each word they "contain" by examining their weights. By looking at the weights we can see how often "half" occurs with the word "off" (as in "half off") and then figure out which topic is our "discount" topic.

TIP

You can find the "weights" of any fitted `sklearn` model or transformation by examining its `components` attribute.

First lets assign words to all the dimensions in our PCA transformation. We need to get them in the right order since our TFIDFVectorizer stores the vocabulary as a dictionary that maps each term to an index number (column number).

```
>>> tfidf.vocabulary_
{'go': 3807,
 'until': 8487,
 'jurong': 4675,
 'point': 6296,
 ...
>>> column_nums, terms = zip(*sorted(zip(tfidf.vocabulary_.values(), tfidf.vocabulary_.keys())))
>>> terms
('!', '',
 '',
 '#',
 '#150',
 ...)
```

No we can create a nice pandas DataFrame containing the weights, with labels for all the columns and rows in the right place.

```
>>> weights = pd.DataFrame(pca.components_, columns=terms, index=['topic{}'.format(i) for i in range(16)])
>>> pd.options.display.max_columns = 8
>>> weights.head(4).round(3)
          !      "      #    #150    ...        ..      ud
topic0 -0.071  0.008 -0.001 -0.000  ...   -0.002  0.001  0.001  0.001
topic1  0.063  0.008  0.000 -0.000  ...    0.003  0.001  0.001  0.001
topic2  0.071  0.027  0.000  0.001  ...    0.002 -0.001 -0.001 -0.001
topic3 -0.059 -0.032 -0.001 -0.000  ...    0.001  0.001  0.001  0.001
```

Some of those columns (terms) aren't very interesting, so let's explore around in our `tfidf.vocabulary_`. Let's see if we can find some of those "half off" terms and which topics they're a part of.

```
>>> pd.options.display.max_columns = 12
>>> deals = weights['! ;) :) half off free crazy deal only $ 80 %'].split().round(3) * 100
>>> deals
          !      ;)      :)      half      off      free      crazy      deal      only      $      80      %
topic0   -7.1     0.1    -0.5    -0.0    -0.4    -2.0    -0.0    -0.1    -2.2     0.3    -0.0    -0.0
topic1    6.3     0.0     7.4     0.1     0.4    -2.3    -0.2    -0.1    -3.8    -0.1    -0.0    -0.2
topic2    7.1     0.2    -0.1     0.1     0.3     4.4     0.1    -0.1     0.7     0.0     0.0     0.1
topic3   -5.9    -0.3    -7.1     0.2     0.3    -0.2     0.0     0.1    -2.3     0.1    -0.1    -0.3
topic4   38.1    -0.1   -12.5    -0.1    -0.2     9.9     0.1    -0.2     3.0     0.3     0.1    -0.1
topic5  -26.5     0.1   -1.5    -0.3    -0.7    -1.4    -0.6    -0.2    -1.8    -0.9     0.0     0.0
topic6  -10.9    -0.5   19.9    -0.4    -0.9    -0.6    -0.2    -0.1    -1.4    -0.0    -0.0    -0.1
topic7   16.4     0.1  -18.2     0.8     0.8    -2.9     0.0     0.0    -1.9    -0.3     0.0    -0.1
topic8   34.6     0.1     5.2    -0.5    -0.5    -0.1    -0.4    -0.4     3.3    -0.6    -0.0    -0.2
topic9    6.9    -0.3   17.4     1.4    -0.9     6.6    -0.5    -0.4     3.3    -0.4    -0.0     0.0
... 
```

```
>>> deals.T.sum()
topic0      -11.9
topic1       7.5
topic2      12.8
topic3     -15.5
topic4      38.3
topic5     -33.8
topic6       4.8
topic7     -5.3
topic8      40.5
topic9      33.1
...
...
```

Topics 4, 8, and 9, appear to all contain positive "deal" topic sentiment. And topics 0, 3, and 5 appear to be "anti-deal" topics, messages about stuff that's the opposite of "deals," negative deals. So words associated with "deals" can have a positive impact on some topics and a negative impact on others. There's no single obvious "deal" topic number.

DEFINITION

The `casual_tokenize` tokenizer splits "`80%`" into `["80", "%"]` and "`$80 million`" into `[$, 80, "million"]`. So unless we use LSA or a 2-gram tokenizer, our NLP pipeline would not notice the difference between `80%` and `$80 million`. They'd both share the token "`80`".

This is one of the challenges of LSA, making sense of the topics. LSA only allows for linear relationships between words. And we usually only have a small corpus to work with. So our topics tend to combine words in ways that humans don't find all that meaningful. Several words from different topics will be crammed together into a single dimension (principle component) in order to make sure the model captures as much variance in usage of our 9,232 words as possible.

4.4.4 Using Truncated SVD for SMS Message Semantic Analysis

Now we can try the `TruncatedSVD` model in `scikit-learn`. This is a more direct approach to LSA that bypasses the `scikit-learn` PCA model so you can see what's going on inside the PCA wrapper. It can handle sparse matrices, so if you're working with large datasets you'll want to use `TruncatedSVD` instead of `PCA` anyway. The SVD part of `TruncatedSVD` will split our TF-IDF matrix into three matrices. The Truncated part of SVD will discard the dimensions that contain the least information about our TF-IDF matrix. These discarded dimensions represent the "topics" (linear combinations of words) that vary the least within our document set. These discarded topics would likely be meaningless to the overall semantics of our corpus. They'd likely contain a lot of stop words and other words that are uniformly distributed across all the documents.

We're going to use `TruncatedSVD` to retain only the 16 most interesting topics, the topics

that account for the most variance in our TF-IDF vectors:

```
>>> from sklearn.decomposition import TruncatedSVD
>>> svd = TruncatedSVD(n_components=16, n_iter=100) ❶
>>> svd_topic_vectors = svd.fit_transform(tfidf_docs.values) ❷
>>> svd_topic_vectors = pd.DataFrame(svd_topic_vectors, columns=columns, index=index)
>>> svd_topic_vectors.round(3).head(6)
   topic0  topic1  topic2    ...  topic13  topic14  topic15
sms0    0.201   0.003   0.037    ...   -0.036   -0.014    0.037
sms1    0.404  -0.094  -0.078    ...   -0.021    0.051   -0.042
sms2!   -0.030  -0.048   0.090    ...   -0.020   -0.042    0.052
sms3    0.329  -0.033  -0.035    ...   -0.046    0.022   -0.070
sms4    0.002   0.031   0.038    ...    0.034   -0.083   -0.021
sms5!   -0.016   0.059   0.014    ...    0.075   -0.001    0.020
```

- ❶ Just like PCA, we'll compute 16 topics but will iterate through the data 100 times (default is 5) to ensure that our answer is almost as exact as PCA
- ❷ `fit_transform` decomposes our TF-IDF vectors and transforms them into topic vectors in one step

These topic vectors from `TruncatedSVD` are exactly the same as what `PCA` produced! This is because we were careful to use a large number of iterations (`n_iter`) and we also made sure all our TF-IDF frequencies for each term (column) were centered on zero (by subtracting the mean for each term).

Look at the weights for each topic for a moment and try to make sense of them. Without knowing what these topics are about, or the words they weight heavily, do you think you could classify these 6 SMS messages as spam or not? Perhaps looking at the "!" label next to the spammy sms message row labels will help. It would be hard, but it is possible, especially for a machine that can look at all 5000 of our training examples and come up with thresholds on each topic to separate the topic space for spam and non-spam.

4.4.5 How well does LSA work for spam Classification?

One way to find out how well a vector space model will work for classification is to see how well cosine similarities between vectors correlate with membership in the same class. Let's see if the cosine similarity between corresponding pairs of documents is useful for our particular binary classification. Let's compute the dot product between the first six topic vectors for the first six SMS messages. We should see larger positive cosine similarity (dot products) between any spam message ("sms2!") and other and non-spam messages have a low cosine similarity.

```
>>> import numpy as np
```

```
>>> svd_topic_vectors = (svd_topic_vectors.T /
    np.linalg.norm(svd_topic_vectors, axis=1)).T
>>> svd_topic_vectors.iloc[:10].dot(svd_topic_vectors.iloc[:10].T).round(1)
   sms0  sms1  sms2!  sms3  sms4  sms5!  sms6  sms7  sms8!  sms9!
sms0    1.0    0.6   -0.1    0.6   -0.0   -0.3   -0.3   -0.1   -0.3   -0.3
sms1    0.6    1.0   -0.2    0.8   -0.2    0.0   -0.2   -0.2   -0.1   -0.1
sms2!   -0.1   -0.2    1.0   -0.2    0.1    0.4    0.0    0.3    0.5    0.4
sms3    0.6    0.8   -0.2    1.0   -0.2   -0.3   -0.1   -0.3   -0.2   -0.1
sms4   -0.0   -0.2    0.1   -0.2    1.0    0.2    0.0    0.1   -0.4   -0.2
sms5!   -0.3    0.0    0.4   -0.3    0.2    1.0   -0.1    0.1    0.3    0.4
sms6   -0.3   -0.2    0.0   -0.1    0.0   -0.1    1.0    0.1   -0.2   -0.2
sms7   -0.1   -0.2    0.3   -0.3    0.1    0.1    0.1    1.0    0.1    0.4
sms8!   -0.3   -0.1    0.5   -0.2   -0.4    0.3   -0.2    0.1    1.0    0.3
sms9!   -0.3   -0.1    0.4   -0.1   -0.2    0.4   -0.2    0.4    0.3    1.0
```

1

- ① Normalizing each topic vector by it's length (L2-norm) allows us to compute the cosine distances with a dot product

Reading down the "sms0" column (or across the "sms0" row) the cosine similarity between "sms0" and the spam messages ("sms2!", "sms5!", "sms8!", "sms9!") is significantly negative. That means that the topic vector for "sms0" is significantly different from the topic vector for spam messages. A non-spam message doesn't talk about the same thing as spam messages.

Doing the same for the "sms2!" column should show a positive correlation with other spam messages. Spam messages share similar semantics. They talk about similar "topics."

This is how semantic search works as well. You can use the cosine similarity between a query vector and all the topic vectors for your database of documents to find the most semantically similar message in your database. The closet document (smallest distance) to the vector for that query would correspond to the document with the closest meaning. Spaminess is just one of the "meanings" mixed into our SMS message topics.

Unfortunately, this similarity between topic vectors within each class (spam and non-spam) isn't maintained for all the messages. It would be hard to "draw a line" between the spam and non-spam messages for this set of topic vectors. You'd have a hard time setting a threshold on the similarity to an individual spam message that would ensure that you'd always be able to classify spam and non-spam correctly. But, generally, the less spammy a message is the further away it is (less similar it is) from another spam message in the dataset. That's what we need if we want to build a spam filter using these topic vectors. And a machine learning algorithm can look at all the topics individually for all the spam and non-spam labels and perhaps draw a hyperplane or other boundary between the spam and non-spam messages.

When using TruncatedSVD, you should discard the eigenvalues before computing the topic vectors. We tricked the `scikit-learn` implementation of `TruncatedSVD` into ignoring the scale information within the eigenvalues (the `Sigma` or `s` matrix in our diagrams) by:

1. Normalizing our TF-IDF vectors by their length (L2-norm)
2. Centering the TF-IDF term frequencies by subtracting the mean frequency for each term (word)

The normalization process eliminates any "scaling" or bias in the eigenvalues and focuses your SVD on the rotation part of the transformation of your TF-IDF vectors. By ignoring the eigenvalues (vector scale or length), we can "square up" the hypercube that bounds the topic vector space. This will allow us to treat all topics as equally important in our model. If you want to use this trick within your own SVD implementation you can normalize all the TF-IDF vectors by the L2-norm before computing the SVD or TruncatedSVD. The `scikit-learn` implementation of `PCA` does this for you by "centering" and "whitening" your data.

Without this normalization, infrequent topics will be given slightly more weight than they would otherwise. Since "spaminess" is a rare topic, occurring only 13% of the time, the topics that measure it would be given more weight by this normalization or eigenvalue discarding. Our resulting topics are more correlated with subtle characteristics, like spaminess, by taking this approach.

TIP

Whichever algorithm or implementation you use for semantic analysis (LSA, PCA, SVD, TruncatedSVD, or LDiA) normalize whatever BOW or TF-IDF vectors. Otherwise you may end up with large scale differences between your topics. Scale differences between topics reduce the ability of your model to differentiate between subtle, infrequent topics. Another way to think of it is that scale variation can create deep canyons and rivers in a contour plot of your objective function, making it hard for other machine learning algorithms to find the optimal thresholds on your topics in this rough terrain.

4.5 Latent Dirichlet Allocation (LDiA)

We've spent most of this chapter talking about Latent Semantic Analysis (LSA) and various ways to accomplish it using `scikit-learn` or even just plain `numpy`. Latent Semantic Analysis (LSA) should be your first choice for most topic modeling, semantic search, or content-based recommendation engines.⁷⁷ Its math is straight-forward and efficient, and it produces a linear transformation that can be applied to new batches of natural language without training, and little loss in accuracy. But now we're going to talk about a more advanced approach that can give slightly better results in some situations.

Footnote 77 A 2015 comparison of content-based movie recommendation algorithms by Sonia Bergamaschi and Laura Po found LSA to be approximately twice as accurate as LDiA:

www.dbgroup.unimo.it/~po/pubs/LNBI_2015.pdf

Latent Dirichlet Allocation (LDiA) does a lot of the things we did to create our topic models with LSA (and SVD under the hood), but unlike LSA, *LDiA* assumes a Dirichlet distribution of word frequencies. *LDiA* is more precise about the statistics of allocating words to topics than the linear math of LSA.

Latent Dirichlet Allocation creates a semantic vector space model (like our topic vectors) using an approach similar to how our brain worked during the thought experiment at the beginning. In our thought experiment, we manually allocated words to topics based on how often they occurred together in the same document. The topic mix for a document can then be determined by the word mixtures in each topic by which topic those words were assigned to. This makes an LDiA topic model much easier to understand, because the words assigned to topics and topics assigned to documents tend to make more sense than for LSA.

LDiA assumes that each document is a mixture (linear combination) of some arbitrary number of topics, that you select when you begin training the LDiA model. LDiA also assumes that each topic can be represented by a distribution of words (term frequencies). The probability or weight for each of these topics within a document, as well as the probability of a word being assigned to a topic, is assumed to start with a Dirichlet probability distribution (the *prior* if you remember your statistics). This is where the algorithm gets its name.

4.5.1 The LDiA Idea

The LDiA approach was developed in 2000 by geneticists in the UK to help them "infer population structure" from sequences of genes.⁷⁸ Stanford Researchers (including Andrew Ng) popularized the approach for NLP in 2003.⁷⁹ But don't be intimidated by the big names that came up with this approach. We'll explain the key points of it in a few lines of Python below. You only need to understand it enough to get a feel for what it's doing (an intuition), so you know what you can use it for in your pipeline.

Footnote 78 "Jonathan K. Pritchard, Matthew Stephens, Peter Donnelly, Inference of Population Structure Using Multilocus Genotype Data" www.genetics.org/content/155/2/945

Footnote 79 www.jmlr.org/papers/volume3/blei03a/blei03a.pdf

Blei and Ng came up with the idea by flipping our thought experiment on its head. They imagined how a machine that could do nothing more than roll dice (generate random numbers) could write the documents in a corpus that we want to analyze. And since we're only working with bags of words, they cut out the part about sequencing those words together to make sense, to write a real document. They just modeled the statistics for the mix of words that would become a part of a particular the BOW for each document.

They imagined a machine that only had 2 choices to make to get started generating the mix of words for a particular document. They imagined that the document generator chose those words randomly, with some probability distribution over the possible choices, like choosing the number of sides of the dice and the combination of dice you add together to create a D&D character sheet. Our document "character sheet" only needs two rolls of the dice. But the dice are very large and there are several of them, with complicated rules about how they are combined to produce the desired probabilities for the different values that we want. This is because we want very particular probability distributions for the number of words and number of topics so that it matches the distribution of these values in real documents analyzed by humans for their topics and words.

The two rolls of the dice represent:

1. Number of words to generate for the document (Poisson distribution)
2. Number of topics to mix together for the document (Dirichlet distribution)

After it has these two numbers, the hard part begins, actually choosing the words for a document. The imaginary BOW generating machine iterates over those topics and

randomly chooses words appropriate to that topic until it hits the number of words that it had decided the document should contain in step 1 above. Deciding the probabilities of those words for topics, the appropriateness of words for each topic is the hard part. But once that has been determined, our "bot" just looks up the probabilities for the words for each topic from a matrix of term-topic probabilities. If you don't remember what that matrix looks like, glance back at the simple example earlier in this chapter.

So all this machine needs is a single parameter for that Poisson distribution (in the dice roll from step 1) that tells it what the "average" document length should be, and a couple more parameters to define that Dirichlet distribution which sets up the number of topics. Then our document generation algorithm needs a term-topic matrix of all the words and topics it likes to use, its vocabulary. And it needs a mix of topics that it likes to "talk" about.

Let's flip the document generation (writing) problem back around to our original problem of estimating the topics and words from an existing document. We need to measure, or compute, those parameters about words and topics for the first 2 steps. Then we need to compute the term-topic matrix from a collection of documents. That's what LDiA does.

Blei and Ng realized that they could determine the parameters for steps 1 and 2 above by analyzing the statistics of the documents in a corpus. For example, for step 1, they could calculate the mean number of words (or n -grams) in all the bags of words for the documents in their corpus, something like this:

```
>>> total_corpus_len = 0
>>> for document_text in sms.text:
...     total_corpus_len += len(casual_tokenize(document_text))
>>> mean_document_len = total_corpus_len / len(sms)
>>> round(mean_document_len, 2)
21.35
```

Or, in a one-liner:

```
>>> sum([len(casual_tokenize(t)) for t in sms.text]) * 1. / len(sms.text)
21.35
```

Keep in mind, you should calculate this statistic directly from your BOWs. You need to make sure you're counting the tokenized and vectorized (`Counter()`-ed) words in your documents. And make sure you've applied any stopword filtering, or other normalizations before you count up your unique terms. That way your count includes all the words in your BOW vector vocabulary (all the n -grams you are counting), but only

those words that your BOWs use (not stopwords, for example). This LDiA algorithm relies on a bag-of-words vector space model, like the other algorithms in this chapter.

The second parameter you need to specify for an LDiA model, the number of topics, is a bit trickier. The number of topics in a particular set of documents can't be measured directly until after you've assigned words to those topics. So, like *K-Means* and *KNN* and other clustering algorithms you must tell it the K ahead of time. You can guess the number of topics (analogous to the K in *K-means*, the number of "clusters") and then check to see if that works for your set of documents. Once you've told LDiA how many topics to look for, it can then find the optimal mix of words to put in each topic.

Of course you can optimize this "hyperparameter" (K , the number of topics)⁸⁰ by adjusting it until it works for your application. You can automate this optimization if you can measure something about the quality of your LDiA language model for representing the meaning of your documents. One "cost function" you could use for this optimization is how well (or poorly) that LDiA model performs in some classification or regression problem, like sentiment analysis, document keyword tagging, or topic analysis. You just need some labeled documents to test your topic model or classifier on.⁸¹

Footnote 80 The symbol used by Blei and Ng for this parameter was actually *Theta* rather than K

Footnote 81 Craig Bowman, a librarian at the [University of Miami in Ohio](#), is using the Library of Congress classification system as the topic labels for Gutenberg Project books. This has to be the most ambitious and [pro-social open-science NLP project](#) I've run across so far.

4.5.2 LDiA Topic Model for SMS Messages

The topics produced by LDiA tend to be more understandable and "explainable" to humans. This is because words that frequently occur together are assigned the same topics, and humans expect that to be the case. Where LSA (PCA) tries to keep things spread apart that were spread apart to start with, LDiA tries to keep things close together that started out close together.

This may sound like it's the same thing, but it's not. The math optimizes for different things. Your optimizer has a different objective function so it will reach a different objective. To keep close high-dimensional vectors close together in the lower dimensional space, LDiA has to twist and contort the space (and the vectors) in nonlinear ways. This is a really hard thing to visualize until you actually do it on something 3D and take "projections" of the resultant vectors in 2D.

If you want to help your fellow readers out and learn something in the process, submit

some additional code to the [horse example](#) in [nlpia](#). You can create word-document vectors for each of the thousands of points in the horse by converting them to integer counts of the words "x", "y", and "z", the dimensions of the 3D vector space. You could then generate synthetic documents from these counts and pass it through all the LDiA and LSA examples above. Then you'd be able to directly visualize how each approach produces a different 2D "shadow" (projection) of the horse.

Let's see how that works for a data set of a few thousand SMS messages, labeled for spaminess. First compute the TF-IDF vectors and then some topics vectors for each SMS message (document). We'll assume use only 16 topics (components) to classify the spaminess of messages, as before. Keeping the number of topics (dimensions) low can help reduce overfitting.⁸²

Footnote 82 See Appendix D if you want to learn more about why overfitting is a bad thing

LDiA works with raw bag-of-words (BOW) count vectors rather than normalized TF-IDF vectors. Here's an easy way to compute BOW vectors in `scikit-learn`:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> from nltk.tokenize import casual_tokenize
>>> np.random.seed(42)

>>> counter = CountVectorizer(tokenizer=casual_tokenize)
>>> bow_docs = pd.DataFrame(counter.fit_transform(raw_documents=sms.text).toarray(), index=index)
>>> column_nums, terms = zip(*sorted(zip(counter.vocabulary_.values(), counter.vocabulary_.keys())))
>>> bow_docs.columns = terms
```

Let's double check that our counts make sense for that first SMS message labeled "sms0":

```
>>> sms.loc['sms0'].text
'Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there got amor
>>> bow_docs.loc['sms0'][bow_docs.loc['sms0'] > 0].head()
'
          1
...
          2
amore      1
available   1
Name: sms0, dtype: int64
```

And here's how to use Latent Dirichlet Allocation (LDiA) to create topic vectors for our SMS corpus.

```
>>> from sklearn.decomposition import LatentDirichletAllocation as LDiA
>>> ldia = LDiA(n_components=16, learning_method='batch')
>>> ldia = ldia.fit(bow_docs)
```

```
>>> ldia.components_.shape
(16, 9232)
```

- ① LDiA takes a bit longer than PCA or SVD, especially for a large number of topics and a large number of words in your corpus.

So our model has allocated our 9,232 words (terms) to 16 topics (components). Let's take a look at the first few words and how they're allocated to our 16 topics. Keep in mind that your counts and topics will be different from mine. LDiA is a stochastic algorithm that relies on the random number generator to make some of the statistical decisions it has to make about allocating words to topics. So your topic-word weights will be different from those below, but they should have similar magnitudes. Each time you run `sklearn.LatentDirichletAllocation` (or any LDiA algorithm) you will get different results unless you set the random seed to a fixed value.

```
>>> components = pd.DataFrame(ldia.components_.T, index=terms, columns=columns)
>>> components.round(2).head()
   topic0  topic1  topic2  topic3  topic4  ...  topic12  topic13  topic14  topic15
!
  184.03   15.00   72.22  394.95   45.48  ...
  "       0.68    4.22    2.41   0.06  152.35  ...
  #       0.06    0.06    0.06   0.06    0.06  ...
#150     0.06    0.06    0.06   0.06    0.06  ...
#5000    0.06    0.06    0.06   0.06    0.06  ...
```

So the exclamation point term ("!") was allocated to most of the topics, but is a particularly strong part of `topic3` where the quote symbol ("") is hardly playing a role at all. Perhaps "`topic3`" might be about emotional intensity or emphasis and doesn't care much about numbers or quotes. Let's see.

```
>>> components.topic3.sort_values(ascending=False)[:10]
!
  394.952246
.
  218.049724
to
  119.533134
u
  118.857546
call
  111.948541
f
  107.358914
,
  96.954384
*
  90.314783
your
  90.215961
is
  75.750037
```

So the top ten tokens for this topic seem to be the type of words that might be used in emphatic directives requesting someone to do something or pay something. It will be interesting to find out if this topic is used more in spam messages rather than non-spam messages. You can see that the allocation of words to topics can be rationalized or reasoned about, even with this quick look.

Before we fit our LDA classifier, we need to compute these LDiA topic vectors for all our documents (SMS messages). And let's see how they are different from the topic vectors produced by SVD and PCA for those same documents.

```
>>> ldial6_topic_vectors = ldia.transform(bow_docs)
>>> ldial6_topic_vectors = pd.DataFrame(ldial6_topic_vectors, index=index, columns=columns)
>>> ldial6_topic_vectors.round(2).head()
   topic0  topic1  topic2  topic3  ...  topic12  topic13  topic14  topic15
sms0    0.00    0.62    0.00    0.00  ...    0.00    0.00    0.00    0.00
sms1    0.01    0.01    0.01    0.01  ...    0.01    0.01    0.01    0.01
sms2!   0.00    0.00    0.00    0.00  ...    0.00    0.00    0.00    0.00
sms3    0.00    0.00    0.00    0.00  ...    0.00    0.00    0.00    0.00
sms4    0.39    0.00    0.33    0.00  ...    0.09    0.00    0.00    0.00
```

You can see that these topics are more cleanly separated. There are a lot of zeros in our allocation of topics to messages. This is one of the things that makes LDiA topics easier to explain to coworkers when making business decisions based on your NLP pipeline results.

So LDiA topics work well for humans, but what about machines? How will our LDA classifier fare with these topics?

4.5.3 LDiA + LDA = *spam Classifier*

Let's see how good these LDiA topics are at predicting something useful, like spaminess. We'll use our LDiA topic vectors to train a Linear Discriminant Analysis model again (like we did with our PCA topic vectors).

```
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
>>> X_train, X_test, y_train, y_test = train_test_split(ldial6_topic_vectors,
...           sms.spam, test_size=0.5, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(X_train, y_train) ①
>>> sms['ldial6_spam'] = lda.predict(ldial6_topic_vectors)
>>> round(float(lda.score(X_test, y_test)), 2) ②
0.94
```

- ➊ Our `ldia_topic_vectors` matrix has as determinant close to zero so you will likely get the warning "Variables are collinear." This can happen with a small corpus when using LDiA because our topic vectors have a lot of zeros in them and some of our messages could be reproduced as a linear combination of the other messages topics. Or there are some SMS messages with very similar (or identical) topic mixes.
- ➋ 94% accuracy on the test set is pretty good, but not quite as good as LSA (PCA) from before.

The algorithms for `train_test_split()` and `LDiA` are stochastic. So each time you run it you will get different results, different accuracy values. If you want to make your pipeline repeatable, look for the `seed` argument for these models and dataset splitters. You can set the seed to the same value with each run to get reproducible results.

One way a "collinear" warning can occur is if our text has a few 2-grams or 3-grams where their component words only ever occur together. So the resulting LDiA model had to arbitrarily split the weights among these equivalent term frequencies. Can you find the words in our SMS messages that is causing this "collinearity" (zero determinant)? You're looking for a word that whenever it occurs, another word (it's pair) is always in the same message.

This search can be done with Python rather than by hand. First, you probably just want to look for any identical bag-of-words vectors in your corpus. These could occur for SMS messages that aren't identical like "Hi there Bob!" or "Bob, Hi there", because they have the same word counts. You can iterate through all the pairings of the bags of words to look for identical vectors. These will definitely cause a "collinearity" warning in either LDiA or LSA.

If you don't find any exact BOW vector duplicates, you could iterate through all the pairings of the words in our vocabulary. You'd then iterate through all the bags of words to look for the pairs of SMS messages that contain those exact same two words. If there aren't any times that those words occur separately in the SMS messages, then you've found one of the "collinearities" in our data set. Some common 2-grams that might cause this are the first and last names of famous people that always occur together and are never used separately, like "Bill Gates" (as long as there are no other "Bill's" in your SMS messages).

TIP

Whenever you need to iterate through all the combinations (pairs or triplets) of a set of objects you can use the built-in Python `product()` function.

```
>>> from itertools import product
>>> all_pairs = [(word1, word2) for (word1, word2) in product(word_list, word_list) if i
```

We got more than 90% accuracy on our test set and we only had to train on half of our available data. But we did getting a warning about our features being collinear. This is due to our limited dataset which gives LDA an "under-determined" problem. The determinant of our topic-document matrix is close to zero, once we discard half of the

documents with `train_test_split`. If we ever need to, we can turn down the LDiA `n_components` to "fix" this. This would tend to combine those topics together that are a linear combination of each other (collinear).

But let's find out how our LDiA model compares to a much higher-dimensional model based on the TF-IDF vectors. Our TF-IDF vectors have many more features (more than 3000 unique terms). So we're likely to experience overfitting and poor generalization. This is where the generalization of LDiA and PCA should help us.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from nltk.tokenize.casual import casual_tokenize
>>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray()
>>> tfidf_docs = tfidf_docs - tfidf_docs.mean(axis=0) ❶
>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs, sms.spam.values, test_size=0.5, random_state=42)
>>> lda = LDA(n_components=1) ❷
>>> lda = lda.fit(X_train, y_train)
>>> round(float(lda.score(X_train, y_train)), 3)
1.0
>>> round(float(lda.score(X_test, y_test)), 3)
0.748
```

- ❶ We're going to "pretend" that there is only 1 topic in all the SMS messages because we're only interested in a scalar score for the "spamminess" topic.
- ❷ Fitting an LDA model to all these thousands of features will take quite a long time. Be patient, it's slicing up your vector space with a 9332-D hyperplane!

The training set accuracy for our TF-IDF based model is perfect! But the test set accuracy is much worse than when we trained it on lower dimensional topic vectors instead of TF-IDF vectors.

And test set accuracy is the only accuracy that really counts. This is exactly what topic modeling (LSA) is supposed to do. It helps us generalize our models from a small training set so it still works well on messages using different combinations of words (but similar topics).

4.5.4 A Fairer Comparison: 32 LDiA Topics

Let's try one more time with more dimensions, more topics. Perhaps LDiA isn't as efficient as LSA (PCA) so it needs more topics to allocate words to. Let's try 32 topics (components).

```
>>> ldia32 = LDiA(n_components=32, learning_method='batch')
>>> ldia32 = ldia32.fit(bow_docs)
```

```
>>> ldia32.components_.shape
(32, 9232)
```

Now let's compute our new 32-D topic vectors for all our documents (SMS messages).

```
>>> ldia32_topic_vectors = ldia32.transform(bow_docs)
>>> columns32 = ['topic{}'.format(i) for i in range(ldia32.n_components)]
>>> ldia32_topic_vectors = pd.DataFrame(ldia32_topic_vectors, index=index, columns=columns32)
>>> ldia32_topic_vectors.round(2).head()
   topic0  topic1  topic2  ...  topic29  topic30  topic31
sms0    0.00    0.5    0.0  ...      0.0    0.0    0.0
sms1    0.00    0.0    0.0  ...      0.0    0.0    0.0
sms2!   0.00    0.0    0.0  ...      0.0    0.0    0.0
sms3    0.00    0.0    0.0  ...      0.0    0.0    0.0
sms4    0.21    0.0    0.0  ...      0.0    0.0    0.0
```

You can see that these topics are even more sparse, more cleanly separated.

And here's our LDA model (classifier) training, this time using 32-D LDiA topic vectors.

```
>>> X_train, X_test, y_train, y_test = train_test_split(ldia32_topic_vectors,
   sms.spam, test_size=0.5, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(X_train, y_train)
>>> sms['ldia32_spam'] = lda.predict(ldia32_topic_vectors)
1
>>> X_train.shape
(2418, 32)
>>> round(float(lda.score(X_train, y_train)), 3)
0.924
2
>>> round(float(lda.score(X_test, y_test)), 3)
0.927
```

- ➊ .shape is another way to check the number of dimensions in our topic vectors
- ➋ Test accuracy is what really matters, and 92.7% is a bit better than the 90.1% we got with 16-D LDiA topic vectors.

So accuracy does indeed improve by almost 3% (up from 90% to almost 93%) if we give LDiA 32 "buckets" (topics) to allocate words to, instead of only 16.

Don't confuse this optimization of the number of "topics" or components with the collinearity problem earlier. Increasing or decreasing the number of topics doesn't fix or create the collinearity problem. That's a problem with the underlying data. If you want to get rid of that warning you need to add "noise" or meta data to your SMS messages as synthetic words, or you need to delete those duplicate word vectors. If you have duplicate word vectors or word pairings that repeat a lot in your documents then no amount of topics is going to fix that.

The larger number of topics allows it to be more precise about topics, and, at least for this

data set, product topics that linearly separate better. But this performance still is not quite as good as the 96% accuracy of PCA + LDA. So PCA is keeping our SMS topic vectors spread out more efficiently, allowing for a wider gap between messages to cut with a hyperplane to separate classes.

Feel free to explore the source code for the Dirichlet Allocation models available in both `scikit-learn` as well as `gensim`. They have an API very similar to LSA (`sklearn.TruncatedSVD` and `gensim.LsiModel`). We'll show you an example application when we talk about summarization in later chapters. Finding explainable topics, like those used for summarization, is what LDiA is good at. And it's not too bad at creating topics useful for linear classification.

TIP

Remember you can find the source code path with the hidden `__file__` attribute on any Python module, like `sklearn.__file__`. And in `ipython`, you can view the source code for any function, class, or object with `??`, like this, `LDiA??`

```
>>> import sklearn
>>> sklearn.__file__
'/Users/hobs/anaconda3/envs/conda_env_nlpia/lib/python3.6/site-packages/sklearn/__init__.py'
```

4.6 Distance and Similarity

We need to revisit those similarity scores we talked about in Chapter 2 and 3 to make sure our new topic vector space works with them. Remember that similarity scores (and distances) can be used to tell how similar or far apart two documents are based on the similarity (or distance) of the vectors we used to represent them.

We can use similarity scores (and distances) to see how well our LSA topic model agrees with the higher-dimensional TF-IDF model of Chapter 3. We'll see how good our model is at retaining those distances after having eliminated a lot of the information contained in the much higher-dimensional bags of words. We can check how far away from each other the topic vectors are and whether that's a good representation of the distance between the documents' subject matter. We want to check that documents which mean similar things are close to each other in our new topic vector space.

LSA preserves large distances, but does not always preserve close distances (the fine "structure" of the relationships between our documents). This is because the underlying SVD algorithm is focused on maximizing the variance between all our documents in the new topic vector space.

Distances between feature vectors (word vectors, topic vectors, document context vectors, etc) drive the performance of an NLP pipeline, or any machine learning pipeline. So what are our options for measuring distance in high-dimensional space? And which ones should we chose for a particular NLP problem? Here are a few commonly used examples. Some may be familiar from geometry class or linear algebra, but many others are probably new to you.

- Euclidean or Cartesian distance, or Root Mean Square Error (RMSE): 2-norm or L₂
- Squared Euclidean distance, sum of squares distance (SSD): L₂²
- Cosine or angular or projected distance: normalized dot product
- Minkowski distance: p-norm or L_p
- Fractional distance, fractional norm: p-norm or L_p for $0 < p < 1$
- City block, Manhattan, or taxicab distance, sum of absolute distance (SAD): 1-norm or L₁
- Jaccard distance, inverse set similarity
- Mahalanobis distance
- Levenshtein or edit distance

The variety of ways to calculate distance is a testament to how important it is. In addition to the pairwise distance implementations in `Sciki-learn` there are many others used in mathematics specialties like topology, statistics, and engineering.⁸³ For reference, here are all the distances you can find in the `sklearn.metrics.pairwise` module:⁸⁴

Footnote 83 other distance metrics: numerics.mathdotnet.com/Distance.html

Footnote 84 docs for `sklearn.metrics.pairwise`:
scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html

Listing 4.7 Pairwise distances available in `sklearn`

```
'cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan', 'braycurtis', 'canberra',
'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis',
'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener',
'sokalsneath', 'sgeuclidean', 'yule'
```

Distance measures are often computed from similarity measures (scores) and vice versa such that distances are inversely proportional to similarity scores. Similarity scores are designed to range between 0 and 1. Typical conversion formulas look like this:

```
>>> similarity = 1. / (1. + distance)
>>> distance = (1. / similarity) - 1.
```

But for distances and similarity scores that range between 0 and 1, like probabilities, it's

more common to use a formula like this:

```
>>> similarity = 1. - distance
>>> distance = 1. - similarity
```

And cosine distances have their own convention for the range of values they use. The angular distance between two vectors is computed as a fraction of the maximum possible angular distance between two vectors, which is 180 degrees or π radians.⁸⁵ As a result angular distances are often expressed in actual radians of angle between two vectors.

Footnote 85 en.wikipedia.org/wiki/Cosine_similarity

```
>>> import math
>>> angular_distance = math.acos(cosine_similarity) / math.pi
>>> distance = 1. / similarity - 1.
>>> similarity = 1. - distance
```

The terms *distance* and *length* are often confused with the term *metric* because many *distances* and *lengths* are valid and useful *metrics*. But unfortunately not all distances qualify to be called *metrics*. Even more confusing, *metrics* are also sometimes called *distance functions* or *distance_metrics* in formal mathematics and set theory texts.⁸⁶

Footnote 86 [en.wikipedia.org/wiki/Metric_\(mathematics\)](https://en.wikipedia.org/wiki/Metric_(mathematics))

DEFINITION A true *metric* must have four mathematical properties that distances or "scores" do not:

1. nonnegativity: *metrics* can never be negative `metric(A, B) >= 0`
2. indiscernibility: two objects are identical if the *metric* between them is zero
`if metric(A, B) == 0: assert(A == B)`
3. symmetry: *metrics* don't care about direction `metric(A, B) = metric(B, A)`
4. triangle inequality: you can't get from A to C faster by going through B in between `metric(A, C) < metric(A, B) + metric(B, C)`

A related mathematical term, *measure*, has both a natural English meaning and a rigorous mathematical definition. You'll find "measure" in both a Merriam Webster dictionary and a math textbook glossary. And the definitions will be completely different. So be careful when talking to your math professor.

To a math professor, a *measure* is the size of a set of mathematical objects. You can

measure a Python set by it's length, but many mathematical sets are infinite. And in set theory things can be infinite in different ways. And *measures* are all the different ways to calculate the `len()` or size of a mathematical set, the ways things are infinite.

DEFINITION

Like *metric*, the word *measure* has a very precise mathematical definition, related to the "size" of a collection of objects. So the word "measure" should also be used carefully in describing any scores or statistics derived from an object or combination of objects in NLP.⁸⁷

Footnote 87 [en.wikipedia.org/wiki/Measure_\(mathematics\)](https://en.wikipedia.org/wiki/Measure_(mathematics))

But in the real world we measure all sorts of things. When we use it as a verb we just mean using a measuring tape, or a ruler, or a scale or a score, to measure something. That's how we use the word "measure" in this book, but we'll try not to use it at all, so that our math professors don't scold us.

4.7 Steering with Feedback

All of the previous approaches to LSA failed to take into account information about the similarity between documents. We created topics that were optimal for a generic set of rules. Our unsupervised learning of these feature (topic) extraction models didn't have any data about how "close" the topic vectors should be to each other. We didn't allow any "feedback" about where the topic vectors ended up, or how they were related to each other.

Steering or "learned distance metrics"⁸⁸ are the latest advancement in dimension reduction and feature extraction. By adjusting the distance scores reported to clustering and embedding algorithms, it's possible to "steer" the your vectors so that they minimize some cost function. In this way you can force your vectors to focus on some aspect of the information content that you are interested in.

Footnote 88 users.cecs.anu.edu.au/~sgould/papers/eccv14-spgraph.pdf

In the previous sections about LSA we ignored all the meta information about our documents. For example, with the SMS messages we ignored the sender of the message. This is a good indication of topic similarity and could be used to inform our topic vector transformation (LSA).

At Talentpair we experimented with matching resumes to job descriptions using the cosine distance between topic vectors for each document. This worked OK. But we

learned pretty quickly that we got much better results when we started "steering" our topic vectors based on feedback from candidates and account managers responsible for helping them find a job. Vectors for "good pairings" were steered closer together than all the other pairings.

One way to do this is to calculate the mean difference between your two centroids (like we did for LDA) and add some portion of this "bias" to all your resume or job description vectors. This should take out the average topic vector difference between resumes and job descriptions. Topics like beer on tap at lunch might appear in a job description but never in a resume. Similarly bizarre person hobbies, like underwater sculpture might appear in some resumes but never a job description. Steering your topic vectors can help you focus them on the topics you are interested in modeling.

If you're interested in refining topic vectors, taking out bias, you can search [Google Scholar](#) for "learned distance/similarity metric" or "distance metrics for nonlinear embeddings."⁸⁹ Unfortunately there are no `scikit-learn` modules that implement this yet. You'd be a hero if you found the time to add some "steering" feature suggestions or code to the [Sciki-Learn project](#).

Footnote 89 www.cs.cmu.edu/~liuy/frame_survey_v2.pdf

4.7.1 Linear Discriminant Analysis (LDA)

Lets train a linear discriminant analysis model on our labeled SMS messages. LDA works similarly to LSA, except it requires classification labels or other scores to be able to find the best linear combination of the dimensions in high dimensional space (the terms in a BOW or TFIDF vector). Rather than maximizing the separation (variance) between all vectors in the new space, LDA maximizes the distance between the centroids of the vectors within each class.

Unfortunately, this means you have to tell the LDA algorithm what "topics" you'd like to model by giving it examples (labeled vectors). Only then can the algorithm compute the optimal transformation from your high dimensional space to the lower dimensional space. And the resulting lower-dimensional vector can't have any more dimensions than the number of class labels or scores that you are able to provide. Since we only have a "spaminess" topic to train on, let's see how accurate our 1-dimensional topic model can be at classifying spam SMS messages.

```
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(tfidf_docs, sms.spam)
>>> sms['lda_spaminess'] = lda.predict(tfidf_docs)
```

```
>>> ((sms.spam - sms.lda_spaminess) ** 2.).sum() ** .5
0.0
>>> (sms.spam == sms.lda_spaminess).sum()
4837
>>> len(sms)
4837
```

It got every single one of them right! Oh, wait a minute. What did we say earlier about overfitting? With 10,000 terms in our TF-IDF vectors it's not surprising at all that it could just "memorize" the answer. Let's do some cross validation this time.

```
>>> from sklearn.model_selection import cross_val_score
>>> lda = LDA(n_components=1)
>>> scores = cross_val_score(lda, tfidf_docs, sms.spam, cv=5)
>>> "Accuracy: {:.2f} (+/- {:.2f})".format(scores.mean(), scores.std() * 2)
'Accuracy: 0.76 (+/- 0.03)'
```

Clearly this is not a good model. This should be a reminder to never get excited about a model's performance on your training set.

Just to make sure that 76% accuracy number is correct, let's reserve a third of our dataset for testing:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs, sms.spam,
   test_size=0.33, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda.fit(X_train, y_train)
LinearDiscriminantAnalysis(n_components=1, priors=None, shrinkage=None,
   solver='svd', store_covariance=False, tol=0.0001)
>>> lda.score(X_test, y_test).round(3)
0.765
```

Again, poor testset accuracy. So it doesn't look like we're just getting unlucky with our sampling of the data. We just have a poor, overfitting model.

Let's see if LSA combined with LDA will help us create an accurate model that is also generalized well so that new SMS messages don't trip it up.

```
>>> X_train, X_test, y_train, y_test = train_test_split(pca_topicvectors.values,
   sms.spam, test_size=0.3, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda.fit(X_train, y_train)
LinearDiscriminantAnalysis(n_components=1, priors=None, shrinkage=None,
   solver='svd', store_covariance=False, tol=0.0001)
>>> lda.score(X_test, y_test).round(3)
0.965
>>> lda = LDA(n_components=1)
>>> scores = cross_val_score(lda, pca_topicvectors, sms.spam, cv=10)
>>> "Accuracy: {:.3f} (+/- {:.3f})".format(scores.mean(), scores.std() * 2)
'Accuracy: 0.958 (+/- 0.022)'
```

So with LSA we can characterize an SMS message with only 16 dimensions and still have plenty of information to classify them as spam or not. And our low-dimensional model is much less likely to overfit. It should generalize well and be able to classify as yet unseen SMS messages or chats.

We've now come full circle back to our "simple" model at the beginning of this chapter. We actually got better accuracy with our simple LDA model before we tried all that semantic analysis. But the advantage of this new model is that we now can create vectors that represent the semantics of a statement in more than just a single dimension.

4.8 Topic Vector Power

With topic vectors we can do things like compare the meaning of words, documents, statements and corpora. We can find "clusters" of similar documents and statements. We're no longer comparing the distance between documents based merely on their word usage. We're no longer limited to keyword search and relevance ranking based entirely on word choice or vocabulary. We can now find documents that are actually relevant to our query, not just a good match for the word statistics themselves.

This is called "semantic search", not to be confused with the "semantic web."⁹⁰ Semantic search is what strong search engines do when they give you documents that don't contain many of the words in your query, but are exactly what you were looking for. These advanced search engines use LSA topic vectors to tell the difference between a Python package in "The Cheese Shop" and python in a Florida pet shop aquarium, while still recognizing its similarity to a "Ruby gem."⁹¹

Footnote 90 The semantic web is the practice of structuring natural language text with the use of tags in an HTML document so that the hierarchy of tags and their content provide information about the relationships (web of connections) between elements (text, images, videos) on a web page.

Footnote 91 Ruby is a programming language with packages called `gem`s

That will give us a tool for finding and generating meaningful text. But our brains are not very good at dealing with high dimensional objects, vectors, hyperplanes, hyperspheres, hypercubes. Our intuitions as developers and machine learning engineers breaks down above 3 dimensions.

For example, to do a query on a 2D vector, like our lat/lon location on Google Maps, we can quickly find all the coffee shops nearby without much searching. We can just scan (with our eyes or with code) near our location and spiral outward with our search. Alternatively, we can create bigger and bigger bounding boxes with our code, checking

for longitudes and latitudes within some range on each, that's just for comparison operations and that should find us everything nearby. Doing this in hyperspace with hyperplanes and hypercubes to form the boundaries of our search is impossible.

As Geoffrey Hinton says, "To deal with hyper-planes in a 14-dimensional space, visualize a 3D space and say 14 to yourself very loudly." If you read Abbott's 1884 *Flatland* when you were young and impressionable you might be able to do a little bit better than this hand waving. You might even be able to poke your head partway out of the window of our 3D world into hyperspace, enough to catch a glimpse of that 3D world from the outside. Like in *Flatland*, we used a lot of 2D visualizations in this chapter to help you explore the shadows that words in hyperspace leave in our 3D world. If you're anxious to check them out, skip ahead to the section showing "scatter matrices" of word vectors. You might also want to glance back at the 3D bag-of-words vector in the previous chapter and try to imagine what those points would look like if you added just one more word to your vocabulary to create a 4-D world of language meaning.

If you're taking a moment to think deeply about 4 dimensions, keep in mind that the explosion in complexity you are trying to wrap your head around is even greater than the complexity growth from 2D to 3D and exponentially greater than the growth in complexity from a 1D world of numbers to a 2D world of triangles, squares, and circles.

NOTE

The explosive growth in possibilities from 1D lines, 2D rectangles, 3D cubes, and on passes through bizarre universes with non-integer fractal dimensions, like a 1.5-dimension fractal. A 1.5-D fractal has infinite length and completely fills a 2D plane while having less than two dimensions!⁹² But fortunately these aren't "real" dimensions.⁹³ So we don't have to worry about them in NLP... unless you get interested in fractional distance metrics, like p -norm, which have noninteger exponents in their formula.⁹⁴

Footnote 92 fractional dimensions,
www.math.cornell.edu/~erin/docs/research-talk.pdf

Footnote 93 "fractal dimensions",
q-what-are-fractional-dimensions-can-space-have-a-fractional-dimension/

Footnote 94 [The Concentration of Fractional Distances](#)

4.8.1 Semantic Search

When you search for a document based on a word or partial word that it contains, that's called *full text search*. This is what search engines do. They break a document into chunks (usually words) that can be indexed with an *inverted index* like you'd find at the back of a textbook. It takes a lot of bookkeeping and guesswork to deal with spelling errors and typos, but it works pretty well.⁹⁵

Footnote 95 A full text index in a database like PostgreSQL is usually based on trigrams of characters, to deal with spelling errors and text that doesn't parse into words.

Semantic search is *full text search* that takes into account the meaning of the words in your query and the documents you are searching. In this chapter you've learned two ways to compute topic vectors that capture the semantics (meaning) of words and documents in a vector, LSA and LDiA. One of the reasons that Latent Semantic Analysis (LSA) was first called Latent Semantic *Indexing* was because it promised to power semantic search with an index of numerical values, like BOW and TF-IDF tables. It was the next big thing in Information Retrieval, semantic search.

But unlike BOW and TF-IDF tables, tables of semantic vectors can't be easily discretized and indexed using traditional inverted index techniques. Traditional indexing approaches work with binary word occurrence vectors, discrete vectors (BOW vectors), sparse continuous vectors (TF-IDF vectors), and low dimensional continuous vectors (3D GIS data). But high-dimensional continuous vectors, like topic vectors from LSA or LDiA, are a challenge.⁹⁶ Inverted indexes work for discrete vectors or binary vectors, like tables of binary or integer word-document vectors, because the index only needs to maintain an entry for each nonzero discrete dimension. Either that value of that dimension is present or not present in the referenced vector or document. Since TF-IDF vectors are sparse, mostly zero, we don't need an entry in our index for most dimensions for most documents.⁹⁷

Footnote 96 Clustering high dimensional data is equivalent to discretizing or indexing high-dimensional data with bounding boxes: en.wikipedia.org/wiki/Clustering_high-dimensional_data

Footnote 97 en.wikipedia.org/wiki/Inverted_index

LSA (and LDiA) produce topic vectors that are high-dimensional, continuous, and dense (zeros are rare). And the semantic analysis algorithm does not produce an efficient index for scalable search. In fact, the curse of dimensionality that we talked about in the previous section makes an exact index impossible. The "Indexing" part of Latent Semantic Indexing (LSI) was a hope, not a reality, so the LSI term is a misnomer.

Perhaps that is why LSA has become the more popular way to describe semantic analysis algorithms that produce topic vectors.

One solution to the challenge of high-dimensional vectors is to index them with a *Locality Sensitive Hash* (LSH). A locality sensitive hash is like a Zip Code (Postal Code) that designates a region of hyperspace so that it can easily be found again later. And like a regular hash, it is discrete and only depends on the values in the vector. But even this doesn't work well once you exceed about 12 dimensions. In the table below, each row represents a semantic vector size, starting with 2 dimensions and working up to 16 dimensions, like the vectors we used earlier for the SMS spam problem.

| Dimensions | 100th Cosine Distance | Top 1 Correct | Top 2 Correct | Top 10 Correct | Top 100 Correct |
|------------|-----------------------|---------------|---------------|----------------|-----------------|
| 2 | .00 | TRUE | TRUE | TRUE | TRUE |
| 3 | .00 | TRUE | TRUE | TRUE | TRUE |
| 4 | .00 | TRUE | TRUE | TRUE | TRUE |
| 5 | .01 | TRUE | TRUE | TRUE | TRUE |
| 6 | .02 | TRUE | TRUE | TRUE | TRUE |
| 7 | .02 | TRUE | TRUE | TRUE | FALSE |
| 8 | .03 | TRUE | TRUE | TRUE | FALSE |
| 9 | .04 | TRUE | TRUE | TRUE | FALSE |
| 10 | .05 | TRUE | TRUE | FALSE | FALSE |
| 11 | .07 | TRUE | TRUE | TRUE | FALSE |
| 12 | .06 | TRUE | TRUE | FALSE | FALSE |
| 13 | .09 | TRUE | TRUE | FALSE | FALSE |
| 14 | .14 | TRUE | FALSE | FALSE | FALSE |
| 15 | .14 | TRUE | TRUE | FALSE | FALSE |
| 16 | .09 | TRUE | TRUE | FALSE | FALSE |

Figure 4.6 Semantic search accuracy deteriorates at around 12D

The table shows how good your search results would be if you used Locality Sensitive Hashing to index a large number of semantic vectors. Once your vector had more than 16 dimensions, you'd have a hard time returning 2 search results that were any good.

So how can we do semantic search on 100D vectors without an index? You now know how to convert the query string into a topic vector using LSA. And you know how to compare two vectors for similarity using the cosine similarity score (the scalar product, inner product, or dot product) to find the closest match. To find precise semantic matches, you need to find all of the closest document topic vectors to a particular query (search) topic vector. However, if we have N documents, we have to do N comparisons with our query topic vector. That's a lot of dot products. We can vectorize the operation

in numpy using matrix multiplication, but that doesn't reduce the number of operations, only makes them 100 times faster.⁹⁸ Fundamentally, exact semantic search still requires $O(N)$ multiplications and additions for each query. So it scales only linearly with the size of your corpus. That wouldn't work for a large corpus, like Google Search or even Wikipedia semantic search.

Footnote 98 Vectorizing your python code, especially doubly-nested for loops for pairwise distance calculations can speed your code by almost 100 fold:

hackernoon.com speeding-up-your-code-2-vectorizing-the-loops-with-numpy-e380e939bed3

Fortunately a lot of smart people worked for decades to solve this problem. They came up with hash-map indexing approach called Locality Sensitive Hashing (LSH) which we describe in Appendix H. Locality Sensitive Hashes are like zip codes for high-dimensional data. They won't tell you how close two continuous vectors are to each other. And the clusters of locations (vectors) within each zip code (hash) won't work for vectors near the borders of zip codes (hashes). But they are fast. And when you're looking for *most* of the locations (vectors) near your query, they work. There are now several open source implementations of some very efficient and accurate Locality Sensitive Hashing (LSH) algorithms. A couple of the easiest to use and install are

- Spotify's Annoy package⁹⁹

Footnote 99 Spotify's researchers compared their annoy performance to that of several alternative algorithms and implementations: github.com/spotify/annoy

- Gensim's `gensim.models.KeyedVector` class.¹⁰⁰

Footnote 100 The approach used in gensim for 100s of dimensions in word vectors will work fine for any semantic or topic vector: radimrehurek.com/gensim/models/keyedvectors.html

Technically these indexing or hashing solutions cannot guarantee that you will find all of the best matches for your semantic search query. But they can get you a really good list of close matches almost as fast as with a conventional reverse index on a TF-IDF vector or Bag-of-Words vector. if you're willing to give up a little precision.¹⁰¹

Footnote 101 If you want to learn about faster ways to find high dimensional vector nearest neighbors, check out Appendix H on Locality Sensitive Hashes. or use the Spotify annoy package to index your topic vectors.

4.9 Summary

In this chapter you've learned how to

- Use SVD to decompose and transform TF-IDF and BOW vector spaces

- Use LSA to compute topic vectors that capture the meaning of text
- Use LDiA to compute explainable topic vectors
- Search for text semantically—find documents or statements based on their meaning
- Use topic vectors to predict whether a social post will be "liked"
- Use topic vectors to classify spam
- How to sidestep the curse of dimensionality

In the next chapters, you'll learn how to fine tune this concept of topic vectors so that the vectors associated with words are more precise and useful. To do this we'll first start learning about neural nets. This will improve your pipeline's ability to extract meaning from short texts or even solitary words.

Deeper Learning (Neural Networks)

P2

In Part One, we gathered the tools for Natural Language Processing and dove into machine learning with statistics-driven vector space models. Towards the end of Part One we found that there was even more meaning to be found when we looked at the statistics of connections between words.¹⁰² We learned about algorithms like Latent Semantic Analysis (LSA) that can help make sense of those connections by gathering words into topics.

Footnote 102 *Conditional probability* is one term for these connection statistics (how often a word occurs given that other words occur before or after the "target" word). *Cross correlation* is another one of these statistics (the likelihood of words occurring together). The *singular values* and *singular vectors* of the word-document matrix can be used to collect words into topics, linear combinations of word counts.

But in Part One we only considered linear relationships between words. And we often had to use our human judgment to design feature extractors and select model parameters. The neural networks of Part Two will accomplish most of the tedious feature extraction work for us. And the models of Part Two will often be more accurate than those we could build with the hand-tuned feature extractors of Part One.

The use of multi-layered neural networks for machine learning is called *Deep Learning*. This new approach to Natural Language Processing and the modeling of human thought is often called "connectionism" by philosophers and neuroscientists.¹⁰³ The increasing access to Deep Learning, through greater availability of computational resources and the rich open source culture, will be our gateway into deeper understanding of language. In this section, we will begin to peel open the "black box" that is Deep Learning and learn how to model text in deeper nonlinear ways.

Footnote 103 plato.stanford.edu/entries/connectionism

We will start with a primer on *Neural Networks*. The following chapters will then examine a few of the various flavors of Neural Networks and how they can be applied to Natural Language Processing. We will also start to look at the patterns not just between words but between the characters within words. And finally we will use machine learning to actually generate novel text. :leveloffset: +1

5

Baby Steps with Neural Networks (Perceptrons and Backpropagation)

In this chapter

- Learning the History of Neural Networks
- Stacking Perceptrons
- Understanding Backpropagation
- Seeing the knobs to turn on Neural Networks
- Implementing a basic Neural Network in Keras

In recent years, a lot of hype has developed around the promise of Neural Networks and their ability to classify and identify input data, and more recently the ability of certain network architectures to generate original content. Companies large and small are using them for everything from image captioning and self-driving cars to identifying solar panels from satellite images and facial recognition. And luckily for us, there are many NLP applications of neural nets as well. While Deep Neural Networks have inspired a lot of hype and hyperbole, our robot overlords are probably further off than any click bait cares to admit. They are, however, quite powerful tools and we can easily use them in an NLP chatbot pipeline to classify input text, summarize documents, and even generate novel works.

This chapter is intended as a primer for those with no experience in Neural Networks. We will not be covering anything specific to NLP in this chapter but having a basic understanding of what is going on under the hood in a Neural Network will be important for the upcoming chapters. If you are familiar with the basics of a Neural Network you can rest easy in skipping ahead to the next chapter where we dive back into processing text with the various flavors of neural nets. While the mathematics of the underlying algorithm, *backpropagation*, are outside the scope of this book, we feel that a high level

feeling for its basic functionality is important to understand how it will help us and our computers understand language and the patterns hidden within.

TIP

Manning publishes two other books on Deep Learning that are tremendous resources for this very topic.

Deep Learning with Python, by François Chollet (www.manning.com/books/deep-learning-with-python) is a deep dive into the wonders of deep learning by the creator of Keras, himself.

Grokking Deep Learning, by Andrew Trask (www.manning.com/books/grokking-deep-learning) is a broad overview of Deep Learning models and practices.

5.1 Neural Networks, the Ingredient List

As the availability of processing power and memory has exploded over the course of the decade, an old technology has come into its own again. First proposed in the 50's by Frank Rosenblatt, the Perceptron¹⁰⁴ offered a novel algorithm for finding patterns in data.

Footnote 104 Rosenblatt, Frank (1957), The Perceptron—a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

The basic concept lies in a rough mimicry of operation of a living neuron cell.

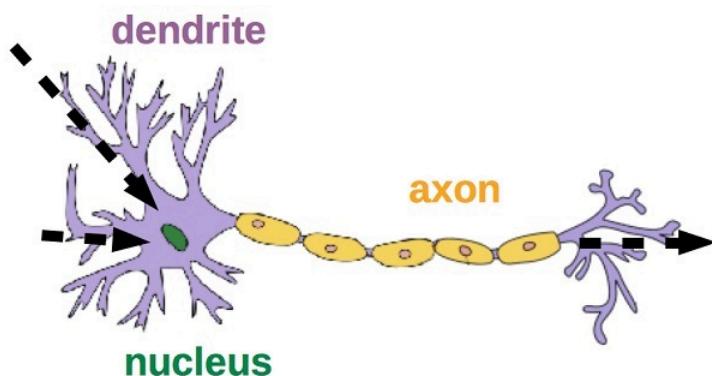


Figure 5.1 Neuron Cell

As electrical signals flow into the cell through the *dendrites* (see image above) into the nucleus, an electric charge begins to build up. When the cell reaches a certain level of charge, it *fires* sending an electrical signal out through the *axon*. However, the dendrites are not all created equal. The cell is more "sensitive" to signals through certain dendrites than others, so it takes less of a signal to fire the axon.

The biology that controls these relationships is most certainly beyond the scope of this

book, but the key concept to notice here is the way the cell *weights* incoming signals to when deciding when to fire. The neuron will dynamically change those weights in the decision making process over the course of its life. We are going to mimic that process.

5.1.1 Perceptron

Rosenblatt's original project was to teach a machine to recognize images. The original Perceptron was a conglomeration of photo-receptors and potentiometers, not a computer in the current sense. But implementation specifics aside, Rosenblatt's concept was to take the *features* of an image and assign a *weight*, a measure of importance, to each one.

The features of the input image were each a small sub-section of the image.

A grid of photo-receptors would be exposed to the image. Each receptor would *see* one small piece of the image. The brightness of the image that a particular photo-receptor could see would determine the strength of the signal that it would send to the associated "dendrite".

Each dendrite had an associated *weight* in the form of a potentiometer. Once enough signal came in, it would pass the signal into main body of the "nucleus" of the "cell". Once enough of those signals from all the potentiometers passed a certain threshold the Perceptron would *fire* down its axon, indicating a positive match on the image it was presented with. If it didn't fire for a given image, that was a negative classification match. Think "hot dog, not hot dog."

5.1.2 A Numerical Perceptron

So far there has been a lot of hand waving about biology and electric current and photo-receptors, let's pause for a second and peel out the most important parts of this concept.

Basically, we would like to take an example from a dataset, show it to an algorithm, and have the algorithm say yes or no. That is all we are doing so far. The first piece we need is determine the *features* of the sample. Choosing appropriate features turns out to be a surprisingly challenging part of machine learning. Examples of feature selection could be: home square footage, last sold price, zip code as features for determining future sale price. Or perhaps you would like to predict the species of a certain flower based solely on measurements of petal length, petal width, and sepal length. The latter three measurements would be the features.

In Rosenblatt's experiment, the features were the subsections of the larger image, one

section per receptor.

We then need a set of *weights* to assign to each of the features. Don't worry yet about where these weights come from. Just think of them as a percentage of the signal to let through into the neuron.

TIP

Generally, you will see the individual features denoted as x_i where i is a reference integer. And the collection of all features for a given example will be denoted as X representing a vector.

$$X = [x_1, x_2, \dots, x_i, \dots, x_n]$$

And similarly, you will see the associate weights for each feature as w_i where i corresponds to the integer in x . And the weights are generally represented as a vector W

$$W = [w_1, w_2, \dots, w_i, \dots, w_n]$$

With the features in hand we just multiply each feature (x_i) by the corresponding weight (w_i) and then sum up.

$$(x_1 * w_1) + (x_2 * w_2) + \dots + (x_i * w_i) + \dots$$

The one piece we are missing here is the neuron's threshold to fire or not. And it is just that, a threshold. Once the weighted sum is above a certain threshold the perceptron outputs 1. Otherwise it outputs 0.

We can represent this threshold with a simple *step function* (labeled "Activation Function" in the image)

Basic Perceptron

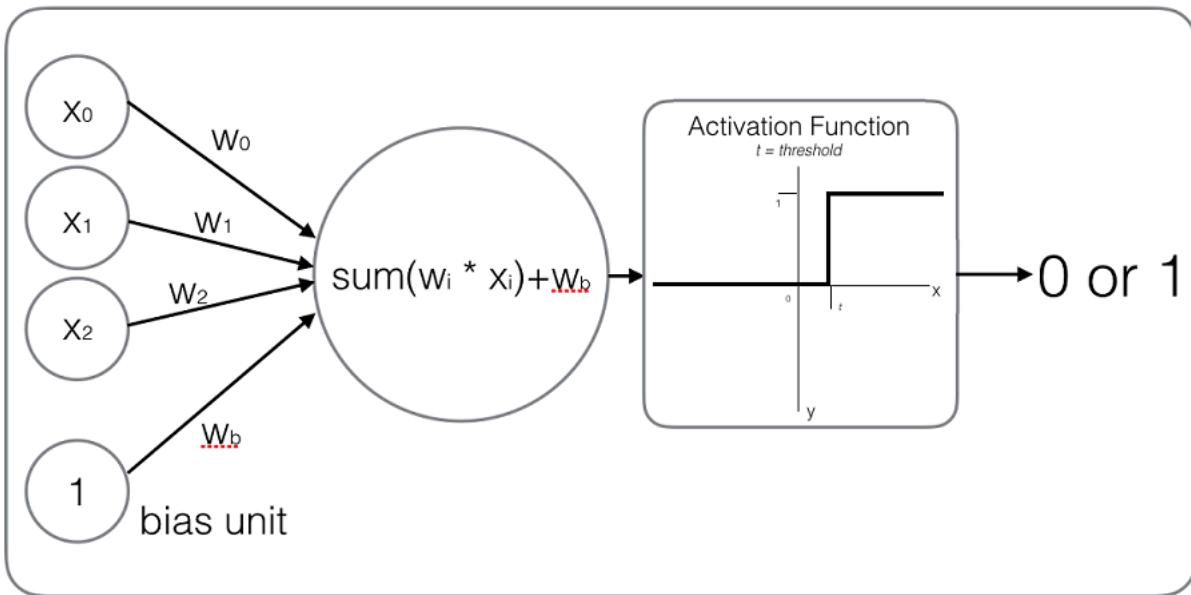


Figure 5.2 Basic Perceptron

5.1.3 Detour through Bias

In the image above and this example there is reference to *bias*, what is this? The bias is an "always on" input to the neuron. The neuron has a weight dedicated to it just as with every other element of the input, and that weight is trained along with the others in the exact same way. This is represented in two ways in the various literature around neural networks. You may see the input represented as the base input vector, say of n -elements, with a 1 appended to the beginning or the end of the vector, giving you an $n+1$ dimensional vector. The position of the one is irrelevant to the network, as long as it is consistent across all of your samples. Other times people presume the existence of the bias term and leave it off the input in a diagram, but the weight associated with it exists separately and is always multiplied by one and added to the dot product of the sample input's values and their associated weights. Both are effectively the same, just a heads up to notice the two common ways of displaying the concept.

The reason for having the bias weight at all, is we need the neuron to be resilient to inputs of all zeros. It may be the case that the network needs to learn to output 0 in the face of inputs of 0, but it may not. Without the bias term the neuron would output $0 * \text{weight} = 0$ for any weights we started with or tried to learn. With the bias term, we would not have the problem. And in case the neuron needs to learn to output 0, in that case, the neuron can learn to decrement the weight associated with the bias term enough to keep the dot product below the threshold.

A rather neat alignment with where we started.

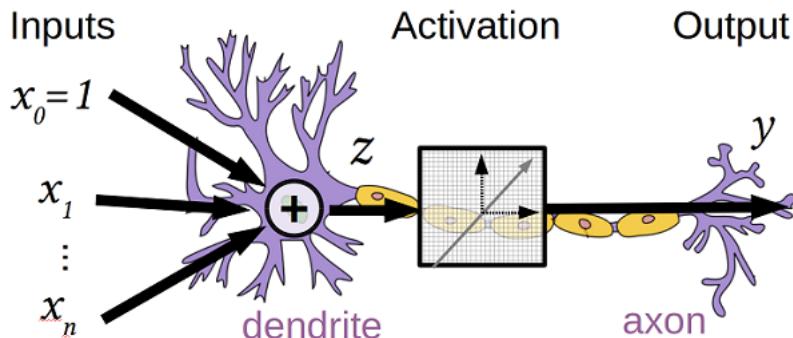


Figure 5.3 Perceptron and Neuron

And in mathematical terms, the output of our Perceptron, denoted $f(x)$ would look like Equation 1:

$$f(\vec{x}) = 1 \text{ if } \sum_{i=0}^n x_i w_i > \text{threshold} \text{ else } 0$$

Figure 5.4 Threshold Activation Function

TIP

The sum of the pairwise multiplications of the input vector (X) and the weight vector (W) is exactly the dot product of the 2 vectors. This is the most basic element of why *Linear Algebra* will factor so heavily in the development of Neural Networks. The other side effect of this is *Graphics Processing Units* in modern computers turn out to be very useful in developing these tools given their hyper-optimization for Linear Algebra operations.

We aren't *learning* anything just yet. But we have achieved something quite important. We've passed data into a model and received an output. That output is likely wrong, given we said nothing about where the values of the weights come from. But this is where things will get interesting.

TIP

The base unit of any Neural Network is the neuron. And the basic Perceptron is a special case of the more generalized *neuron*. We will refer to the Perceptron as a neuron for now, and come back to the terminology when it no longer applies.

5.1.4 A Pythonic Neuron

Calculating the output of the neuron we described above is very straightforward in Python. We can also use the numpy *dot* function to multiply our two vectors together.

```

import numpy as np
example_input = [1, .2, .1, .05, .2]
example_weights = [.2, .12, .4, .6, .90]

input_vector = np.array(example_input) # Convert to numpy array so we can do pairwise
math on the vectors easily
weights = np.array(example_weights)
bias_weight = .2

activation_level = np.dot(input_vector, weights) + (bias_weight * 1)
print(activation_level)

```

0.674

With that, if we use a simple threshold activation function and choose a threshold of .5, our next step would be:

```

threshold = 0.5
if activation_level >= threshold:
    perceptron_output = 1
else:
    perceptron_output = 0

print(perceptron_output)

```

1

Given the example_input, and that particular set of weights, this Perceptron will output 1. But if we have several example_input vectors and the associated expected outcomes with each (a labeled dataset) we can decide if the Perceptron is correct or not for each *guess* based on each input.

5.1.5 Class is in Session

So far we have set up a path toward making predictions based on data, this has been setting the stage for the main act, machine learning. The values of the weights up to this point have been brushed off as arbitrary values so far. In reality they are the key to the whole structure, and we need a way to "nudge" the weights up and down based on the result of the prediction for a given example.

The Perceptron *learns* by altering the weights up or down as a function of how wrong the system's guess was for a given input. But from where does it start? The weights of an untrained neuron start out random! Random values, near zero, usually but not always, chosen from a Normal Distribution. In the example above, you can see why starting the

weights (including the bias weight) at zero would lead only to an output of zero. But establishing slight variations, without giving any track through the neuron too much power, we have a foothold from where to be right and where to be wrong.

And from there we can start to learn. Many different samples are shown to the system and each time the weights are readjusted a small amount based on whether the neuron output was what we wanted or not. With enough examples (and under the right conditions) the error *should* tend toward zero, and the system *learns*.

The trick is, and this is the key to the whole concept, that each weight is adjusted by how much it contributed to the resulting error. A larger weight (which basically lets that data point effect the result more) should be blamed more for the rightness/wrongness of the Perceptron's output for that given input.

Let's assume our example_input from above should have resulted in a 0 instead.

```
expected_output = 0

# new weight = old weight + (expected output - Perceptron output) * input to that weight
# for example in the first index above new_weight = .2 + (0 - 1) * 1 = -0.8

new_weights = []
for i, x in enumerate(example_input):
    new_weights.append(weights[i] + (expected_output - perceptron_output) * x)
weights = np.array(new_weights)

print('Original Weights: ', example_weights)
print('New Weights: ', weights)
```

```
Original Weights: [0.2, 0.12, 0.4, 0.6, 0.9]
New Weights: [-0.8 -0.08 0.3 0.55 0.7]
```

This process of exposing the network over and over to the same training set can, under the right circumstances, lead to an accurate predictor even on input that the Perceptron has never seen.

5.1.6 Logic is a Fun Thing to Learn

So the example above was just some arbitrary numbers to show how the math goes together. Let's apply this to an actual problem. A trivial, toy problem but it will demonstrate the basics of how we can teach a computer a concept, by only showing it labeled examples.

Let's try to get the computer to understand the concept of Logical OR. If either one side or the other of a concept is True (or both sides are) then the logical OR statement is true.

Simple enough. For this toy problem we can easily model every possible example by hand (this is never the case in reality). Each sample consists of the sides, each of which is either True (1) or False (0).

```
# Learn to recognize the boolean OR condition.
sample_data = [[0, 0], # False, False
               [0, 1], # False, True
               [1, 0], # True, False
               [1, 1]] # True, True

expected_results = [0,
                     1,
                     1,
                     1]

activation_threshold = 0.5
```

We need a couple of tools to get started. Numpy just to get used to doing vector (array) multiplication and random to initialize the weights.

```
from random import random
import numpy as np

weights = np.random.random(2)/1000 # Small random float 0 < w < .001
print(weights)
```

[5.62332144e-04 7.69468028e-05]

We need a bias as well.

```
bias_weight = np.random.random()/1000
print(bias_weight)
```

0.0009984699077277136

Then we can pass it through our pipeline and get a prediction for each of our four samples.

```
for idx, sample in enumerate(sample_data):
    input_vector = np.array(sample)
    activation_level = np.dot(input_vector, weights) + (bias_weight * 1)
    if activation_level > activation_threshold:
        perceptron_output = 1
    else:
        perceptron_output = 0
    print('Predicted {}'.format(perceptron_output))
    print('Expected: {}'.format(expected_results[idx]))
    print('=====')
```

```

Predicted 0
Expected: 0
=====
Predicted 0
Expected: 1
=====
Predicted 0
Expected: 1
=====
Predicted 0
Expected: 1
=====
```

So, our random weight values did not help our little neuron out that much. One right and three wrong. Let's send it back to school. Instead of just printing 1 or 0, we will actually update the weights at each iteration.

```

for iteration_num in range(5):
    correct_answers = 0
    for idx, sample in enumerate(sample_data):
        input_vector = np.array(sample)
        weights = np.array(weights)
        activation_level = np.dot(input_vector, weights) + (bias_weight * 1)
        if activation_level > activation_threshold:
            perceptron_output = 1
        else:
            perceptron_output = 0

        if perceptron_output == expected_results[idx]:
            correct_answers += 1
        new_weights = []

        for i, x in enumerate(sample):
            new_weights.append(weights[i] + (expected_results[idx] - perceptron_output) * x) 1

2
        bias_weight = bias_weight + ((expected_results[idx] - perceptron_output) * 1)
        weights = np.array(new_weights)

    print('{} correct answers out of 4, for iteration {}'.format(correct_answers, iteration_num))
```

- ➊ This is where the magic happens. There are more efficient ways doing this, but we broke it out into a loop to reinforce that each weight is updated by force of its input (x_i). If an input was small or zero, the effect on that weight would be minimal, regardless of the magnitude of the error. And conversely, the effect would be large if the input was large in that case.
- ➋ The bias weight is updated as well just like those associated with the inputs.

```

3 correct answers out of 4, for iteration 0
2 correct answers out of 4, for iteration 1
3 correct answers out of 4, for iteration 2
4 correct answers out of 4, for iteration 3
4 correct answers out of 4, for iteration 4
```

Haha! What a good student our little Perceptron is. By updating the weights in the inner

loop, by the first iteration through the data, the Perceptron has already begun to learn and it figures out something about the one more correct than it did without any updates.

In the second and third it over-corrects the weights in various directions and has to learn to go the other direction.

By the time the fourth iteration completes it has learned the relationships perfectly. The subsequent iterations do nothing to update the network as there is an error of 0 at each sample so no weight adjustments are made.

This what is known as *convergence*. A model is said to converge when its error function settles at a minimum. It is often the case in neural networks that the network bounces around looking for optimal weights to satisfy a group of data and cannot converge.

5.1.7 Next Step

As it turns out there needs to be a little more to it than that, as the basic Perceptron has the inherent flaw¹⁰⁵ that if the data is not linearly separable the model will not converge and therefore provide no useful predictive ability.

Footnote 105 [en.wikipedia.org/wiki/Perceptrons_\(book\)#The_XOR_affair](https://en.wikipedia.org/wiki/Perceptrons_(book)#The_XOR_affair)

The early experiments were successful at *learning* to classify images based solely on example images and their classes. The initial excitement of the concept was quickly tempered by the work of Minsky and Papert¹⁰⁶ who showed the Perceptron was severely limited in the kinds of classifications it make. Minsky and Papert showed that if the data samples weren't linearly separable into discrete groups the Perceptron would not be able to *learn* to classify the input data.

Footnote 106 Perceptrons by Minsky and Papert, 1969

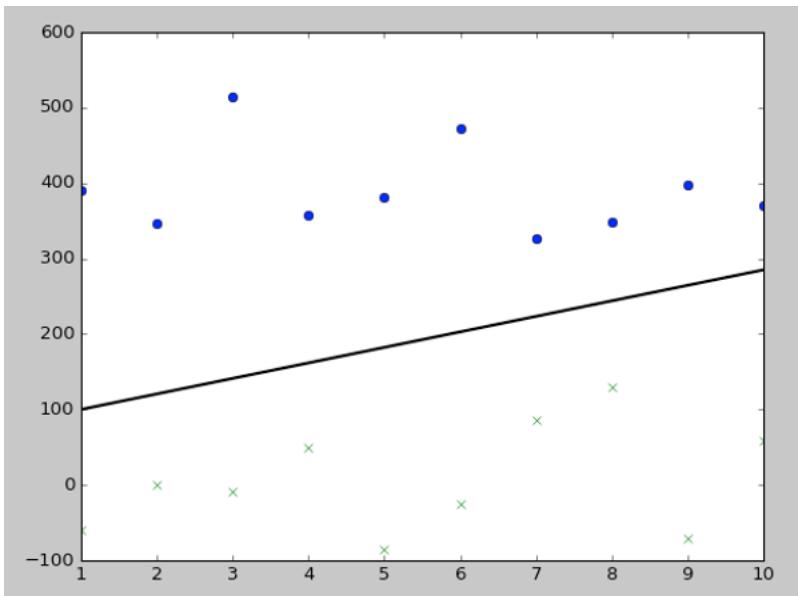


Figure 5.5 Linearly Separable Data

Linear Separable data points, no problem for a Perceptron

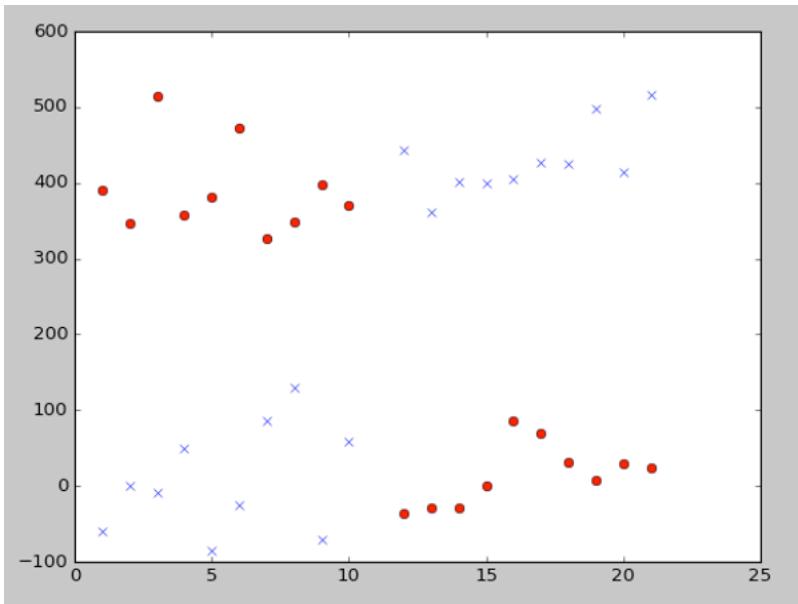


Figure 5.6 Non-Linearly Separable Data

Crossed up data and a single-neuron Perceptron will forever spin its wheels without learning to predict anything better than a random guess, a random flip of a coin. It's not possible to draw a single line between our red and blue classes shown in the diagram above.

A Perceptron can then be said to be solving to find a *linear equation* that describes the dataset.

TIP

If a Perceptron converges to a minimum, it can be said to have found a descriptive linear equation. It does not, however, say anything about which one. If there are multiple solutions to the given dataset, it will settle on one (determined by where its weights started). And, as noted in the paragraphs above it cannot describe a non-linear equation at all.

As most data in the world is not cleanly separable with lines and planes, Minsky and Papert's "proof" relegated the Perceptron to the storage shelves. But the Perceptron idea didn't die easily. It resurfaced again when Geoffrey Hinton and team¹⁰⁷ showed you could use it to solve the XOR problem by using multiple in Perceptrons in concert. The key breakthrough was a way to divide the error appropriately to each of the Perceptrons. The way they did this was resurfacing an earlier concept called *backpropagation*. With this idea for *backpropagation* across neurons and then layers of neurons, the first modern Neural Network was born.

Footnote 107 Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323, 533–536

Even though they could solve complex (nonlinear) problems, Neural Networks were, for a time, too computationally expensive. They proved impractical for common use, and they found their way back to the dusty shelves of academia, and supercomputer experimentation. But eventually computing power and the proliferation of raw data caught up. Computationally expensive algorithms were no longer show-stoppers. And thus the third age of Neural Networks began.

But back to what they found.

5.1.8 Emergence from the From the First AI Winter

As with most great ideas, the good ones will bubble back to the surface eventually. It turns out that the basic idea behind the Perceptron can be extended to overcome the basic limitation, that doomed it at first. If you gather multiple Perceptrons together and feed the input into one (or several) and then feed the output of those Perceptrons into more Perceptrons before finally comparing the output to the expected value the system (a Neural Network) can learn more complex patterns and overcome the limitations of the XOR problem. The key question is: do we know how to update the weights in the earlier layers appropriately?

Let's pause for a moment and formalize a very important part of the process. So far we

have talked about errors and how much the prediction was off base from a Perceptron. All of this refers to a *cost function* or *loss function*. The concept of a cost function is as we have seen the expected value of the network (y) given an input (x), as in Equation 2.

$$J(x) = y - f(x)$$

Figure 5.7 Cost Function

Our goal in training a Perceptron, or a Neural Network in general is simply to minimize this cost function across all available input samples, see Equation 3.

$$\min \sum_{i=1}^n J(x_i)$$

Figure 5.8 Minimized Cost Function

There are other cost functions, such as *Mean Squared Error* as we will see in a bit, but the choice between them is beyond the scope of this book. Be aware of them, but if you grasp the idea that we are minimizing a cost function across a dataset is our ultimate goal, the rest of the concepts presented here will make sense.

5.1.9 Backpropagation

So Hinton, et.al. decided there was a way to use multiple Perceptrons at the same time with one target. This they showed could solve problems that were not linearly separable. The functions they could now approximate non-linear functions as well as linear ones.

But how in the world do we update the weights of these various Perceptrons? What does it even mean to have contributed to an error? Say two Perceptrons set next to each other and each receive the same input. No matter what we do with output (concatenate it, add it, multiply it) when we try to push the error back to the initial weights it will be a function of the input (which was identical on both sides), so they would be updated the same amount at each step and we would never go anywhere.

The concept gets even more mind bending when we imagine a Perceptron that feeds into a second Perceptron as the second's input. Which is exactly what we are going to do.

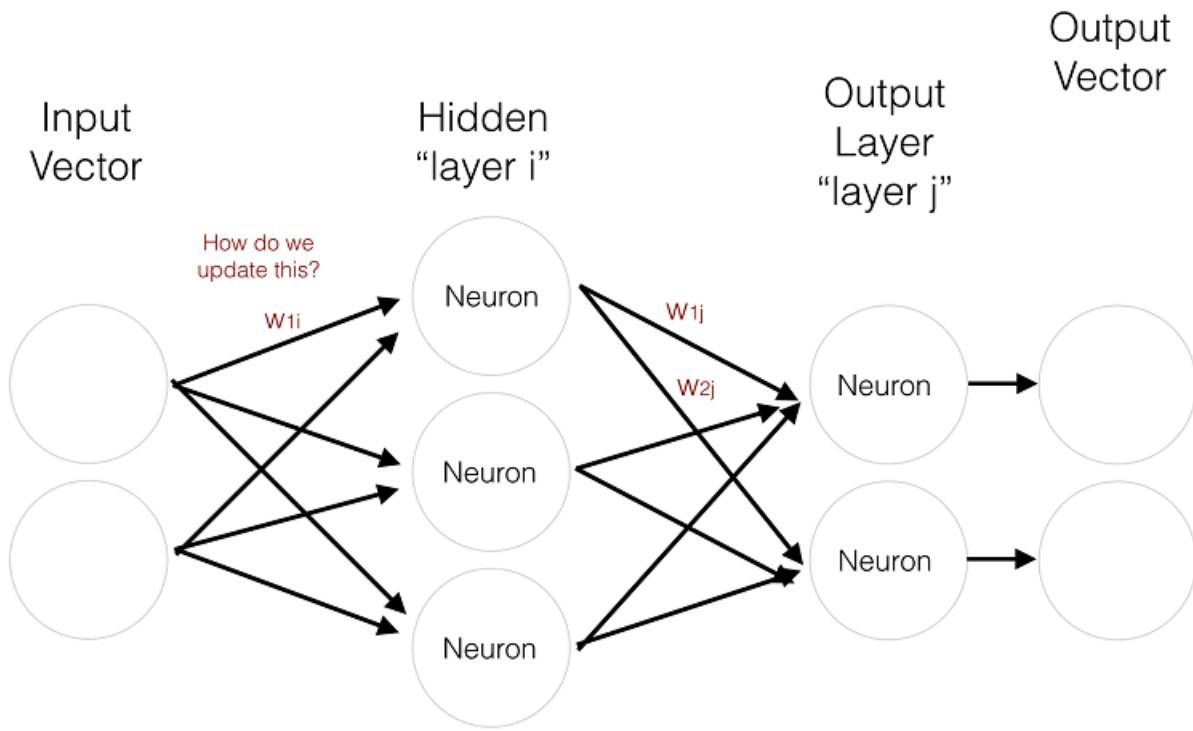


Figure 5.9 Neural Net with Hidden Weights

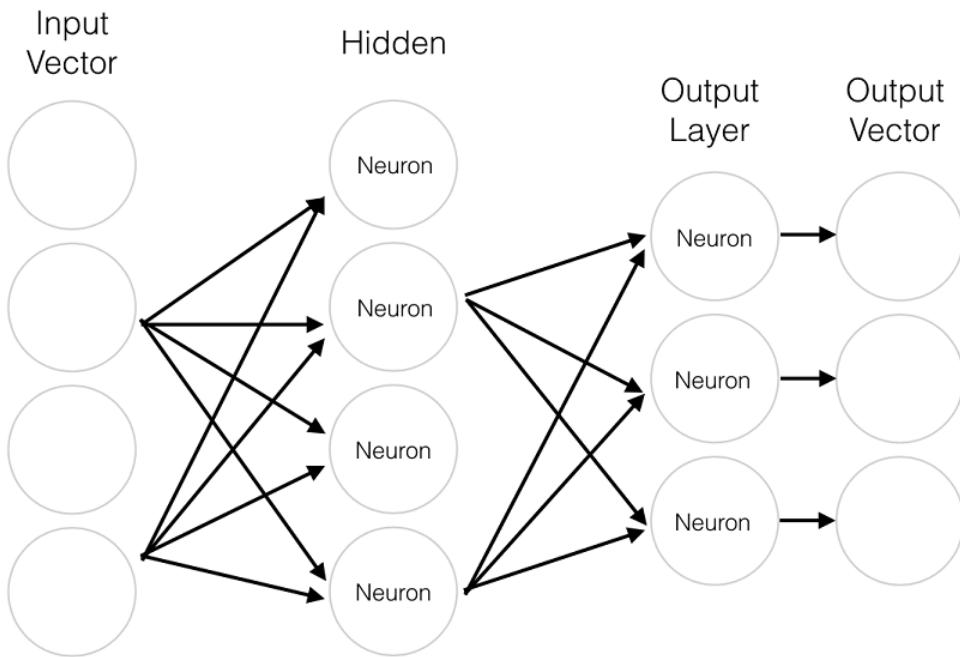
Backpropagation helps us solve this, but we have to tweak our Perceptron a little to get there.

Remember the weights were being updated as a factor of how much they contributed to the overall error. But if a weight is affecting an output that becomes the input for another Perceptron, we no longer have a clear idea of what the error is at the beginning of that second Perceptron.

We need a way to calculate the amount a particular weight (w_{1i} in the image above) contributed to the error given that it contributed to the error via other weights (w_{1j} and w_{2j}) in the next layer. And the way to do that is with *backpropagation*.

Now is a good time to stop using the term Perceptron because we are going to change how the weights in each neuron are updated. From here on out we will refer to the more general *neuron* that includes the Perceptron, but also its more powerful relatives. You will also see neurons referred to as cells or nodes in the literature, and in most cases the terms are interchangeable.

A Neural Network, regardless of flavor, is nothing more than a collection of neurons composed into layers. Once we have an architecture where the output of a neuron becomes the input of another neuron, we begin to talk about *hidden layers* vs an *input* or *output* layer.



In a Fully-Connected Network, all nodes feed into each node in the next layer.
Just some of the connections are drawn here.

Figure 5.10 Fully Connected Neural Net

You will see this described a *fully connected* network. Though not all the connections are drawn in the illustration above, in a fully-connected network each input element has a connection to and associated weight in *every* neuron in the layer. So in a network that takes an 4 dimensional vector as input and has 5 neurons, there will be 20 total weights in the layer (4 weights in each of the 5 neurons).

As with the input to the perceptron, where there was a weight for each input, the neurons in the second layer of a neural network have a weight assigned not to the original input, but to each of the outputs from the first layer. So now you can see the difficulty in calculating the amount a given weight in the first layer contributed to the overall error, as it is passed through not one other weight but through one weight in each of the neurons of the next layer. The derivation and mathematical details of algorithm itself, while extremely interesting, are beyond the scope of this book, but we take a brief moment for an overview so the black box of neural nets aren't left completely in the dark.

Backpropagation, short for backpropagation of the errors, describes how we can discover the appropriate amount to update a specific weight, given the input, the output, and expected value. *Propagation* or forward propagation is an input flowing "forward" through the net and computing the output for the network for that input. To get to backpropagation we need first need to pivot from our Perceptron's activation function to something that is slightly more complex.

Up until now, we have been using a step function as our artificial neuron's *activation*. But as we'll see in a moment, backpropagation requires an activation function that is non-linear and easily differentiable. Now each neuron will output a value *between* two values, like 0 and 1 in the commonly used sigmoid function, Equation 4:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Figure 5.11 Sigmoid Function

Where x is the dot product of the input vector and the weight vector. There are many other activation functions such as *Hyperbolic Tangent* and *Rectified Linear Units*, they all have benefits and downsides. Each shines in different ways in varying types of neural network architectures as we'll see in later chapters.

So why differentiable? If we can calculate the derivative of the function, we can also do partial derivatives of the function, with respect to various variables in the function itself. There is the hint of the magic. 'With respect to various variables.' We have a path toward updating a weight with respect to the amount of input it received!

5.1.10 Derivative All the Things

We start with the error of the network and apply a cost function, say *squared error*, seen as Equation 5.

$$MSE = (y - f(x))^2$$

Figure 5.12 Squared Error Cost Function

We can then lean on the *chain rule* of calculus to calculate the derivative of compositions of functions, as in Equation 6. And the network itself is nothing but a composition of functions (specifically dot products followed by our new non-linear activation function at each step).

$$(f \circ g)' = (f' \circ g) \cdot g'$$

Figure 5.13 Chain Rule

Or more traditionally:

$$(f \circ g)' = F'(x) = f'(g(x))g'(x)$$

Figure 5.14 Also the Chain Rule

We can now use this to find the derivative of the activation function of each neuron with respect to the input that fed it, we can calculate how much that weight contributed to final error and adjust it appropriately.

If the layer is the output layer the update of the weights is rather straightforward, with the help of our easily differentiable activation function. The derivative of the Error with respect to the j-th output that fed it is:

$$\Delta w_{ij} = -\alpha \frac{\partial \text{Error}}{\partial w_{ij}} = -\alpha o_i(o_j - f(x)_j)o_j(1 - o_j)$$

Figure 5.15 Output Layer Derivative

If we are updating the weights of a hidden layer, things are a little more complex, represented in Equation 9:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = -\alpha o_i \left(\sum_{l \in L} \delta_l w_{jl} \right) o_j (1 - o_j)$$

Figure 5.16 Inner Layer Derivative

$f(x)$ in the first equation is the output. Specifically the j-th position of the output vector. The o in these equations is the output of a node in either the i-th layer or the j-th layer, where the output of the i-th layer is the input of the j-th layer. So we have the alpha, the learning rate, times the output of the "earlier" layer times the derivative of the activation function from the "latter" layer *with respect to* the weight that fed the output of the i-th layer into the j-th layer. The sum in the latter equation expresses this for all all inputs in the training set.

It is important to be specific about when the changes are applied to the weights themselves. As you calculate each weight update in each layer, the calculations all depend on the state of the network during the forward pass. So, once the error is calculated, you then calculate the proposed change to each weight in the network. But do NOT apply any of them. At least until you get all the way back to the beginning of the network. Otherwise as you update weights toward the end of the net the derivatives calculated for the lower levels would no longer be the appropriate gradient for that particular input. You can aggregate all of the ups and down for each weight based on each training sample, without updating any of the weights and instead update them at the end of all the training, but we'll discuss more on that choice in a few pages when we discuss *batching*.

And then to train the network, pass all the inputs in. Get the associated error for each input. Backpropagate those error to each of the weights. And then update each weight with total change in error. Once all of the training data has gone through the network once, and the errors backpropagated, we call this an *epoch* of the neural network training cycle. The dataset can then be passed in again and again to further refine the weights. Be careful though or the weights will overfit the training set and no longer be able to make meaningful predictions on novel data points from outside the training set.

Alpha in the equations above is the *learning rate*, a constant used to speed up or slow down the updates of the weights in general. Too large and you could easily over-correct and the next error, presumably larger, would itself lead to a large weight correction the other way, but even further from the goal. Set alpha too small and the model will take too long to converge to be practical or worse, get stuck in a local minimum on the *error surface*.

5.1.11 Let's Go Skiing - The Error Surface

The goal of training in neural networks as we stated earlier is to minimize a cost function by finding the best parameters (weights). Keep in mind, this is not the error for any one particular data point. We want to minimize the cost for all the various errors taken together.

Creating a visualization of this side of the problem can help build a mental model what we are doing when we adjust the weights of the network as we go.

From earlier, *squared error* is a common cost function like Equation 10:

$$MSE = (y - f(x))^2$$

Figure 5.17 Squared Error Cost Function

If you imagine plotting the error as function of the possible weights, given a specific input and a specific expected output, there is some point that that function is closest to 0 and that is our *minimum*, the spot our model has the least error.

This minimum will be the set of weights that gives the optimal output for a given training example. You will often see this represented as a 3 dimensional bowl with 2 of the axes being a 2 dimensional weight vector and the 3rd being the error. A vast simplification of course, but the concept is the same in higher dimensional spaces (for cases with more than 2 weights).

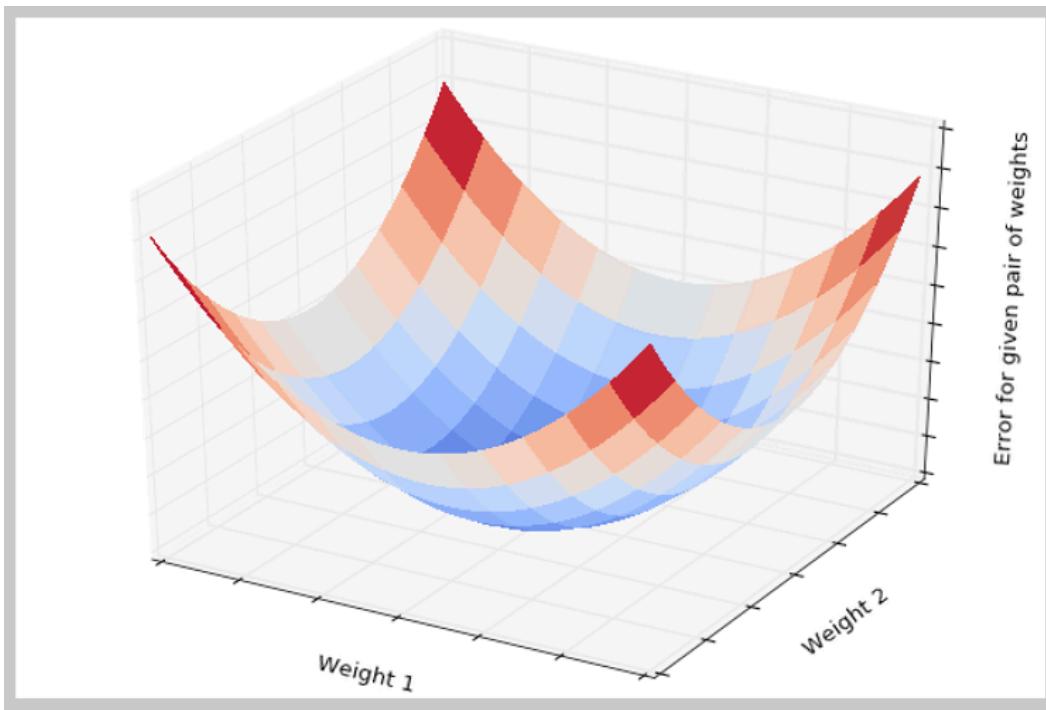


Figure 5.18 Convex Error Curve

We can similarly graph the error surface as a function of all possible weights across all the inputs of a training set. But we need to tweek the error function a little. We need something that represents the aggregate error across all inputs for a given set of weights. For this example we will use *mean squared error* as the z axis.

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

Figure 5.19 Mean Squared Error

Here again, we will get a error surface with a minimum that is located at the set of weights. Those set of weights will represent a model with that best fits the entire training set.

5.1.12 Off the Chair Lift, Onto the Slope

So what does this visualization represent? At each epoch, the algorithm is performing *gradient descent* in trying to minimize the error. So each time we adjust the weights in the direction that will in theory reduce our error the next time. In the case of a convex error surface, this will be great. Standing on the ski slope, look around, find out which way is down and go that way!

But we are not always so lucky as to have such a smooth shaped bowl, there may be many pits and divots scattered about. This is what is know as a *non-convex error curve*.

Again the diagrams are representing weights for 2 dimensional input. But the concept is the same if we have a 10 dimensional input, or 50, or 1000. In those higher dimensional spaces it just doesn't make sense to visualize it anymore, so we trust the math. Once you start using Neural Networks, actually visualizing the error surface becomes less important. You get the same information from watching (or plotting) the error or a related metric over the training time and seeing if it is tending toward 0. That is how you tell if your network is on the right track or not. But these 3D representations are a helpful tool for creating a mental model of the process.

But what about the non-convex error space? Aren't those divots and pits a problem? Yes, yes they are. Depending on where we randomly start our weights, we could end up at radically different weights and the training would stop, as there is no other way to go down from this *local minimum*.

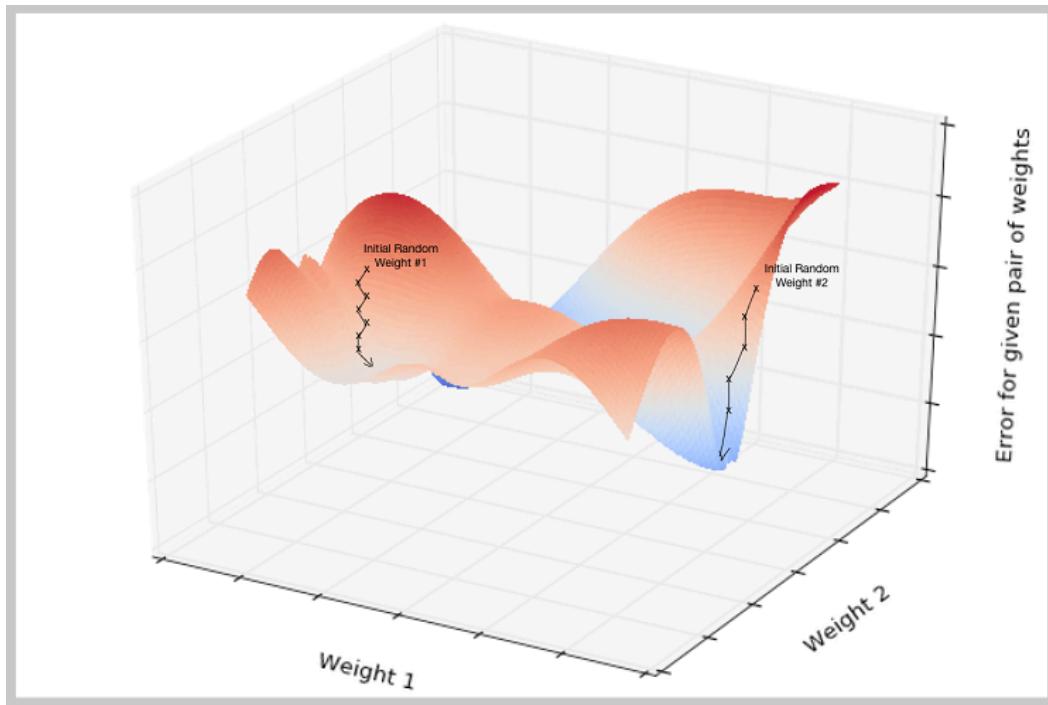


Figure 5.20 Non-Convex Error Curve

And as we get into even higher-dimensional space the local minima will follow us there as well.

5.1.13 Let's Shake Things Up a Bit

Up until now, we have been aggregating the error for all the training examples and skiing down the slope as best we could. This training approach, as described, is *batch* learning, as in the whole batch. But batch learning has a static error surface for the entire training set. With this single static surface it is quite possible, that only heading down hill from a random starting point we would end up in some local minima and not know there are better options for our weight values. There are two other options to training which can help us skirt this issue.

First is *stochastic* gradient descent. In *stochastic* gradient descent we update the weights after each training example, rather than after looking at all of the training examples. By doing this, the error surface is redrawn for each example, as each different input could have a different expected answer. So the error surface for most examples will look different. But we are still just adjusting the weights based on gradient descent, *for that example*. So, instead of gathering up the errors and then adjusting the weights once at the end of the epoch, we update the weights after every individual example. The key point is that we are moving *toward* the presumed minimum. Not all the way to that presumed minimum at any given step.

And as we move toward the various minima on this fluctuating surface, with the right data and right hyperparameters, we can more easily bubble toward the global minimum. If your model is not tuned properly or the training data is inconsistent the model won't converge and you will just spin and turn over and over and the model never learns anything. But in practice stochastic gradient descent proves quite effective in avoiding local minima in most cases. The downfall of this approach is it is slow. Calculating the forward pass, backpropagation, and updating the weights after each example adds that much time to an already slow process.

There is finally the more-common approach, our second training option, *mini-batch*. In mini-batch training, a subset of the training set is passed in and their associated errors are aggregated as in full *batch*. Those errors are then backpropagated as with *batch* and the weights updated for each subset of the training set. Then this is repeated with the next batch and so on until the training set is exhausted. And that again would constitute one epoch. This is a very happy medium, that gives us the benefits of both *batch* (speedy) and *stochastic* (resilient) training methods.

While the details of how *backpropagation* works are fascinating ¹⁰⁸, they aren't necessarily non-trivial and as noted earlier, outside the scope of this book. But a good

mental image to keep handy is that of the error surface. In the end a neural network is just a way to *as fast as possible* walk down the slope of the bowl until you are the bottom. From a given point look around you in every direction, find the steepest way down (not a pleasant image if you are scared of heights) and go that way. At the next step (batch, mini-batch, or stochastic) look around again, find the steepest way and now go that way. Soon enough you'll be by the fire in ski-lodge at the bottom of the valley.

Footnote 108 en.wikipedia.org/wiki/Backpropagation

5.1.14 Keras: Neural Networks in Python

While writing a neural network in raw Python is a fun experiment and can be very helpful in putting all these pieces together, Python is at a disadvantage when it comes to speed, and the shear number of calculations we are dealing with can make even moderately sized networks intractable. There are many libraries in Python though that get us around the speed zone. PyTorch, Theano, Tensorflow, Lasagne and many more. In the examples in this book we will be using Keras (keras.io/).

Keras is a high level wrapper with a very accessible API for Python. The exposed API can be used with 3 different backends almost interchangeably: Theano, Tensorflow from Google, and CNTK from Microsoft. All of these have their own low-level implementations of the basic elements of neural networks and have highly tuned linear algebra libraries to handle the dot products in what turns out to be matrix multiplication as efficiently as possible.

Let's look at the simple XOR problem and see if we can train a network using Keras.

```
import numpy as np

# The base Keras model class

from keras.models import Sequential

# The basic layer of the network
# Dense is a fully-connected set of neurons

from keras.layers import Dense, Activation

# Get stochastic gradient descent, though there are others
from keras.optimizers import SGD

# Our examples of exclusive OR.
# x_train is sample data
# y_train the expected outcome for example
x_train = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])
y_train = np.array([0,
                   1,
```

```
[1],  
[0]))  
  
model = Sequential()  
  
# Add a fully connected hidden layer with 10 neurons  
# The input shape is the shape of an individual sample vector  
# This is only necessary in the first layer, any additional  
# layers will calculate the shape automatically by the definition  
# of the model up to that point  
  
num_neurons = 10  
model.add(Dense(num_neurons, input_dim=2))  
model.add(Activation('tanh'))  
  
# The output layer one neuron to output 0 or 1  
model.add(Dense(1))  
model.add(Activation('sigmoid'))  
print(model.summary())
```

| Layer (type) | Output Shape | Param # |
|---------------------------|--------------|---------|
| dense_18 (Dense) | (None, 10) | 30 |
| activation_6 (Activation) | (None, 10) | 0 |
| dense_19 (Dense) | (None, 1) | 11 |
| activation_7 (Activation) | (None, 1) | 0 |

Total params: 41.0
Trainable params: 41.0
Non-trainable params: 0.0

The summary gives us an overview and we can see then number of weights at each stage, referred to as parameters in the summary. Some quick math, 10 neurons each with 2 weights (1 for each value in the input vector) and 1 weight for the bias gives us 30 weights to learn. The output layer has a weight for each of the 10 neurons in the first layer and one 1 bias weight for a total of 11 in that layer.

The next bit of code is a bit opaque:

```
sgd = SGD(lr=0.1)  
model.compile(loss='binary_crossentropy', optimizer=sgd, metrics=['accuracy'])
```

SGD is the stochastic gradient descent optimizer we imported. This is just how the model will try to minimize the error or *loss*. *lr* is the learning rate, the fraction applied to the derivative of the error with respect to each weight. Higher will speed learning, but may force the model away from the global minimum by shooting past the goal, smaller will be more precise but increase the training time and leave the model more vulnerable to local minima. The loss function itself is also defined as parameter, here `binary_crossentropy`.

And finally the metrics parameter is a list of options for the output stream during training. And compile builds, but does not yet train the model. The weights are initialized and you can use this random state to try and predict from your dataset but you will just get random guesses out.

```
print(model.predict(x_train))
```

The predict method gives the raw output of the last layer, which would be generated by the sigmoid function in this example.

```
[[ 0.5
   [ 0.43494844]
   [ 0.50295198]
   [ 0.42517585]]
```

Not much to write home about. But remember this has no knowledge of the answers just yet, it is just applying its random weights to the inputs. So let's try and train this.

```
# Here is where we train the model
model.fit(x_train, y_train,
           epochs=100)
```

```
Epoch 1/100
4/4 [=====] - 0s - loss: 0.6917 - acc: 0.7500
Epoch 2/100
4/4 [=====] - 0s - loss: 0.6911 - acc: 0.5000
Epoch 3/100
4/4 [=====] - 0s - loss: 0.6906 - acc: 0.5000
...
Epoch 100/100
4/4 [=====] - 0s - loss: 0.6661 - acc: 1.0000
```

TIP

It is possible the network won't converge on the first try. It is always possible that the first compile ends up with base parameters from the random distribution that make finding the global minimum difficult or impossible. If you run into this, you can simply call `model.fit` again with the same parameters (or add even more epochs) and see if the network finds its way eventually. Or reinitialize the network with a different random starting point and try fit from there. If you try the latter, make sure that you don't have a random seed set or you will simply repeat the same experiment over and over.

As it looked at what was actually a tiny data set over and over it finally figured out what was going on. It "learned" what exclusive-or (XOR) was just from being shown

examples! That is magic of neural networks and what will guide us through the next few chapters.

```
print(model.predict_classes(x_train))
print(model.predict(x_train))

4/4 [=====] - 0s
[[0]
 [1]
 [1]
 [0]]
[[ 0.0035659 ]
 [ 0.99123639]
 [ 0.99285167]
 [ 0.00907462]]
```

We call predict again (and predict_classes) on the trained model and we get very different results. It gets 100% accuracy on our tiny set. Of course, accuracy isn't necessarily the best measure of predictive model, but for this toy example it will do.

```
import h5py
model_structure = model.to_json() ①
with open("basic_model.json", "w") as json_file:
    json_file.write(model_structure)

model.save_weights("basic_weights.h5") ②
```

- ① We export the structure of the network to a json blob for later use using Keras' helper method.
- ② The trained weights must be saved separately. The first part just saves the structure of the network. This preserves the weights straight to a file. You must re-instantiate the same model structure to re-load them later.

And there are similar methods to re-instantiate the model so you don't have to retrain every time you want to make a prediction. That will be huge going forward, for while this model takes a few seconds to run, in the coming chapters that will quickly grow to minutes, hours, even in some cases days depending on the hardware and the complexity of the model, so get ready!

5.1.15 Onward and Deepward

As neural networks have spread and spawned the entire field of *deep learning*, much research has been done and continues to be into the details of these systems: different activation functions (e.g. sigmoid, rectified linear units, hyperbolic tangent, to name a few), application of the learning rate, to dial up or down the effect of the error. Dynamically adjusting the learning rate using a *momentum* model to find the global minimum faster, application of *dropout* where a randomly chosen set of weights are ignored in a given training pass to prevent the model from becoming too attuned to its training set (over-fitting), regularization of the weights to artificially dampen a single weight from growing or shrinking too far from the rest of the weights (another tactic to avoid over-fitting). The list goes on and on.

5.1.16 Normalization: Input with Style

Neural networks want vector input and will do their best to work on whatever is fed in them but one key thing to remember is input *normalization*. This is true of many machine learning models, but imagine the case of trying to classify houses, say on their likelihood of selling in a given market, and you only have 2 data points, number of bedrooms and last selling price. This data could be represented as a vector. Say, for a 2 bedroom house that last sold for \$275,000.

```
input_vec = [2, 275000]
```

As the network tries to learn anything about this data, the weights associated with bedrooms in the first layer would need to grow huge very quickly to compete with the large values associated with price. So it is common practice to normalize the data so that it retains its elements' information relative sample to sample, but also works within similar range as the other elements within a single sample vector. There are several approaches to this and all have their pros and cons. Mean normalization, feature scaling, coefficient of variation, just to name a few. But the goal of all is to get the data in some range like [-1, 1] or [0, 1] for each element in each sample without losing information.

We won't have to worry too much about this with NLP as TF-IDF, one-hot encoding, and as we'll soon see, word2vec are all in some form normalized already, but it is important to keep in mind in other implementations of neural networks.

Finally a last bit of terminology. There is not a great deal of consensus on what constitutes a Perceptron vs. Multi-Neuron Layer vs. Deep Learning, but we've found it handy to differentiate between a Perceptron and a Neural Network if you have to use the

derivative of the activation function to properly update the weights. In this book we will use Neural Network and Deep Learning in this context and save the term Perceptron for its (very) important place in history.

5.1.17 Summary

In this chapter we learned how

- Minimizing a cost function is a path toward learning
- Backpropagation algorithm is the means by which a networks LEARNS
- The amount a weight contributes to a model's error is the directly related to the amount it needs to updated
- Neural Networks are at their heart optimization engines
- There are pitfalls to avoid during training
- But Keras has means to make all of this accessible



Reasoning with Word Vectors (Word2vec)

In this chapter

- Learning about word vectors
- Understanding the key concepts of word vectors
- Using pretrained models for your applications
- Reasoning with word vectors to solve real problems
- Visualizing word vectors
- Some surprising uses for word embeddings

One of the most exciting recent advancements in NLP was the "discovery" of word vectors. This chapter will help you understand what they are and how to use them to do some surprisingly powerful things. You will learn how to recover some of the the fuzziness, subtlety of word meaning that was lost in the approximations of the previous chapters.

In the previous chapters we ignored the nearby context of a word. We ignored the words around each word. We ignored the effect the neighbors of a word have on its meaning and how those relationships effect the overall meaning of a statement. Our bag of words concept jumbled all the words from each document together into a statistical bag. In this chapter we'll create much smaller bags of words from a "neighborhood" of only a few words, typically less than ten tokens. We'll also ensure that these neighborhoods of meaning don't spill over into adjacent sentences. This helps focus our word vector training on the words that are relevant.

The tools we used before didn't do a very good job of identifying synonyms, antonyms, or words that just belong to the same category, like people, animals, places, plants, names, or concepts. Our numerical representation of words, *n*-grams, and documents

didn't capture all of the literal meanings of a word, much less the implied or hidden meanings. Some of the connotations of a word got lost in our oversized bags of words.

Word vectors bring back all that meaning. Word vectors are numerical representations of word semantics, or meaning. They capture the "peopleness", "animaleness", "placeness", "thingness" and "conceptness" of words. And they combine all that into a compact vector of floating point values that we can do queries and logical reasoning with.

6.1 Semantic Queries and Analogies

Well what are these awesome word vectors good for? Have you tried to recall the name of someone famous but you only have a general impression of them, like maybe

She invented something to do with physics in Europe in the early 20th century.

If you enter that sentence into a Google or Bing search box you may not get the direct answer you're looking for, "Marie Curie." Google Search will most likely only give you links to lists of famous physicists, both men and women. You'd have to skim several of those pages to find the answer you're looking for. But once you found "Marie Curie", Google or Bing would keep note of that, and might get better at providing you search results in the future.¹⁰⁹

Footnote 109 At least, that's what it did for us in researching this book. We had to use private browser windows to ensure that your search results would be similar to ours.

With word vectors we can search for words or names that combine the meaning of the words "woman", "Europe", "physics", "scientist", and "famous", and that would get us close to the token "Marie Curie" that we're looking for. And all we have to do to make that happen is add up the word vectors for each of those words that we want to combine:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] + wv['scientist']
```

We'll show you the exact way to do this query in this chapter. And we'll even show you how to subtract out gender bias from the word vectors used to compute your answer:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv[physics'] + wv['scientist']  
- wv['male'] - 2 * wv['man']
```

With word vectors you can take the "man" out of "woman"!

6.1.1 Analogy Questions

What if you could rephrase your question as an analogy question? What if your "query" was something like this:

Who is to nuclear physics what Louis Pasteur is to germs?

Again Google Search, Bing, and even Duck Duck Go aren't much help with this one.¹¹⁰ But with word vectors this becomes as simple as subtracting "germs" from "Louis Pasteur" and then adding in some "physics":

Footnote 110 Try them all if you don't believe us

```
>>> answer_vector = wv['Louis_Pasteur'] - wv['germs'] + wv['physics']
```

And if you're interested in trickier analogies about people in unrelated fields, like musicians and scientists, you can do that too!

Who is the Marie Curie of music?

OR

Marie Curie is to science as who is to music?

Can you figure out what the word vector math would be for that question?

You might have seen a questions like these on the English analogy section of standardized tests like SAT, ACT, or GRE exams. Sometimes they are written in formal mathematical notation like

```
MARIE CURIE : SCIENCE :: ? : MUSIC
```

Does that make it easier to guess the word vector math? One possibility is:

```
>>> wv['Marie_Curie'] - wv['science'] + wv['music']
```

And you can answer questions like this for things other than people and occupations, like perhaps sports teams and cities:

The Timbers are to Portland as what is to Seattle?"

In standardized test form that's:

```
TIMBERS : PORTLAND :: ? : SEATTLE
```

But, more commonly, standardized tests use English vocabulary words and ask less fun questions like

```
WALK : LEGS :: ? : MOUTH
```

OR

```
ANALOGY : WORDS :: ? : NUMBERS
```

All those "tip of the tongue" questions are a piece of cake for word vectors, even though they aren't multiple choice. When you are trying to remember names or words, it's rarely possible to think of the *A*, *B*, *C*, and *D* multiple choice options. But NLP comes to the rescue with word vectors.

Word vectors can answer these vague questions and analogy problems. Word vectors are great for questions that you can't even pose in the form of a search query or analogy. We'll talk about some non-query math with word vectors in the "Vector-Oriented Reasoning" section. Word vectors can help you remember any word or name on the tip of your tongue, as long as the word vector for the answer exists in your word vector vocabulary.¹¹¹

Footnote 111 For Google's pretrained word vector model, your word is almost certainly within the 100B word news feed that Mikolov trained it on, unless your word was invented after 2013.

6.2 Word Vectors

In 2012, Thomas Mikolov found a way to encode the meaning of words in a modest number of vector dimensions.¹¹² Mikolov had the insight to train a neural net¹¹³ to predict word occurrences near each target word. He called his algorithm for creating these word vectors word2vec.

Footnote 112 word vectors typically have 100 to 500 dimensions, depending on the breadth of information in the corpus used to train them.

Footnote 113 It's only a single layer network, so almost any linear machine learning model will work as well, e.g. logistic regression or truncated SVD or Naive Bayes

Mikolov's approach enabled `word2vec` to learn the meaning of words merely by processing a large corpus of unlabeled text. No one has to tell the algorithm that "Marie Curie" is a scientist, that the "Timbers" are a soccer team, or that Portland and Seattle are cities. And no one has to tell `word2vec` that soccer is a sport, or that a team is a group of people, or that cities are both places and communities of people. *Word2vec* can learn that and much more, all on its own! All you need is a corpus large enough to mention "Marie Curie" and "Timbers" and "Portland" near other words associated with science or soccer or cities.

This unsupervised nature of Mikolov's `word2vec` is what makes it so powerful. The world is full of unlabeled, uncategorized, unstructured natural language text.

NOTE

Unsupervised vs supervised training refers to two different approaches to machine learning. In supervised training, the training data must be labeled in some way, like the spam label on our SMS messages, or the "Like" label on Tweets. A supervised model can only get better if it can measure the difference between the expected output (the label) and its predictions.

In unsupervised learning, we train the model to perform a task, but without any labels, only the raw data. Clustering algorithms like K-Means or DBSCAN are examples of unsupervised learning. We have to look for patterns in the relationships between the data points themselves.

Instead of trying to train a neural network to learn the target word meanings directly from labels for that meaning, we teach the network to predict words near the target word in our sentences. So in this sense, we do have labels, the words we are trying to predict, the neighbors. But as they are coming from the dataset itself, and require no hand-labeling this fairly falls into the unsupervised category.

And the prediction itself is not what is important here. It is merely a means to an end. What we do care about is the internal representation, the vector, that `word2vec` gradually builds up to help it generate those predictions. This representation will capture much more of the meaning of the target word (its semantics) than the word-topic vectors that came out of Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDiA) in Chapter 4.

TIP

Models that learn by trying to repredict the input using a lower dimensional internal representation are called "autoencoders." If you want to learn more about unsupervised deep learning models that create compressed representations of high dimensional objects like words, search for the term "autoencoder."¹¹⁴ They are also a common way to introduce neural nets, because they can be applied to almost any data set.

Footnote 114 ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/

And word2vec will learn about things you might not think to associate with all words. Did you know that every word has some geography, sentiment (positivity), and gender associated with it? If any word in your corpus has some quality, like "placeness", "peopleness", "conceptness" or "femaleness", then all the other words will also be given a score for these qualities in your word vectors. The meaning of a word "rubs off" on the neighboring words when word2vec learns word vectors.

All words in your corpus will be represented by numerical vectors, similar to the word-topic vectors discussed in Chapter 4. Only this time the "topics" actually mean something more specific, more precise. In LSA, words only had to occur in the same document to have their meaning "rub off" on each other and get incorporated into their word-topic vectors. And our new, more precise, word vector "topics" weights can be added and subtracted to create new word vectors that actually mean something!

A mental model that may help you understand word vectors would be think of word vectors as a list of "weights" or scores. Each weight or score is associated with a specific dimension of meaning for that word, like:

```
>>> from nlpia.book.examples.ch06_nessvectors import *      ①
>>> nessvector('Marie_Curie').round(2)                      ②
placeness      -0.46
peopleness     0.35
animalness     0.17
conceptness    -0.32
femaleness      0.26
```

- ① Only import this module if you have a lot of RAM and a lot of time. The pretrained word2vec model is huge.
- ② You can compute real "nessvectors" for any word in Mikolov's vocabulary using the examples here:
github.com/totalgood/nlpia/blob/master/src/nlpia/book/examples/ch06_nessvectors.py

Mikolov developed the Word2vec algorithm while trying to think of ways to numerically represent words. He wasn't satisfied with the clunky word sentiment math we did in Chapter 4. He wanted to do *vector-oriented reasoning* like what we just did in the previous section with those analogy questions. This concept may sound fancy, but really it just means that we can do math with word vectors that make sense when we translate the vectors back into words. We can add and subtract word vectors to *reason* about the words they represent and answer questions similar to our examples above, like¹¹⁵

Footnote 115 For those not up on sports, the Portland Timbers and Seattle Sounders are Major League Soccer teams.

```
vector('Timbers') - vector('Portland') + vector('Seattle') = ?
```

Ideally we'd like this math (word vector reasoning) to give us

```
vector('Seattle Sounders')
```

Similarly, our analogy question "'Marie Curie' is to 'physics' as __ is to 'classical music'?" can be thought about as a math expression like this

```
vector('Marie_Curie') - vector('physics') + vector('classical_music') = ?
```

In this chapter we want to improve on the LSA (Latent Semantic Analysis) word vector representations we introduced in the previous chapter. Topic vectors constructed from entire documents using LSA are great for document classification, semantic search, and clustering. But the topic-word vectors that LSA produces aren't accurate enough to be used for semantic reasoning or classification and clustering of short phrases or compound words. We'll show you how to train the single-layer neural networks required to compute these new word vectors. And we'll show how and why they have replaced LSA word-topic vectors for many applications.

6.2.1 Vector-Oriented Reasoning

In 2013, Tomas Mikolov and his team published their discoveries around applications of word vector representations¹¹⁶. The paper with the dry-sounding title "Linguistic Regularities in Continuous Space Word Representations" described a surprisingly accurate language model (a leap in accuracy an order of magnitude greater than the incremental improvements demonstrated by other researchers with much more complex models).¹¹⁷ It was so surprisingly good, in fact, that Mikolov's initial paper was rejected by the International Conference on Learning Representations.¹¹⁸ Reviewers thought that the model's performance was too good to be true. It took nearly a year for Mikolov and his team to prove otherwise by releasing the source and resubmitting their paper to the journal of the Association for Computational Linguistics.

Footnote 116 msr-waypoint.com/en-us/um/people/gzweig/Pubs/NAACL2013Regularities.pdf

Footnote 117 Radim ehek's interview of Tomas Mikolov:
rare-technologies.com/rrp#episode_1_tomas_mikolov_on_ai

Footnote 118 ICRL2013 open review
openreview.net/forum?id=idpCdOWtqXd60¬eId=C8Vn84fqSG8qa

The journal article explained the team's discovery that vector vectors of words allowed the same arithmetic that you might have seen in your calculus or linear algebra class.

Suddenly, with Mikolov's word vectors, questions like:

Portland Timbers + Seattle - Portland = ?

can be solved with vector algebra.

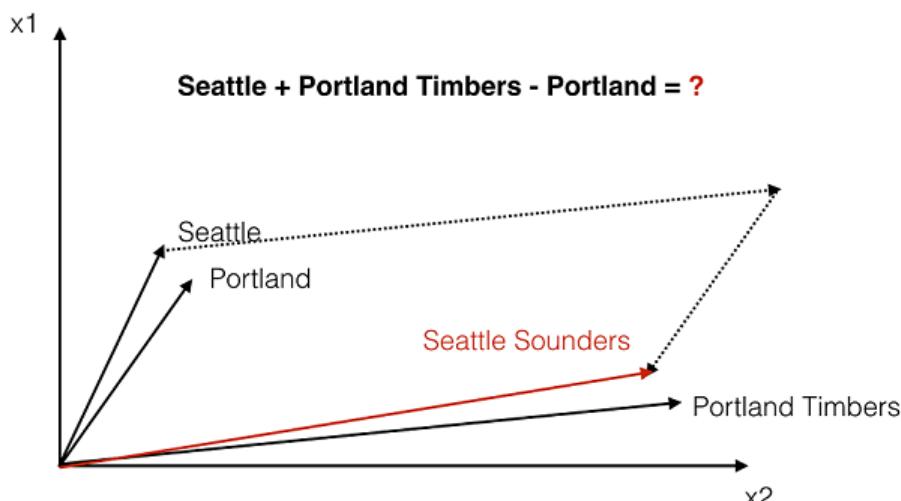


Figure 6.1 Geometry of Word2vec Math

As we will show in more detail later in this chapter, the `word2vec` model contains information about the relationships between words, including similarity. The `word2vec` model "knows" that the terms *Portland* and *Portland Timbers* are roughly the same distance apart as *Seattle* and *Seattle Sounders*. And those distances (differences between the pairs of vectors) are in roughly the same direction. So the `word2vec` model can be used to answer our sports team analogy question. We can add the difference between *Portland* and *Seattle* to the vector which represents the *Portland Timbers* and that should get us close to the vector for the term *Seattle Sounders*.

$$\begin{bmatrix} 0.0168 \\ 0.007 \\ 0.247 \\ \dots \end{bmatrix} + \begin{bmatrix} 0.093 \\ -0.028 \\ -0.214 \\ \dots \end{bmatrix} - \begin{bmatrix} 0.104 \\ 0.0883 \\ -0.318 \\ \dots \end{bmatrix} = \begin{bmatrix} 0.006 \\ -0.109 \\ 0.352 \\ \dots \end{bmatrix}$$

Figure 6.2 Example of the answer computation for the given question

When doing the math on the word vectors you will never end up exactly on another vector (remember these are 100-D continuous real-valued vectors), but the word closest to the resulting vector will often be the answer to our NLP question. The English word associated with that resultant is the natural language answer to our question about sports teams and cities!

Word2vec allows us to transform our natural language vectors of token occurrence counts and frequencies into the vector space of much lower-dimensional Word2vec vectors. In this lower dimensional space we can do our math, and then convert back to a natural language space. You can imagine how useful this is to a chatbot, search engine, question answering system, or information extraction algorithm.

NOTE

The initial paper in 2013 by Mikolov et al. was able to achieve an answer accuracy of only 40%. However, back in 2013 the approach outperformed any other semantic reasoning approach by a significant margin. Since the initial publication, the performance of `word2vec` has improved dramatically by training it on ever larger corpora, like the reference implementation trained on 100 billion words from the Google News Corpus. This is the implementation used in this book.

The research team also discovered that the difference between a singular and a plural word is often roughly the same magnitude, and in the same direction. In other words

$$\vec{x}_{\text{coffee}} - \vec{x}_{\text{coffees}} \approx \vec{x}_{\text{cup}} - \vec{x}_{\text{cups}} \approx \vec{x}_{\text{cookie}} - \vec{x}_{\text{cookies}}$$

Figure 6.3 Singular and plural version of a word show roughly the same distance

But their discovery didn't stop there. They also discovered that the distance relationships go far beyond simple singular vs plural relationships, but also apply to other semantic relationships. Suddenly, we were able to answer questions like

"San Francisco is to California as what is to Colorado?"

San Francisco - California + Colorado = Denver

MORE REASONS TO USE WORD VECTORS

Vector representations of words are useful not only for reasoning and analogy problems, but also for all the other things you use natural language vector space models for. From pattern matching to modeling and visualization, your NLP pipeline's accuracy and usefulness will improve if you use the improved accuracy word vectors from this chapter.

For example, later in this chapter we will show you how to visualize word vectors on 2D "semantic maps" like the one below. You can think of this like a cartoon map of a popular tourist destination or bus and train routes. In these cartoon maps, things that are close to each other semantically as well as geographically get squished together. For cartoon maps the artist adjusts the scale and position of icons for various locations to match the "feel" of the place. With word vectors, the machine too can have a feel for words. So your machine will be able generate impressionistic maps like this using word vectors from this chapter:¹¹⁹

Footnote 119 You can find the code for generating these interactive 2D word plots in [nlpia.book.examples.ch06_w2v_us_cities_visualization](#)

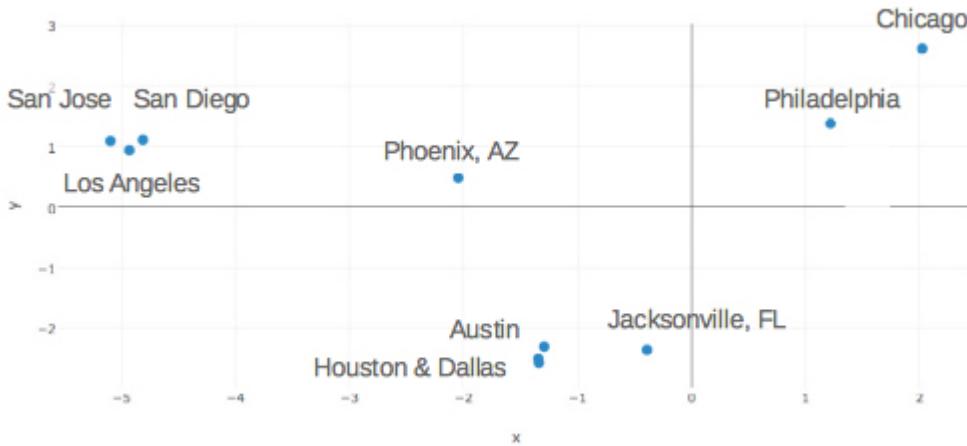


Figure 6.4 Word vectors for ten US cities projected onto a 2D map

If you're familiar with these US cities, you might realize that this isn't a very good geographic map, but it's a pretty good semantic map. I, for one, often confuse the two large Texas cities, Houston and Dallas, and they have almost identical word vectors. And the word vectors for the big California cities make a nice triangle of culture in my mind.

And word vectors are great for chatbots and search engines too. For these applications, word vectors can help overcome some of the rigidity, brittleness of pattern or keyword matching. Say you were searching for information about a famous person from Houston, Texas, but didn't realize they'd moved to Dallas. Looking at the "semantic map" above it looks like it would be easy for semantic search using word vectors to figure that out for you. And character or Part-of-Speech patterns wouldn't understand the difference between "Tell me about a Denver omelet." and "Tell me about the Denver Nuggets." But patterns based on word vectors would likely able to differentiate between the food item (omelet) and the basketball team (Nuggets) and respond to the user appropriately.

6.2.2 How to compute the Word2Vec Representations?

Word vectors represent the semantic meaning of words as vectors in the context of the training corpus. This allows us not just to answer analogy questions but also reason about the meaning of words with vector algebra. But how do we calculate these vector representations? There are two possible ways to train `word2vec` embeddings. The *skip-gram* approach predicts the context of words (*output words*) from a word of interest (*the input word*). The Continuous Bag of Words (CBOW) approach predicts the target word (*the output word*) from the nearby words (*input words*). We'll show you how and when to use each of these to train a Word2Vec model in the coming sections.

The computation of the word vector representations can be very resource intensive. Luckily, for most applications, you won't need to compute your own word vectors. You can rely on pretrained representations for a broad range of applications. Companies that deal with large corpora and can afford the computation have open sourced their pretrained word vector models. In the later sections, we will introduce to you how to use these other pretrained word models, like GloVe and FastText.

TIP

Pretrained word vector representations are available for corpora like Wikipedia, DBpedia, Twitter, and Freebase.¹²⁰ These pretrained models are great starting points for your word vector applications.

Footnote 120

github.com/3Top/word2vec-api#where-to-get-a-pretrained-model

- Google provides a pretrained `word2vec` model based on English Google News articles¹²¹

Footnote 121 bit.ly/GoogleNews-vectors-negative300

- Facebook published their word models, called *fastText*, for 294 languages¹²²

Footnote 122 github.com/facebookresearch/fastText

However, if your domain relies on specialized vocabulary or semantic relationships, general purpose word models won't be sufficient. For example, if the word *python* should unambiguously represent the programming language instead of the reptile, a domain specific word model will be needed. If you need to constrain your word vectors to their usage in a particular domain, then you will need to train them on text from that domain.

SKIP-GRAM APPROACH

In the skip-gram training approach, we are trying to predict the surrounding window of words based on an input word. In our Monet example, for the given word "*painted*" is the training input to the neural network. The corresponding training output examples for the skip-grams are the surrounding words "*Claude*", "*Monet*", "*the*" and "*Grand*".

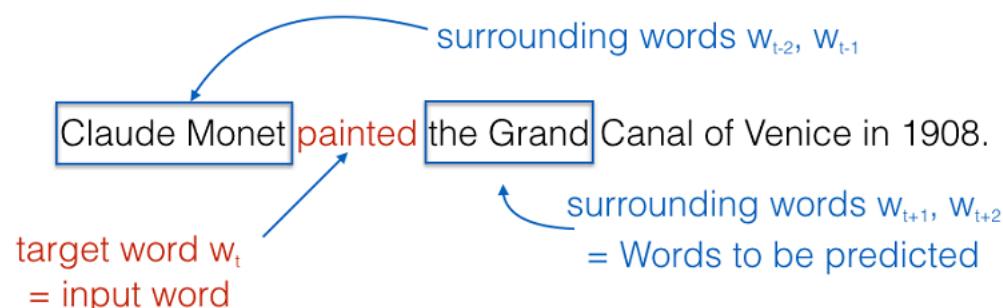


Figure 6.5 Training Input and Output Example for the Skip-gram Approach

NOTE**What is a skip-gram?**

Skipgrams are n -grams which contain gaps due to the fact that we skip over intervening tokens. In our example, we are predicting *Claude* from the input token *painted* and we skip over the token *Monet*.

The structure of the neural network which is used to predict the surrounding words is very similar to the networks we learned about in Chapter 5. As you can see in the next figure the network consists of two layers, where the hidden layer consists of N neurons where N is the number of vector dimensions used to represent a word. Both the input and output layers contain M neurons, where M is the number of words in the vocabulary of the model. The output layer activation function is a softmax, which is commonly used for classification problems.

NOTE**What is softmax?**

The softmax function is often used as the activation function in the output layer of neural networks when the network's goal is to learn classification problems. The softmax will squash the output results between 0 and 1 and the sum of all output notes will always add up to 1. That way, the results of an output layer with a softmax function can be considered as probabilities. If your output vector of a 3 neuron output layer would look like

$$v = \begin{bmatrix} 0.5 \\ 0.9 \\ 0.2 \end{bmatrix}$$

NOTE

The "squashed" vector after the softmax activation ¹²³

Footnote 123 For every of the K output notes, the output value can be calculated through the normalized exponential function above.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

NOTE

would look like

$$\sigma(v) = \begin{bmatrix} 0.309 \\ 0.461 \\ 0.229 \end{bmatrix}$$

The following two examples show the numerical network input and output for the first two surrounding words. In this case, the input word is "*Monet*" and the expected output of the network is either "*Claude*" or "*painted*", depending on the training pair.

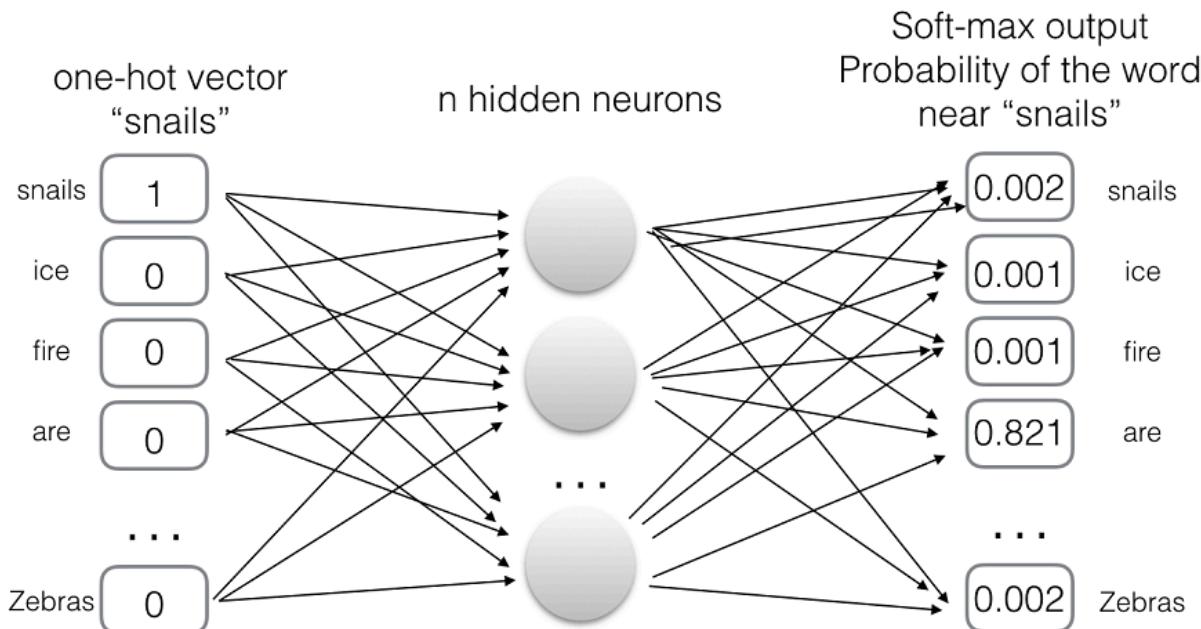


Figure 6.6 Network Example for the Skipgram Training

NOTE When you look at the structure of the neural network for the word embedding, you'll notice that the implementation looks similar to what we discovered in Chapter 5. If you would like to learn more about the details of neural networks, check out the last chapter.

HOW DOES THE NETWORK LEARN THE VECTOR REPRESENTATIONS?

To train a word2vec model we are using techniques from Chapter 2. For example, in the diagram above, $w(t)$ represents the one-hot vector for the token at position t . So if we want to train a word2vec model using a skip-gram window size (radius) of two words, that means we are considering the two words before and the following each target word. We would then use our 5-gram tokenizer from Chapter 2 to turn a sentence like

Claude Monet painted the Grand Canal of Venice in 1806.

into ten 5-grams with the target word as the center whereas n is the number of tokens in the sentence:

Table 6.1 Example word windows for a window size of 2 (before and after the target word)

| Input Word wt | Expected Output $wt-2$ | Expected Output $wt-1$ | Expected Output $wt+1$ | Expected Output $wt+2$ |
|-----------------|------------------------|------------------------|------------------------|------------------------|
| Claude | | | Monet | painted |
| Monet | | Claude | painted | the |
| painted | Claude | Monet | the | Grand |
| the | Monet | painted | Grand | Channel |
| Grand | painted | the | Canal | of |
| Canal | the | Grand | of | Venice |
| of | Grand | Channel | Venice | in |
| Venice | Canal | of | in | 1908 |
| in | of | Venice | 1908 | |
| 1908 | Venice | i | | |

The training set consisting of the input word and the surrounding (output) words are now the basis for the training of the neural network. In the case of four surrounding words, we would use four training iterations where each output word is being predicted based on the input word.

Each of the words are represented as one-hot vectors before they are presented to the network (see Chapter 2). The idea of a one-hot vector becomes useful to understand the output vector. The softmax activation of the output layer nodes (one for each token in the vocabulary) calculates the probability of an output word being found as a surrounding word of the input word. The output vector of word probabilities can then be converted into a one-hot vector where the word with the highest probability will be converted to 1 and all remaining terms will be set to 0. This simplifies the loss calculation.

Once training of the neural network is completed, you will notice that the weights have been trained to represent the semantic meaning. Thanks to the one-hot vector conversion of our tokens, each row in the weight matrix is representing a different word from your corpus. After the training semantically similar words will have very similar vectors because they were trained to predict very similar surrounding words. *This is purely magical!*

After the training is complete and you decide not to train your word model any further, the output layer of the network can be ignored. Only the weights of the inputs to the

hidden layer are used as the embeddings. Or in other words: The weight matrix is your word embedding. The dot product between the one-hot vector representing the input term and the weights then represents the *word vector embedding*.

NOTE

The weights of a hidden layer in a neural network are often represented as a matrix. One column per neuron, one row per weight, or the transpose of that depending. So if you take the dot product of a one hot vector with the trained weight matrix, you will get a vector that is basically one weight from each neuron (from each column of the matrix).

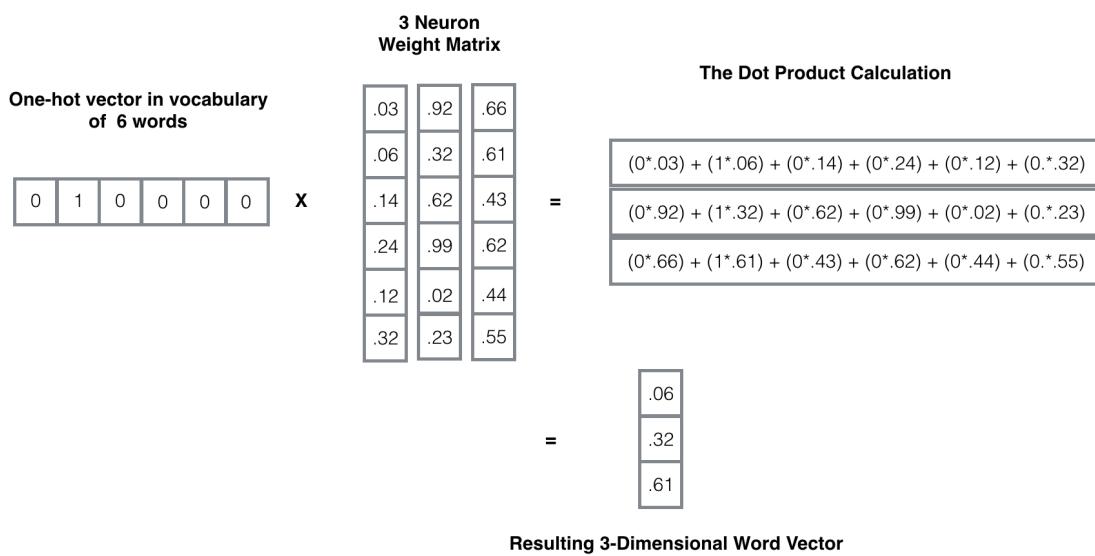


Figure 6.7 Conversion of One-Hot Vector to Word Vector

CONTINUOUS BAG OF WORDS APPROACH

In the Continuous Bag of Words (CBOW) approach, we are trying to predict the center word based on the surrounding words. Instead of creating pairs of input and output tokens, we'll create a multi-hot vector of all surrounding terms as an input vector. The multi-hot input vector is basically the sum of all one-hot vectors of the surrounding tokens to the center, target token.

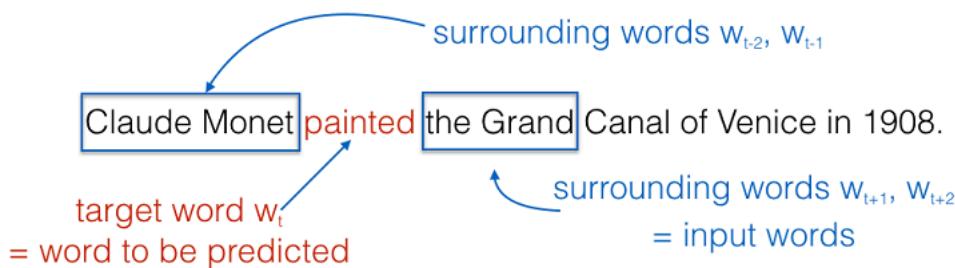


Figure 6.8 Training Input and Output Example for the CBOW Approach

| Input Word w_{t-2} | Input Word w_{t-1} | Input Word w_{t+1} | Input Word w_{t+2} | Expected Output w_t |
|----------------------|----------------------|----------------------|----------------------|-----------------------|
| | | Monet | painted | Claude |
| | Claude | painted | the | Monet |
| Claude | Monet | the | Grand | painted |
| Monet | painted | Grand | Canal | the |
| painted | the | Canal | of | Grand |
| the | Grand | of | Venice | Canal |
| Grand | Canal | Venice | in | of |
| Canal | of | in | 1908 | Venice |
| of | Venice | 1908 | | in |
| Venice | in | | | 1908 |

Based on the training sets, we can create our multi-hot vectors as inputs and map them to the target word as output. The multi-hot vector is the sum of the one-hot vectors of the surrounding words training pairs $w_{t-2} + w_{t-1} + w_{t+1} + w_{t+2}$. We then build the training pairs with the multi-hot vector as the input and the target word w_t as the output. During the training the output is derived from the softmax of the output node with the highest probability

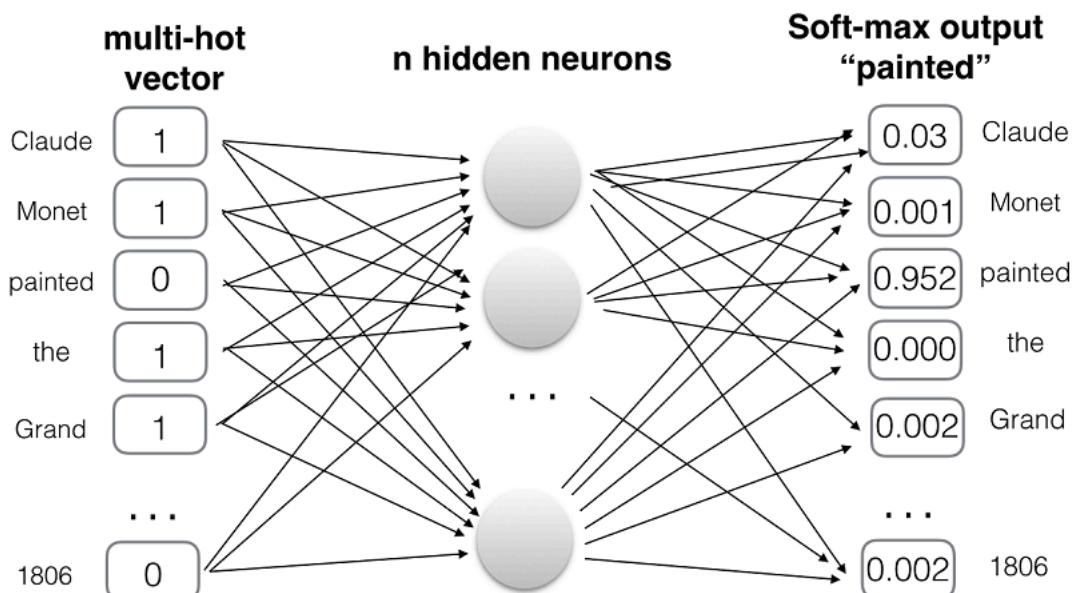


Figure 6.9 Continuous Bag of Words (CBOW) Word2vect network**NOTE****Continuous Bag of Words vs Bag of Words**

In previous chapters we introduced the concept of a bag of words, but what is the difference to a continuous bag of words? To establish the relationships between words in a sentence, we will look a certain window of words in relation to the word of interest. The size of the word window depends size of your corpus, accuracy of the word model, computational resources, etc. All words within the sliding window are considered to be the content of the continuous bag of words for the specific target word, therefore the continuous bag of words is changing while sliding over the document.

Continuous Bag of Words ↗

Claude Monet painted the Grand Canal of Venice in 1908.

Claude Monet painted the Grand Canal of Venice in 1908.

Claude Monet painted the Grand Canal of Venice in 1908.

Figure 6.10 Example for a continuous bag of words

SKIP-GRAM VS. CONTINUOUS BAG OF WORDS: WHEN TO USE WHICH APPROACH

Mikolov highlighted that the skip-gram approach works very well with small corpora and rare terms. This is because with the skip gram approach you'll have more examples due to the network structure. However, the continuous bag of words approach shows higher accuracies for frequent words and it is much faster to train.

COMPUTATIONAL TRICKS OF WORD2VEC

After the initial publication, the performance of word2vec models have been improved through various computational tricks. In this section, we want to highlight three of the improvements.

FREQUENT BI-GRAMS

Some words often occur in combination with other words, e.g. "Elvis" is often followed by "Presley" and therefore form bigrams. Since the word "Elvis" would with a very high probability occur with "Presley", we don't really gain much value from this prediction. In order to improve the accuracy of the `word2vec` embedding, Mikolov's team included some bigrams and trigrams as terms in the `word2vec` vocabulary. The team¹²⁴ used cooccurrence frequency to identify bigrams and trigrams that should be considered single terms using the following scoring function:

Footnote 124 The publication by the team around Tomas Mikolov is an interesting read:
arxiv.org/pdf/1310.4546.pdf

$$[score(w_i, w_j) = \frac{count(w_i w_j) - \delta}{count(w_i) \times count(w_j)}]$$

Figure 6.11 Bigram score

If the words w_i and w_j result in a high score and the score is above the threshold then they will be included in the `word2vec` vocabulary as a pair term. You will notice that the vocabulary of the model contains terms like "New_York" or "San_Francisco". The token of frequently occurring bigrams are combined through a character ("_" is usually used). That way, these terms will be represented as a single one-hot vector instead of two separate ones, e.g. for "San" and "Francisco".

Another effect of the word pairs is that the combination of the words often represents a different meaning than the individual words. For example, the MLS soccer team *Portland Timbers* has a very different meaning than the individual words *Portland* or *Timbers*. However, by adding often occurring bigrams like team names to the `word2vec` model, they can easily be included in the one-hot vector for model training.

SUBSAMPLING FREQUENT TOKENS

Another accuracy improvement to the original algorithm was to subsample frequent words. Common words like *the* or *a* often don't carry significant information. And the co-occurrence of the word *the* with a broad variety of other nouns in the corpus would create meaningless connections between words, muddying the `word2vec` representation with this false semantic similarity training. To avoid this, frequent words are sampled less often during training to not overweight them in the `word2vec` vector space. They are given less influence than the rarer, more important words in determining the ultimate embedded representations of words. The team proposed the following equation to determine the probability of sampling a given word (including it in a particular skipgram during training):

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

Figure 6.12 Word probability

In the earlier equation, $f(w_i)$ represents the frequency of a word across the corpus and t represents a threshold of a frequency from which we want to apply the subsampling. The threshold depends on your corpus size and document domain, but values between *10-5* and *10-6* are often found in the literature.

So, if a word shows up 10 times across your entire corpus of 1 million words, with a subsampling threshold of *10-6*, then the probability of keeping the word in any particular n -gram is 68%. You would skip it 32% of the time while composing your n -grams during tokenization.

NEGATIVE SUBSAMPLING

One last trick the Mikolov came up with was the idea of negative sampling. If a single training pair is presented to the network, it will cause all weights to be updated. However, if our vocabulary contains thousands or millions of words, then updating all weights for the large one-hot vector can be daunting and time consuming. To speed up the training of word models, the idea of negative subsampling becomes very useful.

Instead of updating all weights of the words which weren't included in the word window, the Google team suggested sampling just a few negative samples (in the output vector) to update their weights. Instead of updating all weights, we pick n negative samples (words

which didn't match our expected output) and update the weights that contributed to their specific output. That way, the computation can be reduced dramatically and the performance of the trained network doesn't decrease significantly.

NOTE

If you train your word model with a small corpus, you might want to use a negative sampling rate of 5-20 samples. For larger corpora, you can reduce the sample rate as low as 2-5 samples as suggested by Mikolov and his team.

6.2.3 How to use the *gensim.word2vec* module?

If the previous section sounded too complicated, don't worry. Various companies provide their pretrained word vector models and popular NLP libraries for different programming languages allow you to use the pretrained models efficiently. In the following section, we'll look at how you can take advantage of the magic of word vectors. For this purpose, we are using the popular *gensim* library which you saw in Chapter 4.

If you've already installed the *nlpia* package,¹²⁵ you can download a pretrained word2vec model with

Footnote 125 See the README file at github.com/totalgood/nlpia for installation instructions.

```
>>> from nlpia.data.loaders import get_data
>>> word_vectors = get_data('word2vec')
```

If that doesn't work for you, or you like to "roll your own", you can do a Google search for a word2vec models pretrained on "GoogleNews" documents.¹²⁶ Once you find and download the model in Google's original binary format and put it in a local path you can load it with the *gensim* package like this:

Footnote 126 Google hosts the original model trained by Mikolov on Google Drive [here](#)

```
>>> from gensim.models.keyedvectors import KeyedVectors
>>> word_vectors = KeyedVectors.load_word2vec_format('/path/to/the/model/GoogleNews-vectors-negative300.emb')
```

Working with word vectors can be very memory intensive. If your available memory is limited or if you don't want to wait minutes for the word vector model to load, you can reduce the number of words loaded into memory by passing in the *limit* keyword argument. In the following example, we will load the 200k most common words from the Google News corpus.

```
>>> from gensim.models.keyedvectors import KeyedVectors
>>> word_vectors = KeyedVectors.load_word2vec_format('/path/to/the/model/GoogleNews-vectors-negative300.')
```

However, keep in mind that a word vector model with a limited vocabulary will lead to a lower performance of your NLP pipeline. Therefore, we recommend only loading a limited word vector model during the development phase. Production-ready models and NLP implementations should always take advantage of the largest vocabulary you have available. For all future examples in the chapter, we are using the complete word model.

If you want to determine related terms, the *gensim* package provides a *most_similar* method. Unlike a conventional thesaurus, *word2vec* synonomy (similarity) is a continuous score, a distance. So you can find as many or as few related terms as you like. The keyword argument *positive* takes a list of terms which will be added, similar to our soccer team example from the beginning of this chapter. Similarly, the *negative* argument can be used for subtraction and to exclude unrelated terms. The argument *topn* determines how many related terms should be provided as a return value.

```
>>> word_vectors.most_similar(positive=['cooking', 'potatoes'], topn=5)
[('cook', 0.6973530650138855),
 ('oven_roasting', 0.6754530668258667),
 ('Slow_cooker', 0.6742032170295715),
 ('sweet_potatoes', 0.6600279808044434),
 ('stir_fry_vegetables', 0.6548759341239929)]
>>> word_vectors.most_similar(positive=['germany', 'france'], topn=1)
[('europe', 0.7222039699554443)]
```

Word vector models also allow you to determine unrelated terms. The *gensim* library provides you with a method called *doesnt_match*

```
>>> word_vectors.doesnt_match("potatoes milk cake computer".split())
'computer'
```

To determine the most unrelated term of the list, the method returns the term with the highest distance to all other list terms.

If you want to perform calculations (e.g. the famous example *king + woman - man = queen*, which was the example that got Mikolov and his advisor excited in the first place), you can do that by adding a *negative* argument to the *most_similar* method call.

```
>>> word_vectors.most_similar(positive=['king', 'woman'], negative=['man'], topn=2)
[('queen', 0.7118192315101624), ('monarch', 0.6189674139022827)]
```

The *gensim* library also allows you to calculate the similarity between two terms. If you

want to compare two words and determine their cosine similarity, use the method *similarity*.

```
>>> word_vectors.similarity('princess', 'queen')
0.70705315983704509
```

If you want to develop your own functions and work with the raw word vectors, you can access them through Python's *getitem* built-in functionality. You can treat the loaded model object as a dictionary where your word of interest is the dictionary key. Each float in the returned array represents one of the vector dimensions. In the case of Google's word model, your numpy arrays will have a shape of 1x300.

```
>>> word_vectors['phone']
array([-0.01446533, -0.12792969, -0.11572266, -0.22167969, -0.07373047,
       -0.05981445, -0.10009766, -0.06884766,  0.14941406,  0.10107422,
       -0.03076172, -0.03271484, -0.03125   , -0.10791016,  0.12158203,
       0.16015625,  0.19335938,  0.0065918 , -0.15429688,  0.03710938, ...]
```

If you're wondering what all those numbers *mean* you can find out, but it would take a lot of work. You would need to examine some synonyms and see which of the 300 numbers above they all share. Alternatively you can find the linear combination of these numbers that make up dimensions for things like "placeness" and "femaleness", like we did at the beginning of this chapter.

6.2.4 How to generate your own Word vector representations?

In some cases you may want to "roll your own" domain-specific word vector models. This can improve the accuracy of your model if your NLP pipeline is processing documents that use words in a way that you wouldn't find GoogleNews before 2006, when Mikolov trained the reference `word2vec` model. Keep in mind, you need a *lot* of documents to do this as well as Google and Mikolov did. But if your words are particularly rare on Google News, or your texts use them in unique ways within a restricted domain, e.g. medical texts or transcripts, a domain-specific word model may improve your model accuracy. In the following section, we will show you how to train your own `word2vec` model.

For the purpose of training a domain-specific `word2vec` model, we will again turn to *gensim*, but before we can start training the model, we will need to preprocess our corpus with the tools we discovered in Chapter 2.

PREPROCESSING STEPS

First we need to break our documents into sentences and the sentences into tokens. *gensim's* word2vec model expects a list of sentences, where each sentence is broken up into tokens. Your training input should look similar to this structure below:

```
>>> token_list
[
    ['to', 'provide', 'early', 'intervention/early', 'childhood', 'special', 'education',
     'services', 'to', 'eligible', 'children', 'and', 'their', 'families'],
    ['essential', 'job', 'functions'],
    ['participate', 'as', 'a', 'transdisciplinary', 'team', 'member', 'to', 'complete',
     'educational', 'assessments', 'for']
    ...
]
```

To segment sentences and then convert sentences into tokens, you can apply the various strategies we learned in Chapter 2. Detector Morse is a sentence segmenter that improves upon the accuracy segmenter available in NLTK and gensim for some applications.¹²⁷. Once you have converted your documents into lists of token lists (one for each sentence), you are already for your word2vec training.

Footnote 127 Detector Morse by Kyle Gorman and OHSU on pypi and at github.com/cslu-nlp/DetectorMorse is a sentence segmenter with state-of-the-art performance (98%) and has been pretrained on sentences from years of text in the Wall Street Journal. So if your corpus includes language similar to that in the WSJ, DectectorMorse is likely to give you the highest accuracy currently possible (around 98%). *DetectorMorse* can also be retrained on your own dataset if you have a large set of sentences from your domain

TRAIN YOUR DOMAIN-SPECIFIC WORD2VEC MODEL

Now we can get started by loading the *word2vec* module.

```
>>> from gensim.models.word2vec import Word2Vec
```

The training requires a few setup details

```
>>> num_features = 300          ①
>>> min_word_count = 3          ②
>>> num_workers = 2              ③
>>> window_size = 6              ④
>>> subsampling = 1e-3           ⑤
```

- ① Number of vector elements (dimensions) to represent the word vector
- ② Min number of word count to be considered in the word2vec model. If your corpus

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

is small, reduce the min count. Increase the min count if you are training with a large corpus.

- ③ Number of CPU cores used for the training. If you want to set the number of cores dynamically, check out import multiprocessing; num_workers = multiprocessing.cpu_count()
- ④ Context window size
- ⑤ Subsampling rate for frequent terms

With this, you are ready to start our training

```
>>> model = Word2Vec(
...     token_list,
...     workers=num_workers,
...     size=num_features,
...     min_count=min_word_count,
...     window=window_size,
...     sample=subsampling)
```

Depending on the size of your corpus, the training can take some time. For smaller corpora, the training can be completed within minutes. However, for a comprehensive word model, the corpus size should be much larger than a few thousand documents. You need to have several examples of all the different ways all the different words in your corpus are used. If you start processing larger corpora, e.g. the Wikipedia corpus, expect a much longer training time and a much larger memory consumption.

`word2vec` models can consume quite a bit of memory. But remember that only the weight matrix for the hidden layer is of interest. Once you have trained your word model sufficiently, you can reduce the memory footprint by freezing your model and discarding unnecessary information. If you run

```
>>> model.init_sims(replace=True)
```

The `gensim` model will "freeze" itself, storing the weights of the hidden layer and discarding the output weights which predict word cooccurrences. This reduces the memory consumption by roughly 50%. The output weights aren't part of the vector used for most `word2vec` applications. However the model cannot be trained further once the weights of the output layer have been discarded.

You can save the trained model with the following command and preserve it for later use:

```
>>> model_name = "my_domain_specific_word2vec_model"
>>> model.save(model_name)
```

If you want to test your newly trained model, you can use it with the same method we learned in the previous section.

```
>>> from gensim.models.word2vec import Word2Vec
>>> model_name = "my_domain_specific_word2vec_model"
>>> model = Word2Vec.load(model_name)
>>> model.most_similar('radiology')
```

6.2.5 Word2vec vs GloVe (*Global Vector*)

Word2vec was a breakthrough, but it relies on a neural network model which must be trained using backpropagation. Backpropagation is usually less efficient than direct optimization of a cost function using gradient descent. Stanford NLP researchers¹²⁸ led by Jeffrey Pennington set about to understand the reason why Word2vec worked so well and find the cost function that was being optimized. They started by counting the word co-occurrences and recording them in a square matrix. They found they could compute the singular value decomposition (SVD)¹²⁹ of this co-occurrence matrix, splitting it into the same two weight matrices that Word2vec produces.¹³⁰ The key was to normalize the co-occurrence matrix the same way. However, in some cases the Word2vec model failed to converge to the same global optimum that the Stanford researchers were able to achieve with their SVD approach. It's this direct optimization of the global vectors of word co-occurrences (co-occurrences across the entire corpus) that gives GloVe its name.

Footnote 128 Stanford GloVe Project: nlp.stanford.edu/projects/glove/

Footnote 129 See Chapter 5 and Appendix C for more details on SVD

Footnote 130 *GloVe: Global Vectors for Word Representation* by Jeffrey Pennington, Richard Socher, and Christopher D. Manning: nlp.stanford.edu/pubs/glove.pdf

Thus GloVe can produce matrices equivalent to the input weight matrix and output weight matrix of Word2vec, producing a language model with the same accuracy as Word2vec, but in much less time. GloVe accomplishes this by using the text data more efficiently. GloVe can be trained on smaller corpora and still converge.¹³¹ And SVD algorithms have been refined for decades, so GloVe has a head start on the debugging and optimization tweaks that Mikolov and word2vec just started. Whereas backpropagation of the weights in the neural nets that Mikolov relied on has only recently become efficient and accurate enough to produce useful models.

Footnote 131 Gensim's comparison of Word2vec and GloVe performance:
rare-technologies.com/making-sense-of-Word2vec/#glove_vs_word2vec

Even though Word2vec first popularized the concept of semantic reasoning with word vectors, your workhorse should probably be GloVe to train new word vector models. If you do, you'll be more likely to find the global optimum for those vector representations, giving you more accurate results.

Advantages of GloVe:

- Faster training
- Better RAM/CPU efficiency (large corpora)
- More efficient use of data (small corpora)
- More accurate for the same amount of training

6.2.6 fastText

Researchers from Facebook took the concept of Word2vec one step further ¹³² by adding a new twist to the model training. The new algorithm, which they named *fastText*, predicts the surrounding *n-character* grams rather than just the surrounding words, like Word2vec. For example, the word "whisper" would generate the following 2 and 3-character grams:

Footnote 132 Enriching Word Vectors with Subword Information, Bojanowski et al.:
arxiv.org/pdf/1607.04606.pdf

"wh", whi, hi, his, ...

fastText is then training a vector representation for every *n*-character gram, this includes words, misspelled words, partial words, and even single characters. The advantage of this approach is that it handles rare words much better than the original Word2vec approach.

As part of the fastText release, Facebook published pretrained fastText models for 294 languages. On the Github page of Facebook research ¹³³, you can find models ranging from *Abkhazian* to *Zulu*. The model collection even includes very rare languages like *Saterland Frisian* which is only spoken by a handful of Germans. The pretrained fastText models provided by Facebook have only been trained on the available Wikipedia corpora. Therefore the vocabulary and accuracy of the models will vary across languages.

Footnote 133 github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md

HOW TO USE THE PRETRAINED FASTTEXT MODELS?

The use of fastText is just like using Google's Word2vec model. Head over to the model repository¹³⁴ and download *bin+text* model for your language of choice. After the download is completed, unzip the binary language file. With the following code you can then load it into gensim:

Footnote 134 github.com/facebookresearch/fastText/blob/master/pretrained-vectors.md

```
>>> from gensim.models.fasttext import FastText
>>> ft_model = FastText.load_fasttext_format(model_file=MODEL_PATH)
>>> ft_model.most_similar('soccer')
```

- 1
- 2
- 3

- ➊ If you are using a gensim version before 3.2.0, you need to change this line to from `gensim.models.wrappers.fasttext import FastText`
- ➋ The `model_file` should point to the folder where you have stored the model's bin and vec files
- ➌ After loading the model, use it like any other word model in gensim

Gensim's fastText API shares a lot of functionality with the Word2vec implementations. All methods you learned about earlier in this chapter also apply to the fastText models.

6.2.7 Word2vec vs LSA

You might now be wondering how Word2vec and GloVe word vectors compare to the LSA (Latent Semantic Analysis) topic-word vectors of Chapter 4. Even though we didn't say much about the LSA topic-document vectors in Chapter 4, LSA gives us those too. LSA topic-document vectors are the sum of the topic-word vectors for all the words in those documents. If we wanted to get a word vector for an entire document that is analogous to topic-document vectors we'd sum all the word2vec word vectors in each document. That's pretty close to how Doc2vec document vectors work. We'll show you those further on in this chapter.

If your LSA matrix of topic vectors is of size $N_{words} \times N_{topics}$, then the LSA word vectors are the rows of that LSA matrix. These row vectors capture the meaning of words in a sequence of around 200-300 real values like Word2vec does. And LSA topic-word vectors are just as useful as word2vec vectors for finding both related and unrelated terms. As you learned in the GloVe discussion above, word2vec vectors can be created using the exact same SVD algorithm used for LSA. However, word2vec gets more use out of the same number of words in its documents by creating a sliding window that

overlaps from one document to the next. This way it can reuse the same words 5 times before sliding on.

What about incremental or online training? Both LSA and word2vec algorithms allow adding new documents to our corpus and adjusting our existing word vectors to account for the co-occurrences in the new documents. However, only the existing "bins" in our lexicon can be updated. Adding completely new words would change the total size of your vocabulary and therefore your one-hot vectors would change. That requires starting the training over if you want to capture the new word in your model.

LSA trains faster compared to Word2vec. And for long documents it does a better job of discriminating and clustering those documents.

The "killer app" for word2vec is the semantic reasoning it popularized. LSA topic-word vectors can do that to, but it usually isn't very accurate. You'd have to break documents into sentences and then only use short phrases to train your LSA model if you want to approach the accuracy and "wow" factor of word2vec reasoning. With word2vec you can determine the answer to questions like "Harry Potter" + "University" = "Hogwarts"¹³⁵

Footnote 135 As a great example for domain-specific word2vec models, check out the models around Harry Potter, the Lord of the Rings, etc. github.com/nchah/word2vec4everything#harry-potter

Advantages of LSA:

- Faster training
- Better discrimination between longer documents

Advantages of Word2vec and GloVe:

- More efficient use of large corpora
- More accurate vector-oriented reasoning, like analogy question answering

6.2.8 Visualizing Word Relationships

The semantic word relationships can be very powerful and its visualizations can lead to interesting discoveries. In this section, we will demonstrate steps to visualize the word vectors in 2D.

NOTE

If you need a quick visualization of your word model, we highly recommend using Google's *Tensorboard* word embedding visualization functionality. For more details, check out our section on *How to Visualize Word Embeddings* in Chapter 13 *Scaling Up*.

To get started, let's load all the word vectors from the Google Word2vec model of there Google News corpus. As you can imagine, this corpus included a lot of mentions of "Portland" and "Oregon" and a lot of other city and state names. We'll use the `nlpia` package to keep things simple, so you can start playing with Word2vec vectors quickly.

```
>>> import os
>>> from nlpia.data.loaders import get_data
>>> from gensim.models.word2vec import KeyedVectors
>>> wordvector_path = get_data('word2vec') ①
>>> wv = KeyedVectors.load_word2vec_format(wordvector_path, binary=True)
>>> len(wv.vocab)
3000000
```

- ① This downloads a copy of the pretrained Google News word vectors to `os.path.join(BIGDATA_PATH, 'GoogleNews-vectors-negative300.bin.gz')`

WARNING

The Google News `word2vec` model is huge: 3 million words with 300 vector dimensions each. The complete word vector model will require 3 GB of available memory. If your available memory is limited or you quickly want to load a few most frequent terms from the word model, check out Chapter 13, "Scaling Up".

This `KeyedVectors` object in `gensim` now holds a table of 3 million `word2vec` vectors. We loaded these vectors from a file created by Google to store a Word2vec model that they trained on a large corpus based on Google News articles. There should definitely be a lot of words for states and cities in all those news articles. Here's just a few of the words in our vocabulary, starting at the 1 millionth word:

```
>>> import pandas as pd
>>> vocab = pd.Series(wv.vocab)
>>> vocab.iloc[1000000:100006]
Illington_Fund          Vocab(count:447860, index:2552140)
Illingworth              Vocab(count:2905166, index:94834)
Illingworth_Halifax      Vocab(count:1984281, index:1015719)
Illini                   Vocab(count:2984391, index:15609)
IlliniBoard.com           Vocab(count:1481047, index:1518953)
Illini_Buffs              Vocab(count:2636947, index:363053)
```

Notice that compound words and common *n*-grams are joined together with an underscore character ("_"). Also notice that the "value" in the key-value mapping is a `gensim` `Vocab` object which contains not only the index location for a word, so we can retrieve the Word2vec vector, but also the number of times it occurred in the Google News corpus.

As you have seen earlier, if you want to retrieve the 300-D vector for a particular word, you can use the square brackets on this `KeyedVectors` object to `getitem` any word or *n*-gram:

```
>>> wv['Illini']
array([ 0.15625   ,  0.18652344,  0.33203125,  0.55859375,  0.03637695,
       -0.09375   , -0.05029297,  0.16796875, -0.0625     ,  0.09912109,
      -0.0291748   ,  0.39257812,  0.05395508,  0.35351562, -0.02270508,
       ...]
```

The reason we chose the 1 millionth word (in lexical alphabetic order), is because first several thousand "words" are punctuation sequences like "#" and other symbols that occurred a lot in the Google News Corpus. We just got lucky that "Illini" ¹³⁶ showed up in our list. Let's see how "close" this "Illini" vector is to the vector for "Illinois":

Footnote 136 The word "Illini" refers to a group of people, usually football players and fans, rather than a single geographic region like "Illinois" (where most of fans of the "Fighting Illini" live).

```
>>> import numpy as np
>>> np.linalg.norm(wv['Illinois'] - wv['Illini'])           ①
3.3653798
>>> np.dot(wv['Illinois'], wv['Illini']) / (
...     np.linalg.norm(wv['Illinois']) * np.linalg.norm(wv['Illini'])) ②
0.5501352
>>> 1 - __
0.4498648
```

- ① Euclidean distance
- ② Cosine similarity
- ③ Cosine distance

These distances mean that the words "Illini" and "Illinois" are only moderately close to one another in meaning.

Now let's retrieve all the Word2vec vectors for US cities so we can use their distances to plot them on a 2D map of meaning. How would you find all the cities and states in that Word2vec vocabulary in that `KeyedVectors` object? You could use cosine distance like we did above to find all the vectors that are close to the words "state" or "city".

But rather than reading through all 3 million words and word vectors, lets load another dataset containing a list of cities and states (regions) from around the world.

```
>>> from nlpiab.data.loaders import get_data
>>> cities = get_data('cities')
```

```
>>> cities.head(1).T
geonameid           3039154
name                El Tarter
asciiname          El Tarter
alternatenames     Ehl Tarter,
latitude            42.5795
longitude           1.65362
feature_class        P
feature_code         PPL
country_code         AD
cc2                 NaN
admin1_code          02
admin2_code          NaN
admin3_code          NaN
admin4_code          NaN
population          1052
elevation            NaN
dem                  1721
timezone             Europe/Andorra
modification_date    2012-11-03
```

This dataset from Geocities contains a lot of information, including Latitude and Longitude and population. You could use this for some fun visualizations or comparisons between geographic distance and Word2vec distance. But for now we're just going to try to map that Word2vec distance on a 2D plane and see what it looks like. Let's focus on just the United States for now:

```
>>> us = cities[(cities.country_code == 'US') & (cities.admin1_code.notnull())].copy()
>>> states = pd.read_csv('http://www.fonz.net/blog/wp-content/uploads/2008/04/states.csv')
>>> states = dict(zip(states.Abbreviation, states.State))
>>> us['city'] = us.name.copy()
>>> us['st'] = us.admin1_code.copy()
>>> us['state'] = us.st.map(states)
>>> us[us.columns[-3:]].head()
      city   st   state
geonameid
4046255    Bay Minette  AL  Alabama
4046274        Edna  TX    Texas
4046319  Bayou La Batre  AL  Alabama
4046332      Henderson  TX    Texas
4046430       Natalia  TX    Texas
```

Now we have a full state name for each city in addition to its abbreviation. Let's check to see which of those state names and city names exist in our Word2vec vocabulary:

```
>>> vocab = pd.np.concatenate([us.city, us.st, us.state])
>>> vocab = np.array([word for word in vocab if word in wv.wv])
>>> vocab[:10]
```

Even when we only look at United States cities, we'll find a lot of large cities with the same name, like Portland, Oregon and Portland, Maine. So let's incorporate into our city

vector the essence of the state where that city is located. The way you combine the meanings of words in Word2vec is just to add the vectors together. That's the magic of "vector oriented reasoning."

Here's one way to add the Word2vecs for the states to the vectors for the cities and put all these new vectors in a big DataFrame. We use either the full name of a state or just the abbreviations (whichever one is in our Word2vec vocabulary)

```
city_plus_state = []
for c, state, st in zip(us.city, us.state, us.st):
    if c not in vocab:
        continue
    row = []
    if state in vocab:
        row.extend(wv[c] + wv[state])
    else:
        row.extend(wv[c] + wv[st])
    city_plus_state.append(row)
us_300D = pd.DataFrame(city_plus_state)
```

Depending on your corpus, your word relationship can represent different attributes, e.g. geographical proximity, cultural or economic similarities. But the relationships heavily depend on the training corpus and they will reflect the corpus.

WARNING Word Vectors are biased!

Word vectors learn word relationship based on the training corpus and word model will be optimized to reflect the corpus's semantics. Depending on your training corpus, your word vector models can be biased! Be considerate of the source of your corpus. Otherwise the word models could cause tremendous consequences.

The example below shows the potential bias of a word model. If we calculate the distance between *man* and *nurse* as well as *woman* and *nurse*, we will notice that the term *woman* is closer to the term *nurse* than the term *man*.

```
>>> word_model.distance('man', 'nurse')
0.74527711742556
>>> word_model.distance('woman', 'nurse')
0.5586440322993163
```

Depending on your corpus, these biases can be stronger or weaker. The identification and the exclusion of these biases is an ongoing topic in the current NLP research and we urge the readers to be mindful of the implications.

The news articles used as the training corpus share a common component which is the semantical similarity of the cities. Semantically similar locations in the articles seems to

interchangable and therefore the word model learned that they are similar. If we would have trained on a different corpus, our word relationship might have differed.

Cities that are similar in size and culture are clustered close together despite being far apart geographically, like San Diego and San Jose, or vacation destinations like Honolulu and Reno.

Fortunately we can use conventional algebra to add the vectors for cities to the vectors for states and state abbreviations. As we discovered in Chapter 4, we can use tools like the *Principal Components Anslysis* (PCA) to reduce the vector dimensions from our 300 dimensions to a human-understandable 2D representation. The PCA will reduced number of vector dimensions while keeping the variability of the data. This feature we discovered in Chapter 4 is extremely powerful, especially when visualizing higher dimensional vectors like word vectors.

We don't even have to normalize the length of the vectors after summing the city + state + abbrev vectors, since PCA will take care of that for us. However, that last line has quite a bit of "data wrangling" complexity. It deals with "South_Carolina" and "South_Dakota" which are surprisingly absent from the Google News word vector embeddings. Whenever a state's full name isn't available we just use the abbreviation twice in the sum. That way city vectors in SC and SD won't be short-changed on the cultural and semantic features of their parent state names too much.

We saved these "augmented" city word vectors in the `nlpia` package so you can load them to use in your application. Below we load them to perform PCA so we can plot them in 2 dimensions.

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> us_300D = get_data('cities_us_wordvectors')
>>> us_2D = pca.fit_transform(us_300D.iloc[:, :300])
```

①

②

- ➊ PCA here is for visualization of high dimensional vectors only, not for calculating Word2vec vectors
- ➋ The last column (# 301) of this DataFrame contains the name, which is also stored in the DataFrame index.

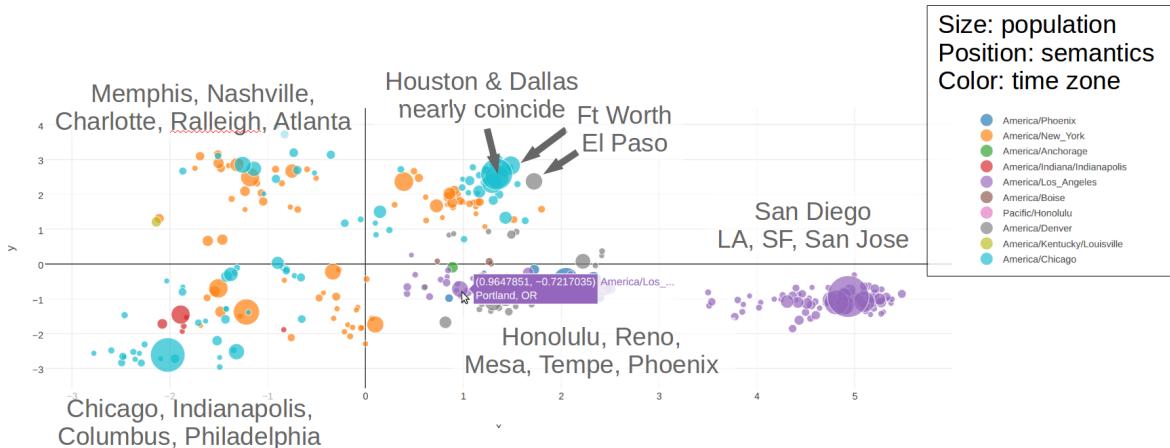


Figure 6.13 GoogleNews Word2vec 300D vectors projected onto a 2D map using PCA

NOTE

Low semantic distance (distance values close to zero) represent high similarity between words. The semantic distance, or "meaning" distance, is determined by the words occurring nearby in the documents used for training. The Word2vec vectors for two terms are *close* to each other in word vector space if they are often used in similar contexts (used with similar words nearby). For example *San Francisco* is *close* to *California* because they are often occur nearby in sentences and the distribution of words used near them are similar. A large distance between two terms expresses a low likelihood of shared context and thus shared meaning (they are semantically dissimilar), like *cars* and *peanuts*.

If you'd like to explore the city map above, or try your hand at plotting some vectors of your own, here's how. We've built a wrapper for plotly's offline plotting API that should help it handle DataFrames where you've "denormalized" your data. The plotly wrapper expects a DataFrame with a row for each sample and column for features you'd like to plot. These can be categorical features (like time zones) and continuous real-valued features (like city population). The resulting plots are interactive and useful for exploring many types of machine learning data, especially vector-representations of complex things like words and documents.

```
>>> import seaborn
>>> from matplotlib import pyplot as plt
>>> from nlpia.plots import offline_plotly_scatter_bubble
>>> df = get_data('cities_us_wordvectors_pca2_meta')
>>> html = offline_plotly_scatter_bubble(
...     df.sort_values('population', ascending=False)[:350].copy().sort_values('population'),
...     filename='plotly_scatter_bubble.html',
...     x='x', y='y',
...     size_col='population', text_col='name', category_col='timezone',
```

```
...      xscale=None, yscale=None, # 'log' or None
...      layout={}, marker={'sizeref': 3000})
{'sizemode': 'area', 'sizeref': 3000}
```

To produce the 2D representations of your 300D word vectors you need to use a dimension reduction technique. We used PCA (Principle Component Analysis). To reduce the amount of information lost during the compression from 300D to 2D it helps to reduce the range of information contained in the input vectors. So we limited our word vectors to those associated with cities. This is like limiting the domain or subject matter of a corpus when computing TF-IDF (Term Frequency Inverse Document Frequency) or BOW (Bag of Words) vectors.

For a more diverse mix of vectors with greater information content, you will probably need a nonlinear embedding algorithm like TSNE (T-Distributed Nonlinear Embedding). We'll talk about TSNE and other neural net techniques in later chapters. TSNE will make more sense once you've grasped the word vector embedding algorithms here.

6.2.9 Unnatural Words

Word embeddings like Word2vec are useful not only for English words but also for any sequence of symbols where the sequence and proximity of symbols is representative of their meaning. If your symbols have semantics, embeddings may be useful. As you may have guessed, word embeddings also work for other languages than English. Embedding work also for pictoral languages like traditional Chinese and Japanese (Kanji) or the mysterious heiroglyphics in Egyptian tombs. Embeddings and vector-based reasoning even works for languages that attempt to obfuscate the meaning of words. You can do vector-based reasoning on a large collection of "secret" messages transcribed from "Pig Latin" or any other language invented by children or the Emperor of Rome. A *Ceaser Cipher*¹³⁷ like RO13 or a *substitution cipher*¹³⁸ are both vulnerable to vector-based reasoning with Word2vec. You don't even need a decoder ring. You just need a large collection of messages or *n*-grams that your Word2vec embedder can process to find cooccurrences of words or symbols.

Footnote 137 en.wikipedia.org/wiki/Caesar_cipher

Footnote 138 en.wikipedia.org/wiki/Substitution_cipher



Figure 6.14 Decoder Rings

Word2vec has even been used to glean information and relationships from unnatural words or ID numbers like college course numbers ("CS-101"), model numbers ("Koala E7270", "Galaga Pro"), and even serial numbers, phone numbers, and zip codes.¹³⁹ To get the most useful information about the relationship between ID numbers like this, you'll need a variety of sentences that contain those ID numbers. And if the ID numbers often contain a structure where the position of a symbol has meaning, it can help to tokenize these ID numbers into their smallest semantic packet (like words or syllables in natural languages).

Footnote 139 medium.com/towards-data-science/a-non-nlp-application-of-word2vec-c637e35d3668

6.2.10 Document Similarity with Doc2vec

The concept of Word2vec can also be extended to sentences, paragraphs or entire documents. The idea of predicting the next word based on the previous words can be extended by training a paragraph or document vector¹⁴⁰. In this case, the prediction not only considers the previous words, but also the vector representing the paragraph or the document. It can be considered as an additional word input to the prediction. Over time, the algorithm learns a document or paragraph representation from the training set.

Footnote 140 arxiv.org/pdf/1405.4053v2.pdf

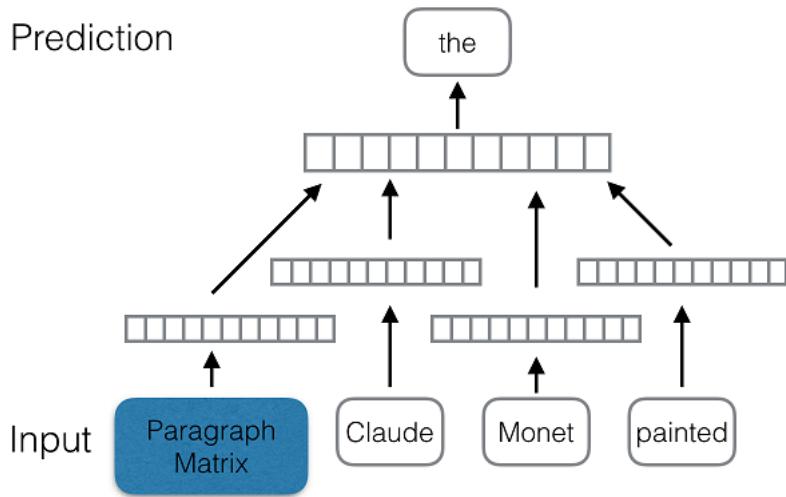


Figure 6.15 Doc2vec training uses an additional document vector as input

How are documents vectors generated for unseen documents after the training phase? During the *inference stage*, the algorithm adds more document vectors to the document matrix and "trains" the added vector based on the *frozen* word vector matrix, and its weights. By inferring a document vector, we can now create a semantic representation of the whole document.

By expanding the concept of Word2vec with an additional document or paragraph vector used for the word prediction, we can now use the trained document vector for various tasks, e.g. finding similar documents in a corpus.

HOW TO TRAIN DOCUMENT VECTORS?

Similar to our training of word vectors, we are using the *Gensim* package to train document vectors.

```

import multiprocessing
from gensim.models.doc2vec import TaggedDocument, Doc2Vec
from gensim.utils import simple_preprocess

cores = multiprocessing.cpu_count()

corpus = ['This is the first document ...', 'another document ...']

training_corpus = []
for i, text in enumerate(corpus):
    tagged_doc = TaggedDocument(simple_preprocess(text), [i])
    training_corpus.append(tagged_doc)

model = Doc2Vec(size=100, min_count=2, workers=cores, iter=10)
model.build_vocab(training_corpus)
model.train(training_corpus, total_examples=model.corpus_count, epochs=model.iter)

```

- ➊ Python's standard library provides a multiprocessing module which is handy for setting up the model training
- ➋ The gensim package provides a module to train and infer document vectors, as well as pre-process your documents
- ➌ The training can be accelerate if multiple CPU cores can be used. `cpu_count` will determine the number of cores
- ➍ Have your corpus is memory or a generator prepared to provide the training data
- ➎ Thank you MEAP reader 24231 for suggesting a more efficient numpy data structure
- ➏ gensim provides a data structure to annotate documents with labels. In our case, we give every document an individual ID, but you could also train the model to label documents with specific labels.
- ➐ Instantiate the Doc2Vec object. In our example, we use a window size of 10 words and a vector length of 100
- ➑ Before the model can be trained, the training vocabulary needs to build
- ➒ Kick off the training step, in our case for 10 epochs

TIP

If you are running low on RAM, and you know the number of documents ahead of time (i.e. your corpus object isn't a iterator or generator) then you might want to use a preallocated `numpy array` instead of Python list for your `training_corpus`:

```
training_corpus = np.empty(len(corpus),
                           dtype=object); ... training_corpus[i] = ...
```

Once the Doc2vec model is trained, you can infer document vectors for new, unseen documents by calling `infer_vector` on the instantiated and trained model.

```
model.infer_vector(simple_preprocess('This is an completely unseen document'), steps=10)
```

➌

- ➌ Doc2vec requires a "training" step when inferring new vectors. In our example, we update the trained vector through 10 steps (or iterations)

With these few steps, you can quickly train an entire corpus of documents and find similar documents. You could do that by generating a vector for every document in your corpus and then calculating the cosine distance between each document vector. Another common task is to cluster the vectors of the documents of a corpus with something like *K-Means* to create a document classifier..

6.2.11 Summary

In this chapter you've learned

- What *Vector-Oriented Reasoning* is and some ideas for using it in your NLP pipeline
- How to train `word2vec` models on the vocabulary in your domain
- How to use `gensim` in your NLP pipeline to take advantage of existing word vector models that have been pretrained on millions of documents
- How Word2Vec is a lot like the semantic analysis you learned about in Chapter 4, only better (for some applications)
- How to apply Word2Vec to unnatural language like ID's and genomic data

In the next chapters, you'll discover some more applications for word vectors. They are especially powerful when combined with Convolutional Neural Nets for classification tasks. And we'll show you how to merge broad pre-trained word vector models with precise domain-specific word vectors trained on your corpus in your domain. This will help you power-up your smaller corpus to the greater accuracy of word vectors trained on much larger corpora.



Getting Words in Order with Convolutional Neural Networks (CNNs)

In this chapter

- Using Neural Networks for NLP
- Finding meaning in word patterns
- Building a Convolutional Neural Network (CNN)
- Vectorizing natural language text in a way that suits neural networks
- Training a CNN
- Classifying the sentiment of novel text

Language's true power is not necessarily in the words themselves, but in the spaces between the words, in the order and combination of words. Sometimes meaning is hidden beneath the words, in the intent and emotion that formed that particular combination of words. Understanding the intent beneath the words is a critical skill for an empathetic, emotionally intelligent listener or reader of natural language, be it human or machine.¹⁴¹ Just as in thought and ideas, it's the connections between words that create depth, information, complexity. With a grasp on the meaning of individual words, and multiple clever ways to string them together, how do we look beneath them and measure the meaning of a combination of words with something more flexible than counts of N-gram matches? How do we find meaning, emotion, "*latent semantic information*", from a sequence of words, so we can do something with it? And even more ambitious, how do we impart that hidden meaning to text generated by a cold, calculating machine?

Footnote 141 *International Association of Facilitators Handbook*,
books.google.com/books?id=TgWsY7oSgtsC&lpg=PT35&dq=%22beneath%20the%20words%22%20empathy%20listenerir

Even the phrase "machine-generated text" inspires dread of a hollow, tinned voice issuing a chopped list of words. Machines may get the point across, but little more than

that. What's missing? The tone, the flow, the character that you expect a person to express in even the most passing of engagements. Those subtleties exist between the words, underneath the words, in the patterns of how they are constructed. As a person communicates, they will underlay patterns in their text and speech. Truly great writers and speakers will actively manipulate these patterns, to great effect. And our innate ability to recognize them, even if on a less than conscious level, is the same reason that machine produced text tends to "sound" terrible. The patterns aren't there. But we can find them in human-generated text and impart them to our machine friends.

In the past few years, research has quickly blossomed around Neural Networks. With widely available open source tools, the power of Neural Networks to find patterns in large datasets quickly transformed the NLP landscape. The Perceptron quickly became the Feed Forward Network (a Multi-Layer Perceptron) which led to the development of new variants: Convolutional Neural Nets and Recurrent Neural Nets, ever more efficient and precise tools to fish patterns out of large datasets.

As we have seen already with *Word2Vec*, neural networks have opened entirely new approaches to NLP. While neural networks' original design purpose was to enable a machine to *learn* to classify input, the field has since grown from just learning classifications (topic analysis, sentiment analysis) to actually being able to generate novel text based on previously unseen input: Translating a new phrase to another language, generating responses to questions not seen before (chatbot, anyone?), and even generating new text based on the style of a particular author.

A complete understanding of the mathematics of the inner workings of a Neural Network is not critical to employing the tools presented in this chapter. However, it does help to have a basic grasp of what is going on inside. If you understand the examples and explanations in Chapter 5 it will improve your intuition about where to use neural networks and the kinds of tweaks to your neural network architecture (like the number of layers or number of neurons) that may help a network work better for your problem. This intuition will help you see how Neural Networks can give depth to our chatbot. Neural networks promise to make our chatbot a better listener and a little less superficially chatty.

7.1 Learning Meaning

The nature of words and their secrets are most tightly correlated to (after their definition, of course) their relation to each other. That relationship can be expressed in many ways, like:

7.1.1 Word Order

The dog chased the cat. The cat chased the dog.

Two statements that don't mean quite the same thing.

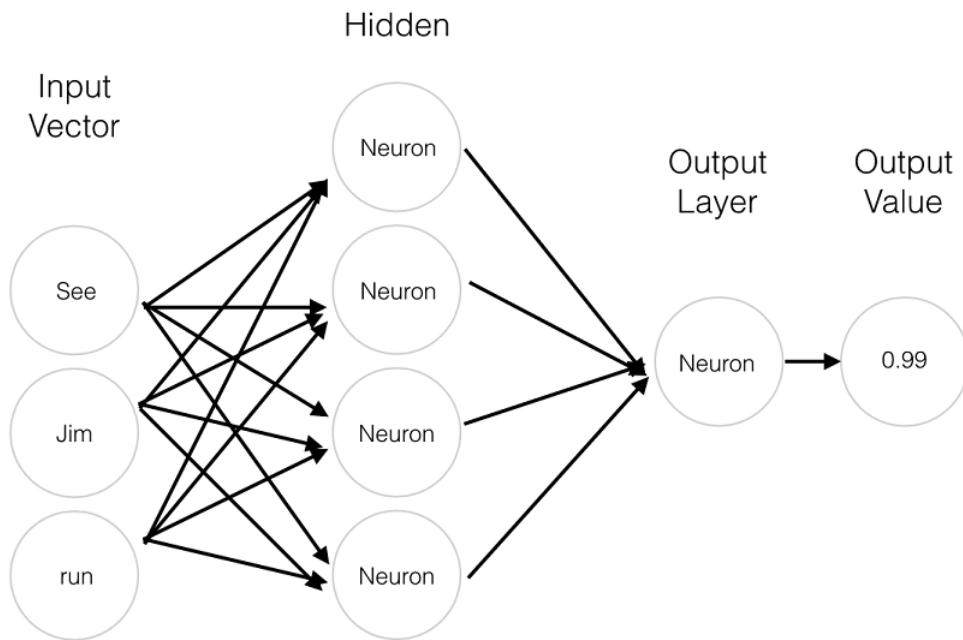
WORD PROXIMITY

The ship's hull, despite years at sea, millions of tons of cargo, and two mid-sea collisions, shone like ne

In this example, the word "shone" refers directly to the word "hull". But they are on far ends of the sentence from each other. It would be nice if there were a way to capture that relationship (spoiler: there is).

These relationships can be explored for patterns (along with patterns in the presence of the words themselves) in two ways: spatially and temporally. You can think of the difference as: in the former we examine the statement as if written on page, we are looking for relationships in the position of words. In the latter we explore it as it comes as if spoken, the words and letters become *time series* data. These are very closely related, but they mark a key difference in how we will deal with them with Neural Network tools.

Basic Feed Forward Networks (Multi-Layer Perceptron) are capable of pulling patterns out of data. However, the patterns they will discover are found by relating weights to the input and there is nothing that captures the relations of the tokens spatially or temporally. But Feed Forward is only the beginning of Neural Network architectures out there. The two most important choices for Natural Language Processing are currently Convolutional Neural Nets and Recurrent Neural Nets and the many flavors of each.



In a Fully-Connected Network, all nodes feed into each node in the neuron in the next layer.
Text input with binary classification.

Figure 7.1 Fully Connected Neural Net

In the image above, three tokens are passed into the net and each is associated with specific weights in the hidden neurons.

TIP

How are we *passing tokens into the net*? We can use various approaches. The two major ones we will use in this chapter are the ones we developed in the previous chapters: one-hot encoding and word vectors. We can either one-hot encode them, a vector that has a zero for every possible vocabulary word we want to consider, with a 1 in the position of the word we are encoding. Or we can use the trained word vectors that we discovered in Chapter 6. Basically, we need the words to be represented as numbers to do math on them.

Now, if we swapped the order of these tokens from "See Jim run" to "run See Jim" and passed that into the network, unsurprisingly a different answer may come out. Remember each input position is associated with a specific weight inside each hidden neuron (x_1 is tied to w_1 , x_2 to w_2 , etc.).

A Feed Forward network may be able to learn specific relationships of tokens such as these, as they appear together in a sample but in different positions. However you can easily see how longer sentences of five, ten, fifty tokens where all of the possible pairs, triplets, etc. in all of the possible positions for each pair, triplet, etc. quickly becomes an intractable problem. Luckily we have other options.

7.2 Toolkit

Python is one of the richest languages for working with Neural Nets. While a lot of the major players (Hi, Google and Facebook!) have moved to lower level languages for the implementation of these very expensive calculations, the extensive resources poured into early models using Python for development have left their mark. Two of the major programs for Neural Network architecture are Theano (deeplearning.net/software/theano/) and Tensorflow (www.tensorflow.org/), both rely heavily on C for their underlying computations but both have very robust Python API's. Facebook put their efforts into a Lua package called Torch, luckily there is now a Python API for that as well in PyTorch (pytorch.org/) Each of these however, for all their power, are heavily abstracted tool sets for building models from scratch. But the Python community is quick to the rescue with libraries to ease the use of these underlying architectures. There are several available (Lasagne (Theano), Skflow (Tensorflow)), but we will be using Keras (keras.io/), for its balance of friendly API and versatility. Keras can use either Tensorflow or Theano as its backend and each has its advantages and weaknesses, but we will use Tensorflow for the examples. We also need the h5py package for saving the internal state of our trained model.

By default, Keras will use Tensorflow as the backend, and the first line output at runtime will remind you which backend you are using for processing. The backend can easily be changed in a config file, with an environment variable, or in your script itself. The documentation in Keras is very thorough and clear, we highly recommend you spend some time there. But a quick overview of what is here: `Sequential()` is a class that is a neural net abstraction that gives us access to the basic API of Keras, specifically the methods `compile` and `fit` that will do the heavy lifting of building the underlying weights and their interconnected relationships (`compile`), calculating the errors in training, and most importantly applying backpropagation (`fit`). `epochs`, `batch_size`, and `optimizer` are all hyper-parameters that will require tuning and in some senses - art.

Unfortunately, there is no one size fits all rule for tuning a neural network, best practices come with experience and intuition. And both of those come with trial and error. There is nothing scary about them, just know that they are there and we will come back to them as they become relevant. Now lets steer this back toward Natural Language Processing via the world of image processing.

Images? Bear with us for a minute, the trick will become clear.

7.3 Convolutional Neural Nets

Convolutional Neural Nets get their name from the concept of sliding (or convolving) a small window over the data sample.

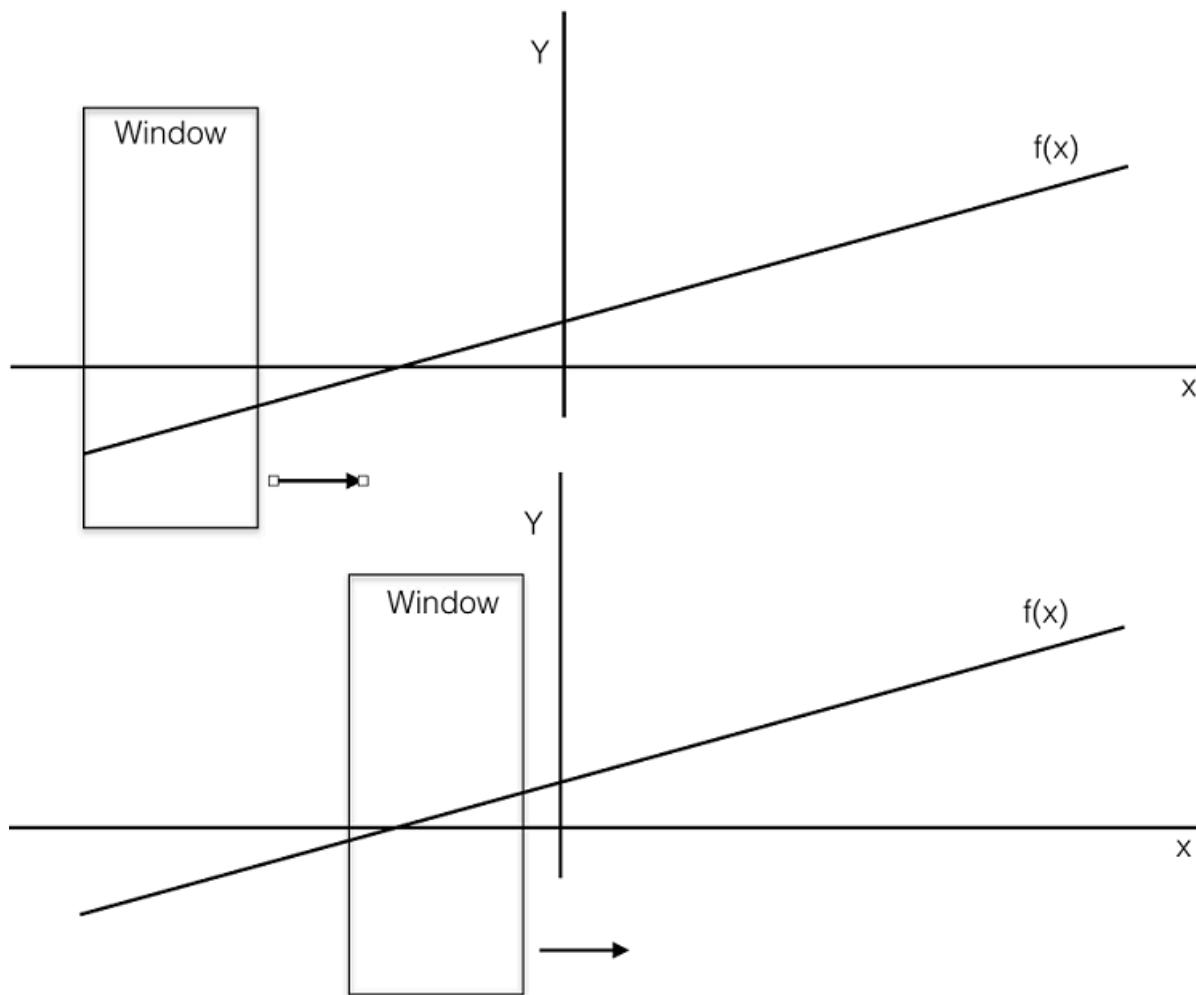


Figure 7.2 Window Convolving Over Function

There are many places in mathematics that convolutions appear and they are usually related to time series data. The higher order concepts related to those use cases aren't important or even relevant for our application in this chapter. The key concept to remember is to visualize that box sliding over a field. We are going to start sliding them over images to get the concept. And then we will start sliding the window over text. But always come back to that mental image of a window sliding over a larger piece of data, and we are just looking at what can be seen through the window.

7.3.1 Building Blocks

Convolutional Neural Nets first came to prominence in image processing and image recognition. Since the net is capable of capturing spatial relationships between data points of each sample the net can suss out whether the image contains a cat or a dog driving a bulldozer.

A Convolutional Net, or Convnet (yeah that extra n in their is hard to say), achieves its magic not by assigning a weight to each of the elements (say each pixel of an image) of each sample, as in a traditional Feed Forward net. Instead it defines a set of *filters* (also known as *kernels*) that move across the image. Our *convolution!*

In image recognition the elements of each data point could be a 1 (on) or 0 (off) for each pixel in a black and white image.



Figure 7.3 Small Telephone Pole Image

Figure 7.4 Small Telephone as Pixel Values

Or it could be the intensity of each pixel in a gray-scale image, or the intensity in each of the color channels of each pixel in a color image.

Each filter we make is going to *convolve* or slide across the input sample (in this case, our pixel values). Let's pause and describe what we mean by sliding. We will not be doing anything in particular while the window is "in motion". You can think of this as a series of snapshots. Look through the window, do some processing, slide the window down a bit, do the processing again.

TIP

This sliding/snapshot routine is precisely what makes Convolutional Neural Nets, highly parallelize-able. Each snapshot for a given data sample can be calculated independently of all the others for that given data sample. No need to wait for the first snapshot to happen before taking the second.

How big are these filters we are talking about? The size is a parameter to be chosen by the model builder and is highly dependent on the content of data. However there are some common starting points. In image based data you will commonly see a window size of three by three (3, 3) pixels. We will get into a little more detail about the window size choice later in the chapter when we get back to actual NLP uses.

7.3.2 Step Size

It is important to notice the distance traveled during the sliding phase is a parameter. And more importantly, it is almost never as large as the filter itself. Each snapshot usually has an overlap with its neighboring snapshot.

The distance each convolution "travels" is known as the *stride* and is typically set to 1. Only moving one pixel (or anything less than the width of the filter) will create overlap in the various inputs to the filter from one position to the next. A larger stride that has no overlap between filter applications will lose the "blurring" effect of one pixel (or in our case: word) relating its neighbors.

This overlap has some very interesting properties. That will become apparent later as we see how the filters change over time.

7.3.3 Filter Composition

Okay, so far we have been describing windows sliding over data, looking at the data through the window, but we have said nothing about what we do with the data we see.

Filters are composed of two parts:

- A set of weights (exactly like the weights in the Feed Forward Neurons from chapter 5)
- An activation function

As we said earlier they are typically 3x3 (but often other size and shapes)

TIP

While these are very similar to the Feed Forward neurons from Chapter 5 the main critical difference is that each filter's weights are fixed for the entire sweep of the input sample. There will be many filters in a Convolutional Neural Net and they will all be different, but each individual filter will be fixed for each of its snapshots of the image.

As each filter slides over the image, one *stride* at a time, it will pause and take "snapshot" of the pixels it is currently covering. The values of those pixels are then multiplied by the weight associated with that position in the filter.

So say we are using a 3x3 filter. We start in the upper left corner and snapshot the first pixel (0, 0) by the first weight (0, 0). The second pixel (0, 1) by weight (0, 1), etc.

The products of pixel and weight (at that position) are then summed up and passed into the activation function (most often this is ReLU and we will come back to that in a moment).

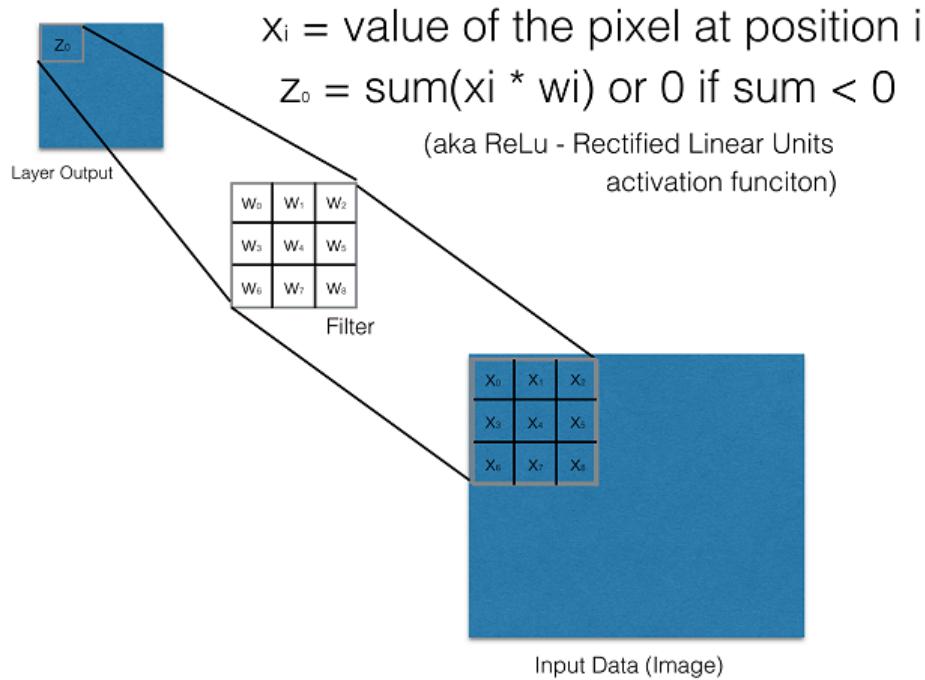


Figure 7.5 Convolutional Neural Net Step

The output of that activation function is recorded as a positional value in an output "image". The filter slides one stride-width and takes the next snapshot and puts the output value next to the output of the first.

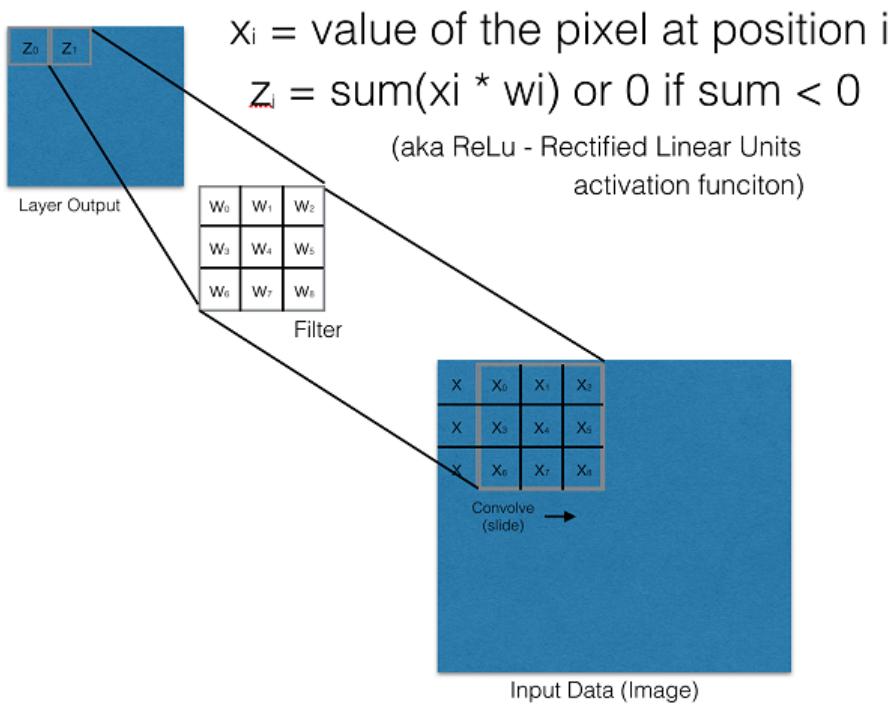


Figure 7.6 Convolution

There will be several of these filters in a layer and as they each convolve over the entire image, they each create a new 'image'. A 'filtered' image if you will. Say we have n filters. After this process. We would have n new, 'filtered' images for each filter we defined.

We'll get back to what we do with these n new images in a moment.

7.3.4 Padding

Something funny happens at the edges, however. If we start a 3x3 filter in the upper left corner of an input image and stride one pixel at a time across and stop when the rightmost edge of the filter reaches the rightmost edge of the input, the output "image" will be 2 pixels narrower than the source input.

There are multiple strategies for dealing with this and Keras has tools to help. The first is to just ignore the fact that the output is slightly smaller. The Keras argument for this is "padding=valid". If this is the case, you just have to be careful and take note of the new dimensions as you pass the data into the next layer. The downfall of this strategy is the data in the edge of the original input is under-sampled as the interior data points are passed into each filter multiple times, from the overlapped filter positions. On a large image this may not be an issue, but as we soon bring this concept to bear on a Tweet, for example, under-sampling a word at the beginning of a 10 word dataset could drastically change the outcome.

The next strategy is actually known as *padding*. Padding consists of adding enough data to the outer edges of the input so that the first real data point is treated just as the innermost data points are. The downfall of this strategy is we are adding potentially unrelated data to the input, which in itself can skew the outcome. We don't care to find patterns in fake data that we generated after all. There are several ways to pad the input to try and minimize the ill effects though.

```
from keras.models import Sequential
from keras.layers import Conv1D

model = Sequential()
model.add(Conv1D(filters=16,
                 kernel_size=3,
                 padding='same',
                 activation='relu',
                 strides=1,
                 input_shape=(100, 300)))
```

①

②

- ① —'same' or 'valid' are the options.
- ② —input_shape is still the shape of your unmodified input. The padding will happen under the hood

More on the implementation details in a moment. Just be aware of these troublesome bits, and know a good deal of what could be rather annoying data wrangling has been abstracted away for you nicely by the tools we will be using.

There are other strategies where the pre-processor attempts to guess at what the padding should be mimicking the data points that are already on the edge, but we won't have use for that in NLP applications, for it is fraught with its own peril.

CONVOLUTIONAL PIPELINE

So we have n filters and n new images now. What do we do with that? This, like most applications of neural networks, starts from the same place: a labeled dataset. And likewise we have a similar goal. To predict a label given a novel image. The simplest next step is to take each of those filtered images and string them out as input to a Feed Forward layer and then proceed as normal from chapter 5.

TIP

It is possible however to pass these filtered images into a second Convolutional layer with its own set of filters. In practice this is the most common architecture and we will brush on that later. It turns out the multiple layers of convolutions leads to a path to learning layers of abstractions: first edges, then shapes/colors, and eventually concepts!

But however many layers (convolutional or otherwise) once we have a final output (and hence an error) we can backpropagate.

Because the activation function was differentiable we can backpropagate as normal and update the weights of the individual filters themselves. And through this, the network learns what kind of filters it needs to get the right output for a given input.

You can think of this as the network learning to detect and extract information for the later layers to act on more easily.

7.3.5 Learning

The filters themselves, as in any Neural Network, start out with weights that are initialized to random values near zero. So how is the output "image" going to be anything more than noise? At first, in the first few iterations of training, it will be just that, noise.

But the classifier we are building will have some amount of error from the expected label for each input, and that input can be backpropagated through the activation function to the values of the filters themselves. To backpropagate the error, we have to take the derivative of the error with respect to the weight that fed it.

And as the convolution layer comes earlier in the net, it is specifically the derivative of the gradient from the layer above with respect to the weight that fed it. This calculation is similar to normal backpropagation as the weight generated output in many positions for a given training sample.

The specific derivations of the gradient with respect to the weights of a convolutional filter are beyond the scope of this book, but a shorthand way of thinking about it is for a given weight in a given filter, the gradient is the sum of the normal gradients that were created for each individual position in the convolution during the forward pass. Represented mathematically in Equation 1.

$$\frac{\partial E}{\partial w_{ab}} = \sum_{i=0}^m \sum_{j=0}^n \frac{\partial E}{\partial x_{ij}} \frac{\partial x_{ij}}{\partial w_{ab}}$$

Figure 7.7 Summation of Gradients for a Filter Weight

Slightly more going on, but basically the same concept as a regular Feed Forward net, were we are figuring out how much each particular weight contributed to the overall error of the system. Then we decide how best to correct that toward a weight that will cause less error in the future training examples. None of these details are vital for the understanding of the the use of Convolutional Neural Nets in Natural Language Processing, we do offer them though in an effort to build intuition on how to tweak and grow the tools we present later in the chapter.

7.4 Narrow Windows Indeed

Yeah, yeah, okay, images. But where talking about language here, remember? Let's see some words to train on. It turns out we can use Convolutional Neural Networks for Natural Language Processing by using *Word Vectors* that we learned about in Chapter 6 (also know as *word embeddings*) instead of an image's pixel values as the input to our network.

As relative vertical relations between words would be arbitrary, depending on the page width, there is no relevant information in the patterns that may emerge there. There is relevant information is in the relative "horizontal" positions though.

TIP

This is speaking in terms of Western languages. The same concepts hold true for languages that are read top to bottom *before* reading right or left, such as Japanese. But in those cases we would focus on "vertical" relationships rather than "horizontal".

This means we only want to focus on the relationships of tokens in one spatial dimension. Instead of 2 dimensional filter that we would convolve over a 2 dimensional input (a picture) we will convolve 1 dimensional filters over a 1 dimensional input, such as a sentence.

It also means that our filter *shape* will be 1 dimensional instead of 2.

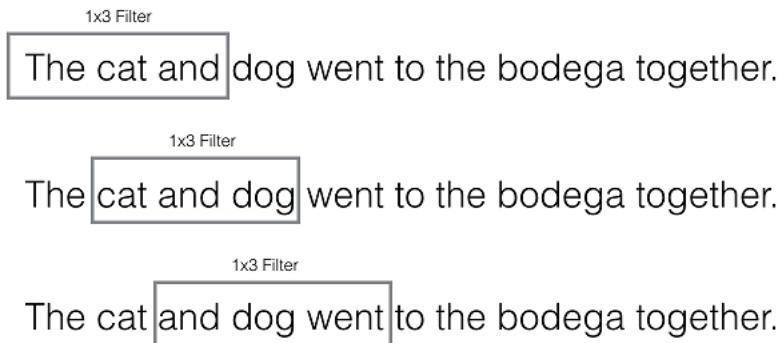


Figure 7.8 1 Dimensional Convolution

The "second" dimension in the image is assumed to be the full length of the word vector. We will only be concerned with the "width" of the filter. In the image above the filter is 3 tokens wide. Aha! Notice each word token (or later character token) is a "pixel" in our sentence "image".

TIP

The term 1-dimensional filter can be a little misleading as we get to word embeddings. The vector representation of the word itself extends "downward" as in the image below, but the filter covers the whole length of that dimension in one go. The dimension we are referring to when we say 1-dimensional convolution, is the "width" of the phrase; the dimension we are traveling across. In a 2-dimensional convolution, of an image say, we would scan the input from side to side and top to bottom, hence the 2 dimensional name. Here we only slide in one dimension, left to right.

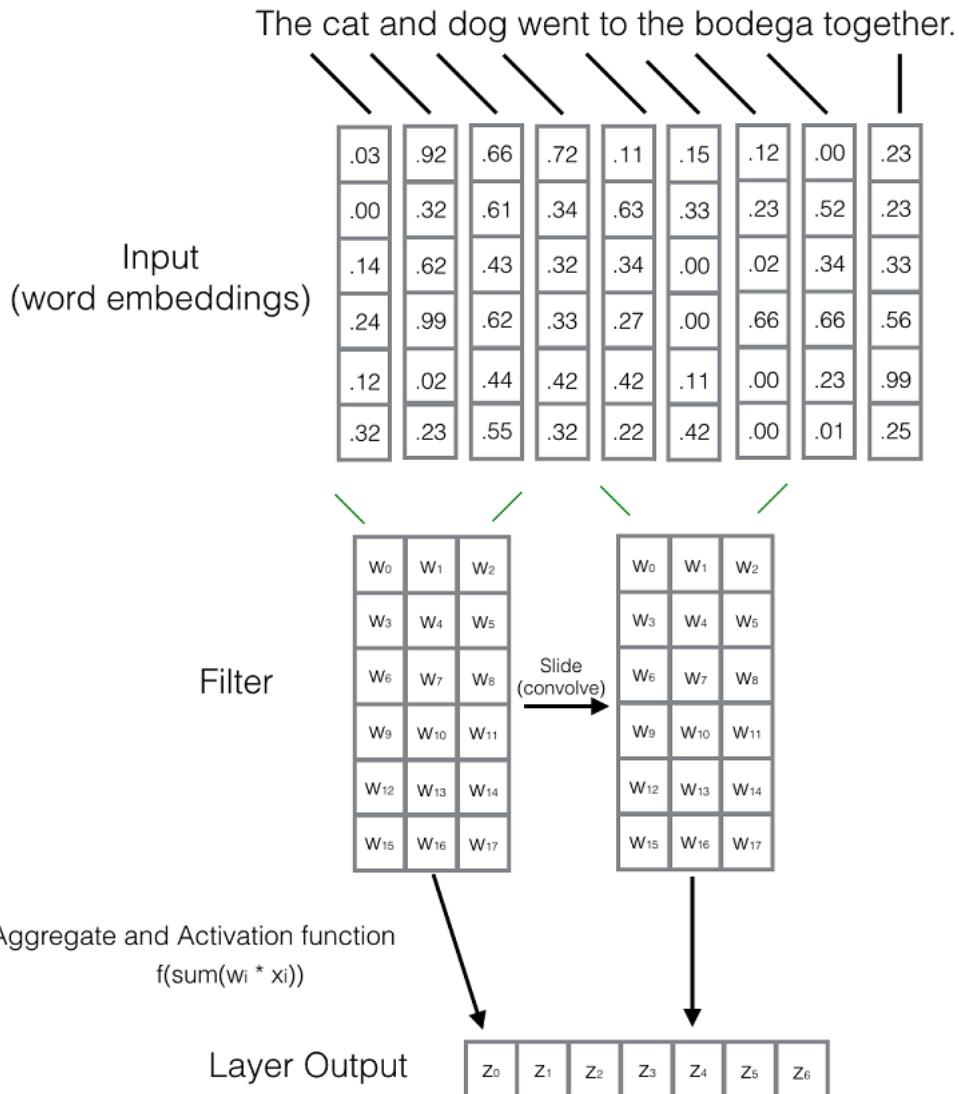


Figure 7.9 1 Dimensional Convolution with Embeddings

As mentioned earlier, the term convolution is actually a bit of shorthand. But it bears repeating: the actual sliding has no effect on the model. It is the data at multiple positions that dictates what is going on. The order the "snapshots" are calculated in isn't important as long as the output is reconstructed in the same way the windows onto the input were oriented.

The values of the weights in the filters themselves are unchanged for a given input sample, during the forward pass. Which means we can take a given filter and take all of its "snapshots" in parallel and compose the output "image" all at once. This is the Convolutional Neural Network's secret to speed.

This speed on top of its robustness to noise in the data is why research keep coming back to this tool.

7.4.1 Implementation in Keras: Prepping the Data

Let's take a look at this in Python with the example Convolutional Neural Network classifier provided in the Keras documentation. They have crafted a 1-dimensional Convolutional Net to examine the IMDB movie review dataset.

Each data point is pre-labeled with either a 0 (negative sentiment) and 1 (positive sentiment). We are going to swap out their example IMDB movie review dataset for one in raw text, so we can get our hands dirty with the pre-processing of the text as well. And then we will see if we can use this trained network to classify text it has never seen before.

```
import numpy as np
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.layers import Conv1D, GlobalMaxPooling1D
```

- 1
- 2
- 3
- 4
- 5

- 1 Keras takes care of most of this but it likes to see Numpy arrays
- 2 A helper module to handle padding input
- 3 The base keras Neural Network model
- 4 The layer objects we will pile into the model
- 5 Our convolution layer, and pooling

First download the original dataset from ai.stanford.edu/~amaas/data/sentiment. A dataset compiled for the 2011 paper Learning Word Vectors for Sentiment Analysis¹⁴². Once downloaded, unzip it to a convenient directory and look inside. We are just going to use the "train" folder, but there are other toys in there, so feel free to look around.

Footnote 142 Maas, Andrew L. and Daly, Raymond E. and Pham, Peter T. and Huang, Dan and Ng, Andrew Y. and Potts, Christopher, Learning Word Vectors for Sentiment Analysis, Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, June 2011, Association for Computational Linguistics

The reviews in the train folder are broken up into text files in either the pos or neg folders. So we will first need to read those in Python with their appropriate label and then shuffle the deck so the samples aren't all positive and then all negative. Training with the sorted labels will skew training toward whatever comes last, especially when certain hyper-parameters like *momentum* are used.

```

import glob
import os

from random import shuffle

def pre_process_data(filepath):
    """
    This is dependent on your training data source but we will try to generalize it as
    best as possible.
    """
    positive_path = os.path.join(filepath, 'pos')
    negative_path = os.path.join(filepath, 'neg')

    pos_label = 1
    neg_label = 0

    dataset = []

    for filename in glob.glob(os.path.join(positive_path, '*.txt')):
        with open(filename, 'r') as f:
            dataset.append((pos_label, f.read()))

    for filename in glob.glob(os.path.join(negative_path, '*.txt')):
        with open(filename, 'r') as f:
            dataset.append((neg_label, f.read()))

    shuffle(dataset)

    return dataset

dataset = pre_process_data('<path to your downloaded file>/aclimdb/train')
print(dataset[0])

```

So the first example should look something like the example below. Your's will differ depending on how they were shuffled but that is fine. The first element in the tuple is the *target* value for sentiment. 1 for positive sentiment, 0 for negative.

```
(1, 'I, as a teenager really enjoyed this movie! Mary Kate and Ashley worked great
together and everyone seemed so at ease. I thought the movie plot was very good and
hope everyone else enjoys it to! Be sure and rent it!! Also they had some great soccer
scenes for all those soccer players! :)')
```

The next step is to tokenize and vectorize the data. We will use the Google News pre-trained word2vec vectors, so download those via the `nlpia` package or directly from Google¹⁴³.

Footnote 143 drive.google.com/file/d/0B7XkCwpI5KDYNlNUlSS21pQmM/edit?usp=sharing

We will use gensim to unpack the vectors, just like we did in Chapter 6. You can experiment with the `limit` argument to the `load_word2vec_format` method, a higher number will get you more vectors to play with, but memory quickly becomes an issue and return on investment drops quickly in really high values for *limit*.

Let's write a helper function to tokenize the data and then create a list of the vectors for

those tokens to use as our actual data to feed the model.

```
from nltk.tokenize import TreebankWordTokenizer
from gensim.models.keyedvectors import KeyedVectors
word_vectors = KeyedVectors.load_word2vec_format('<path to your download
word2vec file>/GoogleNews-vectors-negative300.bin.gz', binary=True, limit=200000)

def tokenize_and_vectorize(dataset):
    tokenizer = TreebankWordTokenizer()
    vectorized_data = []
    expected = []
    for sample in dataset:
        tokens = tokenizer.tokenize(sample[1])
        sample_vecs = []
        for token in tokens:
            try:
                sample_vecs.append(word_vectors[token])
            except KeyError:
                pass # No matching token in the Google w2v vocab
        vectorized_data.append(sample_vecs)

    return vectorized_data
```

Note we are throwing away information here. The GoogleNews word2vec vocabulary includes some stopwords but not all of them. So a lot of common words like 'a' will be thrown out in our function. This is not ideal by any stretch, but this will give you a baseline for how well Convolutional Neural Nets (CNNs) can perform even on lossy data. To get around this you can train your word2vec models separately and make sure you have better vector coverage. The data also has a lot of html tags like
. Those are things we do want to exclude as they aren't usually relevant to the sentiment of text.

We also need to collect the target values, 0 for a negative review, 1 for a positive review, in the same order as the training samples.

```
def collect_expected(dataset):
    """ Peel off the target values from the dataset """
    expected = []
    for sample in dataset:
        expected.append(sample[0])
    return expected
```

And then we simply pass our data into those functions:

```
vectorized_data = tokenize_and_vectorize(dataset)
expected = collect_expected(dataset)
```

We next split the prepared data into a training set and a test set. We are just going to split our imported dataset 80/20, but you can feel free to use the test folder from the original

download for this. That will provide more training data which is almost always better in training your models.

The next code block buckets the data into the training set, *x_train*, that we will show the network along with "correct" answers, *y_train*, and a testing dataset, *x_test* that we hold back, along with its answers, *y_test*. We can then let the network make a "guess" about samples from the test set and we can validate that it is learning a something that generalizes outside of the training data. *y_train* and *y_test* are the associated "correct" answers for each example in the respective sets *x_train* and *x_test*.

```
split_point = int(len(vectorized_data)*.8)

x_train = vectorized_data[:split_point]
y_train_ = expected[:split_point]
x_test = vectorized_data[split_point:]
y_test = expected[split_point:]
```

The next block of code sets most of the hyper-parameters for the net. The *maxlen* variable holds the maximum length of review we will consider. As each input to a convolutional neural net must be equal in dimension, we truncate any sample that is longer than 400 tokens and pad the shorter samples out to 400 tokens with Null or 0, actual "PAD" tokens are commonly used to represent this when showing the original text. Again this introduces data into the system that wasn't previously in the system. The network itself can learn that pattern as well though, so that PAD == "ignore me" becomes part of the network's structure, so it's not the end of the world.

Note of caution: This padding is not the same as introduced earlier. Here we are padding out the input to be of consistent size. We will separately need to decide the issue of padding the beginning and end of each training sample based on whether or not we want the output to be of similar size and the end tokens to be treated the same as the interior ones, or whether we don't mind the out first/last tokens being treated differently.

```
maxlen = 400
batch_size = 32
embedding_dims = 300
filters = 250
kernel_size = 3
hidden_dims = 250
epochs = 2
```

- ➊ How many samples to show the net before backpropagating the error and updating

the weights

- ② Length of the token vectors we will create for passing into the Convnet
- ③ Number of filters we will train
- ④ The width of the filters, actual filters will each be a matrix of weights of size: embedding_dims x kernel_size or 50 x 3 in our case
- ⑤ Number of neurons in the plain Feed Forward net at the end of the chain
- ⑥ Number of times we will pass the entire training dataset through the network

TIP

The `kernel_size` (filter size or window size) is a scalar value here as opposed to the 2-dimensional type filters we had with images. So our filter will look at the word vectors for 3 tokens at a time. It is helpful to think of the filter sizes, in the FIRST LAYER ONLY, as looking at n-grams of the text. In this case we are looking at 3-grams of our input text. But this could easily by 5 or 7 or more. The choice is data and task dependent, so experiment freely with this parameter for your models.

Keras has a preprocessing helper method `pad_sequences` that in theory could be used to pad our input data, but unfortunately it only works sequences of scalars, and we have sequences of vectors. So let's write a helper function of our own to do that for us.

```
# Must manually pad/truncate

def pad_trunc(data, maxlen):
    """ For a given dataset pad with zero vectors or truncate to maxlen """
    new_data = []

    # Create a vector of 0's the length of our word vectors
    zero_vector = []
    for _ in range(len(data[0][0])):
        zero_vector.append(0.0)

    for sample in data:
        if len(sample) > maxlen:
            temp = sample[:maxlen]
        elif len(sample) < maxlen:
            temp = sample
            # Append the appropriate number 0 vectors to the list
            additional_elems = maxlen - len(sample)
            for _ in range(additional_elems):
                temp.append(zero_vector)
        else:
            temp = sample

        # Finally tack the augmented data onto our new list
        new_data.append(temp)

    return new_data
```

Then we pass in our train and test data into the pad/truncator. We then need to convert it Numpy arrays to make Keras happy. Finally we have a tensor in the shape (number of

samples, sequence length, word vector length)

```
x_train = pad_trunc(x_train, maxlen)
x_test = pad_trunc(x_test, maxlen)

x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims))
y_train = np.array(y_train)
x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
y_test = np.array(y_test)
```

Phew, finally, we are ready to build a neural network.

7.4.2 Convolutional Neural Network Architecture

We start with the base Neural Network model class *Sequential*. As with the Feed Forward network from Chapter 5, Sequential is one of the base classes for Neural Networks in Keras. From here we can start to layer on the magic.

The first piece we add is a Convolutional Layer. In this case we are assuming it is okay that the output is of smaller dimension than the input and setting the padding to "valid". Each filter will start its pass with its left-most edge at the start of the sentence and stop with its right-most edge on the last token.

Each shift (stride) in the convolution will be 1 token. The kernel (window width) we already set to 3 tokens above. And we are using the 'relu' activation function. So at each step, we will multiply the weight in the filter times the value in the 3 tokens it is looking at (element-wise), sum up those answers and pass them through as is, if they are greater than 0, else we pass 0. That last pass through of positive values and 0's is the *Rectified Linear Units* activation function or *ReLU*.

```
print('Build model...')

model = Sequential()

# we add a Convolution1D, which will learn filters
# word group filters of size filter_length:
model.add(Conv1D(filters,
                 kernel_size,
                 padding='valid',
                 activation='relu',
                 strides=1,
                 input_shape=(maxlen, embedding_dims)))
```

1

2

- ① The standard model definition pattern for Keras. There is a parallel model definition referred to as the Functional API, and we will see that in a later chapter.
- ② Adding 1 Convolutional Layer to the network. There are many more keyword arguments, but we are just using their defaults for now. But definitely check out the

documentation.

7.4.3 Pooling

We've started a neural network, so ... everyone into the pool! Pooling is the Convolutional Neural Network's path to dimensionality reduction. In some ways we are speeding up the process by allowing for parallelization of the computation. But you may notice we make a new "version" of the data sample, a filtered one, for each filter we define. In the example above that would be 250 filtered versions come out of the first layer. Pooling will mitigate that somewhat, but it also has another striking property.

The key idea is we are going to evenly divide the output of the each filter into a subsections. Then for each of those subsections, we will select or compute a representative value. And then we set the original output of the aside and use the collections of representative values as the input to the next layers.

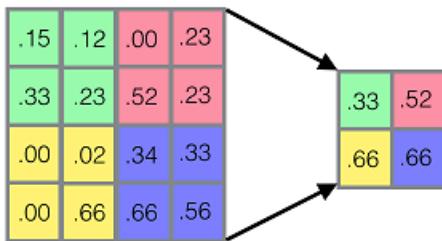
But wait. Isn't throwing away data terrible. Usually, this would not be the best course of action. But it turns out, this is a path toward learning higher order representations of the source data. The filters are being trained to find patterns. The patterns are revealed in *relationships* between words and their neighbors! Just the kind of subtle information we set out to find.

In image processing, the first layers will tend to learn to be edge detectors, places where pixel densities rapidly shift from one side to the other. While later layers learn concepts like shape and texture, layers after that may learn "content" or "meaning". Similar processes will happen with text.

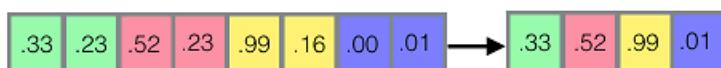
TIP

In an image processor the pooling region would usually be a 2x2 pixel window (and these do not overlap, like our filters do), but in our 1D convolution they would be a 1D window 1x2 or 1x3 say.

2D Max Pooling



1D Max Pooling



1D Global Max Pooling

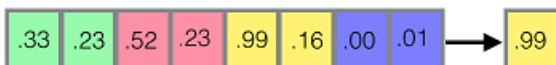


Figure 7.10 Pooling Layers

There are two choices for pooling *Average* and *Max*. *Average* is the more intuitive of the two in that by taking the average of the subset of values you would in theory retain the most data. *Max Pooling* however has an interesting property, in that by taking the largest activation value for the given region, the network sees that subsections most prominent feature. The network has a path toward learning what it should look at, regardless of exact pixel level position!

In addition to dimensionality reduction and the computational savings that come with it, we gain something else special, *location invariance*. If an element of the the original input is jostled slightly in position in a similar but distinct input sample, the *Max Pooling* layer will still output something similar. This is a huge boon in image recognition world, and it serves a very similar purpose in Natural Language Processing.

In this simple example from Keras, we are using the `GlobalMaxPooling1D` layer. So instead, of taking the max of a small subsection of each filter's output, we are taking the max of the entire output for that filter. This results in a large amount of information loss. But even tossing aside all of that good information, our toy model will not be deterred.

```
# we use max pooling:  
model.add(GlobalMaxPooling1D())
```

1

- ➊ The parallel options described above are MaxPooling1D(n) or AvgPooling1D(n) where n is the size of the area to pool. In those options will n will default to 2 if not provided.

Okay, outta the pool, grab a towel. Let's recap the path so far:

- For each input example we applied a filter (weights and activation function)
- Convolved across the length of the input, which would output a 1d vector slightly smaller than the original input (1x398 which is input with the filter starting left-aligned and finishing right-aligned) for each filter
- For each filter output (there are 250 of them, remember) we took the single maximum value from that 1d vector
- At this point we have a single vector (per input example) that is (1x250 the number of filters)

Now for each input sample we have a 1D vector that the network thinks is a good representation of that input sample. This is a *semantic* representation of the input! A very crude one to be sure. And it will only be semantic in the context of the training target which is sentiment. There will not be an encoding of the content of the movie being reviewed, say, just an encoding of its sentiment.

Of course we haven't done any training yet, so actually it is a garbage pile of numbers but we'll get back to that later. But this is an important point to stop and really understand what is going on, for once the network is trained, this *semantic* representation (we like to think of it as a "thought vector") can really be useful. Much like the various ways we embedded words into vectors so we can math on them, we now have something that represents whole groupings of words.

Enough of the excitement, back to the hard work of training. We have a goal to work toward and that is our labels for sentiment. So we take our current vector and pass it into a standard Feed Forward network, in Keras that is a *Dense* layer. The current setup has the same number of elements in our semantic vector and the number of nodes in the dense layer, but that is just coincidence. Each of the 250 (`hidden_dims`) neurons in the Dense layer each have 250 weights for the input from the pooling layer. We temper that with a dropout layer to prevent overfitting.

7.4.4 Dropout

Dropout (represented as layer by Keras, as below) is a special technique developed to prevent overfitting in neural networks. It is not specific to Natural Language Processing but it does work just as well here.

The idea is that on each training pass, if you "turn off" a certain percentage of the input

going to the next layer, randomly chosen on each pass, the model will be less likely to learn the specifics of the training set, "over-fitting", and instead learn more nuanced representations of the patterns in the data and thereby be able to generalize and make accurate predictions when it sees completely novel data.

This is achieved on a lower level by assuming the output coming into the Dropout layer (the output from the layer below) is 0 for that particular pass. The reason this works is, on that pass, the contribution to the overall error of each of the neuron's weights that would receive the dropouts zero input is also effectively 0. Therefore those weights will not get updated on the backpropagation pass. The network is then forced to rely on relationships amongst varying weight sets to achieve its goals (hopefully they won't hold this tough love against us).

TIP

Don't worry too much about this point, but it is worth noting that Keras will do some magic under the hood for Dropout layers. Since we are randomly turning off a percentage of the inputs on each forward pass of the training. And since we don't turn off any during inference of the data after training. The strength of the signal going into layers after a Dropout layer would be significantly higher in during the non-training inference stage.

Keras mitigates this in the training phase by proportionally boosting all inputs that are not turned off, so the aggregate signal that goes into the next layer is of the same magnitude as it will be during inference stages.

The parameter passed into the Dropout layer in Keras is the percentage of the inputs to randomly turn off. In this example only 80% of the embedding data, randomly chosen for each training sample, will pass into the next layer as it is. The rest will go in as 0's. 20% dropout is common, but dropout up to 50% can have satisfying results, so here we have one more hyper-parameter to play with.

And then we use the Rectified Linear Units activation (`relu`) on the back end of each neuron.

```
# We add a vanilla hidden layer:
model.add(Dense(hidden_dims))
model.add(Dropout(0.2))
model.add(Activation('relu'))
```

7.4.5 The Cherry on the Sundae

The last layer or output layer, is the actual classifier, so here we have neuron that fires based on the sigmoid activation function. Sigmoid gives us a value between 0 and 1. During validation Keras will consider anything below 0.5 to be classified as 0 and anything above 0.5 to be a 1. But in terms of *loss* calculated it will be the target minus the actual value provided by the sigmoid ($y - f(x)$).

```
# We project onto a single unit output layer, and squash it with a sigmoid:  
model.add(Dense(1))  
model.add(Activation('sigmoid'))
```

That is a Convolutional Neural Network model fully defined in Keras. Nothing left but to compile it and train it.

```
model.compile(loss='binary_crossentropy',  
              optimizer='adam',  
              metrics=['accuracy'])
```

The loss function is what the network will try to minimize. Here we use 'binary_crossentropy'. At the time of writing there are 13 available loss functions defined in Keras, and you have the option to define your own. We won't go into the use cases for each of those, but the two workhorses to know about are: binary_crossentropy and categorical_crossentropy.

They are both very similar in their mathematical definitions and in many ways you can think of binary_crossentropy as a special case of categorical_crossentropy. The important thing to know is when to use which. Here we have one output neuron that is either on or off. So we will use binary_crossentropy.

Categorical is commonly used when you are predicting one of many classes. In those cases, your target will be an n-dimensional vector, one-hot encoded, a position for each of your n classes. The last layer in your network in this case would be:

```
model.add(Dense(num_classes))  
model.add(Activation('softmax'))
```

①

① Where num_class is ... well, you get the picture.

In this case, target minus output ($y - f(x)$) would be an n-dimensional vector subtracted from an n-dimensional vector. And categorical_crossentropy would try to minimize that

difference.

But back to our binary classification.

OPTIMIZATION

The parameter *optimizer* is any of a list of strategies to optimize the network during training, such as Stochastic Gradient Descent, Adam, RMSProp. The optimizers themselves are each different approaches to minimizing the loss function in a neural network, the math behind the details of each is beyond the scope up this book, but be aware of them and try different ones for your particular problem. While many may converge for a given problem, some may not, and all of them will do so at different paces.

Their magic comes from dynamically altering the parameters of the training, specifically the *learning rate*, based on the current state of the training. For example the starting learning rate (Remember: *alpha* is the learning rate applied to the weight updates we saw in Chapter 5) may decay over time. Or some methods may apply *momentum* and increase the learning rate if the last movement of the weights in that particular direction was successful at decreasing the loss.

Each optimizer itself has a handful of hyper-parameters like learning rate. Keras has generally accepted defaults for these values, so you shouldn't have to worry about them too much at first.

FIT

```
model.fit(x_train, y_train,
          batch_size=batch_size,      ①
          epochs=epochs,             ②
          validation_data=(x_test, y_test))
```

- ① The number of data samples processed before the backpropagation updates the weights. The cumulative error for the n samples in the batch is applied at once.
- ② The number of times the training will run through the entire training data set, before stopping.

Where *compile* builds the model, *fit* is where the magic happens. All of the inputs times the weights, all of the activation functions, all the backpropagation is all kicked off by this one statement. Depending on your hardware, the size of your model, and the size of your data this can take anywhere from a few seconds to a few months. Using a GPU can

greatly reduce the training time in most cases and if you have access to one, by all means use it. There are a few extra steps to pass environment variables to Keras to direct it to use the GPU, but our model is small enough we can run it on most modern CPUs in a reasonable amount of time.

7.4.6 Let's Get to Learning (Training)

One last step, before we hit run. We would like to save the model state after training. Since we aren't going to hold the model in memory for now, we can grab its structure in a json file and save the trained weights in another file for later re-instantiation.

```
model_structure = model.to_json() ①
with open("cnn_model.json", "w") as json_file:
    json_file.write(model_structure)

# After the model is trained
model.save_weights("cnn_weights.h5")
```

- ① Note that this does not save the weights of the network. Only the structure.

Now our trained model will be persisted on disk, should it converge we won't have to train it again.

Keras also provides some amazingly useful callbacks during the training phase that are passed into the fit method as keyword arguments such as, *checkpointing* which will iteratively save the model only when the accuracy or loss has improved, or *EarlyStopping* that will stop the training phase early if the model is no longer improving based on a metric you provide. And probably most exciting, they have implemented a Tensorboard callback. Tensorboard only works with Tensorflow as a backend, but it provides an amazing level of introspection into your models and can be indispensable when troubleshooting and fine tuning. Let's get to learning! Running the *compile* and *fit* steps above should lead to:

```
Using Tensorflow backend.
Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
x_train shape: (25000, 400)
x_test shape: (25000, 400)
Build model...
Train on 20000 samples, validate on 5000 samples
Epoch 1/2 [=====] - 417s - loss: 0.3756 - acc: 0.8248 - val_loss: 0.3531 - val_acc: 0.8390
Epoch 2/2 [=====] - 330s - loss: 0.2409 - acc: 0.9018 - val_loss: 0.2767 - val_acc: 0.8840
```

Your final loss and accuracies may vary a bit and that is side effect of the random initial weights chosen for all of the neurons. You can overcome this by passing a seed into the randomizer. That will force the same values to be chosen for the "random" weights on each run. This can be helpful in debugging and tuning your model. Just keep in mind, the starting point can itself force the model into a *local minimum* or even prevent the model from converging, so trying a few different seeds is recommended.

To set the seed add these two lines above your model definition. The integer passed in as the argument to seed is unimportant, but as long as it is consistent the model will initialize its weights to small values in the same way.

```
import numpy as np
np.random.seed(1337)
```

We have not seen definitive signs of *over-fitting* as the accuracy improved for both the training and validation sets. So we could let the model run for another epoch or two and see if we could improve more without over-fitting. A Keras model can continue the training from this point if it is still in memory, or has been reloaded from a save file. Just call the *fit* method again (change the sample data or not) and the training will resume from that last state.

TIP

Overfitting will be apparent when the loss continues to drop for the training run, but the val_loss at the end of each epoch starts to climb compared to the previous epoch. Finding that happy medium where the validation loss curve starts to bend back up is a major key to creating a good model.

Great. Done. Now, what did we just do?

The model was described and then compiled into an initial untrained state. We then called *fit* to actually learn the weights of the filters and the Feed Forward fully connected network at the end as well as the weights of each of the 250 individual filters by backpropagating the error encountered at each example all the way back down the chain.

The progress meter reported *loss* which we specified as "binary_crossentropy". So for each batch Keras is reporting a metric of how far we away from the label we provided for that sample. The accuracy is a report of "percentage correct guesses". This metric is fun to watch but certainly can be misleading, especially if you have a lopsided dataset. Imagine you have 100 examples, 99 of them are positive examples and only one of them should be predicted as negative. If you predict all 100 as positive without even looking at

the data you will still be 99% accurate. But this isn't helpful in generalizing. The val_loss and val_acc are the same metrics on the test dataset provided in:

```
validation_data=(x_test, y_test)
```

The validation samples are never shown to the network for training, only passed in to see what the model predicts for them, and then reported on against the metrics. Backpropagation doesn't happen for these samples. This helps keep track of how well the model will generalize to novel, real-world data.

We've trained a model. The magic is done. The box has told us we it figured everything out. We believe it. So what? Let's get some use out of our work.

7.4.7 Using the Model in a Pipeline

Once we have a trained model we can then pass in a novel sample and see what the network thinks. This could be an incoming chat message or tweet to your bot or as we're going to do on a made up example.

First, reinstate your trained model, if it is no longer in memory.

```
from keras.models import model_from_json
with open("cnn_model.json", "r") as json_file:
    json_string = json_file.read()
model = model_from_json(json_string)

model.load_weights('cnn_weights.h5')
```

Let's make up a sentence with an obvious negative sentiment and see what the network has to say about it.

```
sample_1 = "I'm hate that the dismal weather that had me down for so long, when will it
break! Ugh, when does happiness return? The sun is blinding and the puffy
clouds are too thin. I can't wait for the weekend."
```

With the model pre-trained, testing a new sample is very quick. There are still thousands and thousands of calculations to do, but for each sample we only need one forward pass and no backpropagation to get a result.

```
vec_list = tokenize_and_vectorize([(1, sample_1)]) ①
test_vec_list = pad_trunc(vec_list, maxlen) ②
```

```
test_vec = np.reshape(test_vec_list, (len(test_vec_list), maxlen, embedding_dims))
model.predict(test_vec)
```

- ➊ We pass a dummy value in the first element of the tuple just because our helper expects it from the way processed the initial data. That value won't ever see the network, so it can be whatever.
- ➋ Tokenize returns a list of the data (length 1 here)

The Keras *predict* method will give you the raw output of the final layer of the net. In this case we have one neuron and as the last layer is a sigmoid it will output something between 0 and 1.

```
array([[ 0.12459087]], dtype=float32)
```

The Keras *predict_classes* method will give us the expected 0 or 1. If you have a multi-class classification problem the last layer in your network will likely be a softmax function and the outputs of each node will be the probability (in the network's eyes) that each node is the right answer. Calling *predict_classes* there will return the node associated with the highest valued probability.

But back to our example:

```
model.predict_classes(test_vec)
```

```
array([[0]], dtype=int32)
```

A "negative" sentiment indeed.

A sentence that contains words like happiness, sun, puffy, clouds isn't necessarily a sentence full of positive emotion. Just as a sentence with dismal, break, and down isn't necessarily a negative sentiment. But with a trained neural network we were able to detect the underlying pattern, to learn something that generalized from data, without ever hard-coding a single rule.

7.4.8 Where Do We Go From Here?

In the introduction we talked about CNN's in their importance in image processing. One key point that was breezed over is the ability of the network to process *channels* of information. In the case of a black and white image there is one channel in the 2 dimensional image. Each data point is the gray-scale value of that pixel which gives us a 2 dimensional input. In the case of color, the input is still a pixel intensity, but it is separated out into its red, green, and blue components. The input then becomes a 3 dimensional tensor that is passed into the net. And the filters follow suit and become 3 dimensional as well, still a 3x3 or 5x5 or whatever in the x,y plane but also 3 layers deep, resulting in filters that are 3 pixels wide x 3 pixels high x 3 channels deep. This leads to an interesting application in Natural Language Processing.

As our input to the network was a series of words represented as vectors lined up next to each other, 400 (maxlen) words wide x 300 elements long and we used word2vec embeddings for the word vectors. But there are multiple ways to generate word embeddings as we have seen in earlier chapters. If we pick several and restrict them to the an identical number of elements we can stack them as we would would picture *channels*. This is an interesting way to add information to the network, especially if the embeddings come from disparate sources. It remains to be seen if this provides significant improvement over the baseline Convolutional Neural Net, especially given the multiplier effect it has on all the computations at train time, but it does tie back to the concepts in image processing nicely.

We touched briefly on the output of the convolutional layers (before we step into the Feed Forward layer). This *semantic representation* is an important artifact. It is in many ways a numerical representation of the the thought and details of the input text. Specifically in this case it is a representation of the thought and details through the lens of sentiment analysis, as all the "learning" that happened was in response to whether the sample was labeled as a positive or negative sentiment. The vector that was generated by training on a set that was labeled for another specific topic and classified as such would contain much different information. It is not common to use the intermediary vector directly from a Convolutional Neural Net, but in the coming chapters we will see examples from other neural network architectures were the details of that intermediary vector become very important, and in some cases are the end goal itself.

So why would you choose a Convolutional Neural Net (CNN) for your NLP classification task. The main benefit they provide is efficiency. In many ways, because of the pooling layers and the limits created by filter size (though you can make your filters

large if you wish), we are throwing away a good deal of information. But that does not mean they aren't useful models. As we have seen they were able to efficiently detect and predict sentiment over a relatively large dataset, and while we relied on the word2vec embeddings, CNN's can perform on much less rich embeddings without mapping the entire language.

Where can you take CNN's from here? A lot can depend on the available datasets, but richer models can be achieved by stacking Convolutional layers and passing the output of the first set of filters as the "image" sample into the second set and so on. Research has also found that running the model with multiple size filters and concatenating the output of each size filter, for each sample, into a longer *thought vector* ahead of passing it into the Feed Forward network at the end for classification can provide more accurate results. The world is wide open, sometimes frustratingly so. Experiment and enjoy.

7.4.9 Summary

In this chapter,

- A convolution is window sliding over something larger (keeping the focus on a subset of the greater whole)
- Neural Networks can treat text just as they treat images and "see" them
- Handicapping the learning process actually helps
- Sentiment exists not just in the words but in the patterns they are used
- There are many knobs to turn on a neural network

8

Loopy (Recurrent) Neural Networks (RNNs)

In this chapter

- Concept of memory in a neural net
- Structure of recurrent neural net
- Data handling for RNNs
- Backpropagation Through Time (BPTT)

So we saw in Chapter 7 that Convolutional Neural Nets can analyze a fragment or sentence all at once, keeping track of the words nearby each other in the sequence by passing a filter composed of shared weights over those words ("convolving" over them). Words that occurred in clusters could be detected together. If those words jostled a little bit in position, the network could be resilient to it. Most importantly, concepts that appeared in relation to one another could have a big impact on the network. But what if we want to look at the bigger picture and consider those relationships over a longer period of time, a broader window than just three or four tokens of a sentence. Can we give the net a concept of what went on earlier? A memory?

For each training example (or batch of unordered examples) and output (or batch of outputs) of a Feed Forward Net, the weights of the net will be adjusted in the individual neurons based on the error through backpropagation. This we've seen. However, the effects of the learning stage of the next example are largely independent of the order of input data. Convolutional Neural Nets make an attempt to capture that ordering relationship, by capturing localized relationships, but there is another way.

In a Convolutional Neural Network we passed in each sample as a collection of word tokens gathered together. The word vectors arrayed together to form a matrix. The shape

of the matrix was (length-of-word-vector x number-of-words-in-sample).

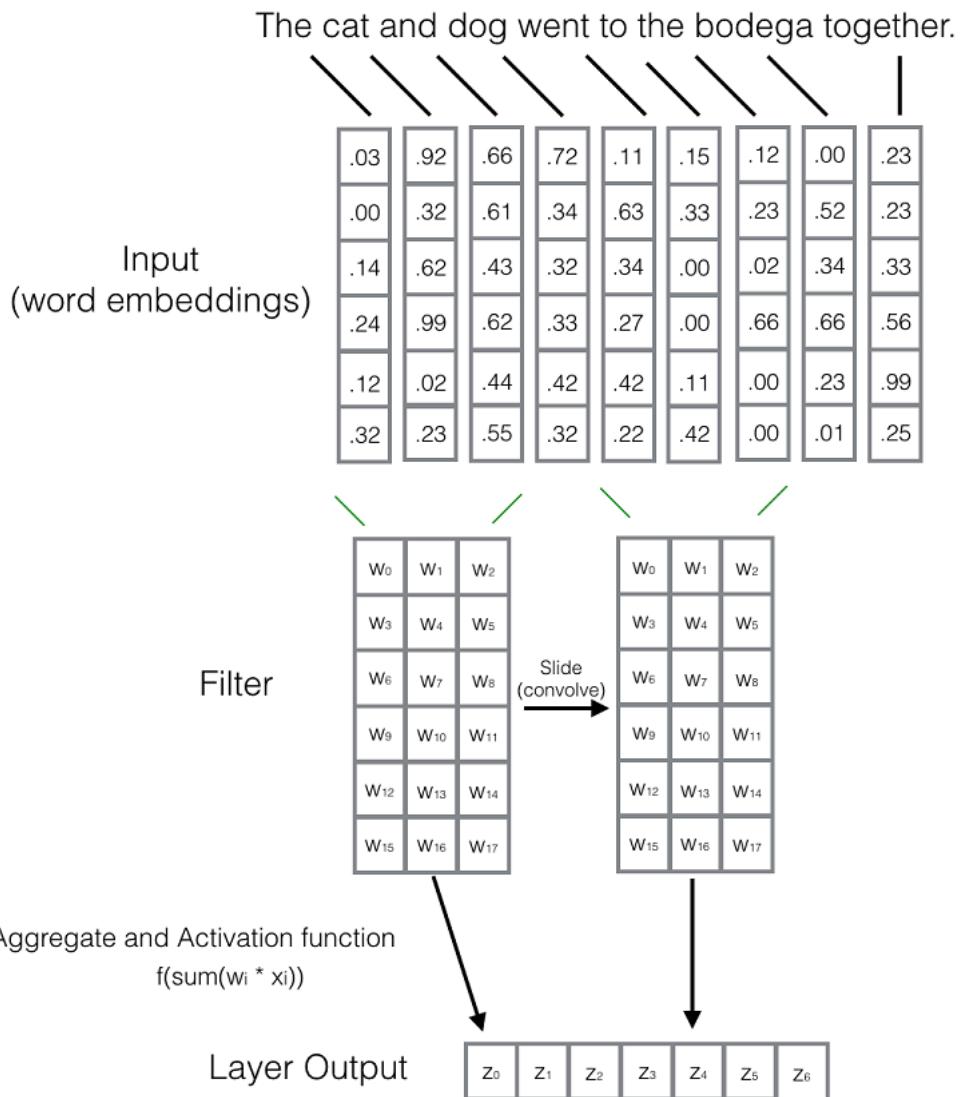


Figure 8.1 1 Dimensional Convolution with Embeddings

But that sample could just as easily be passed into a standard Feed Forward network from chapter 5 right?

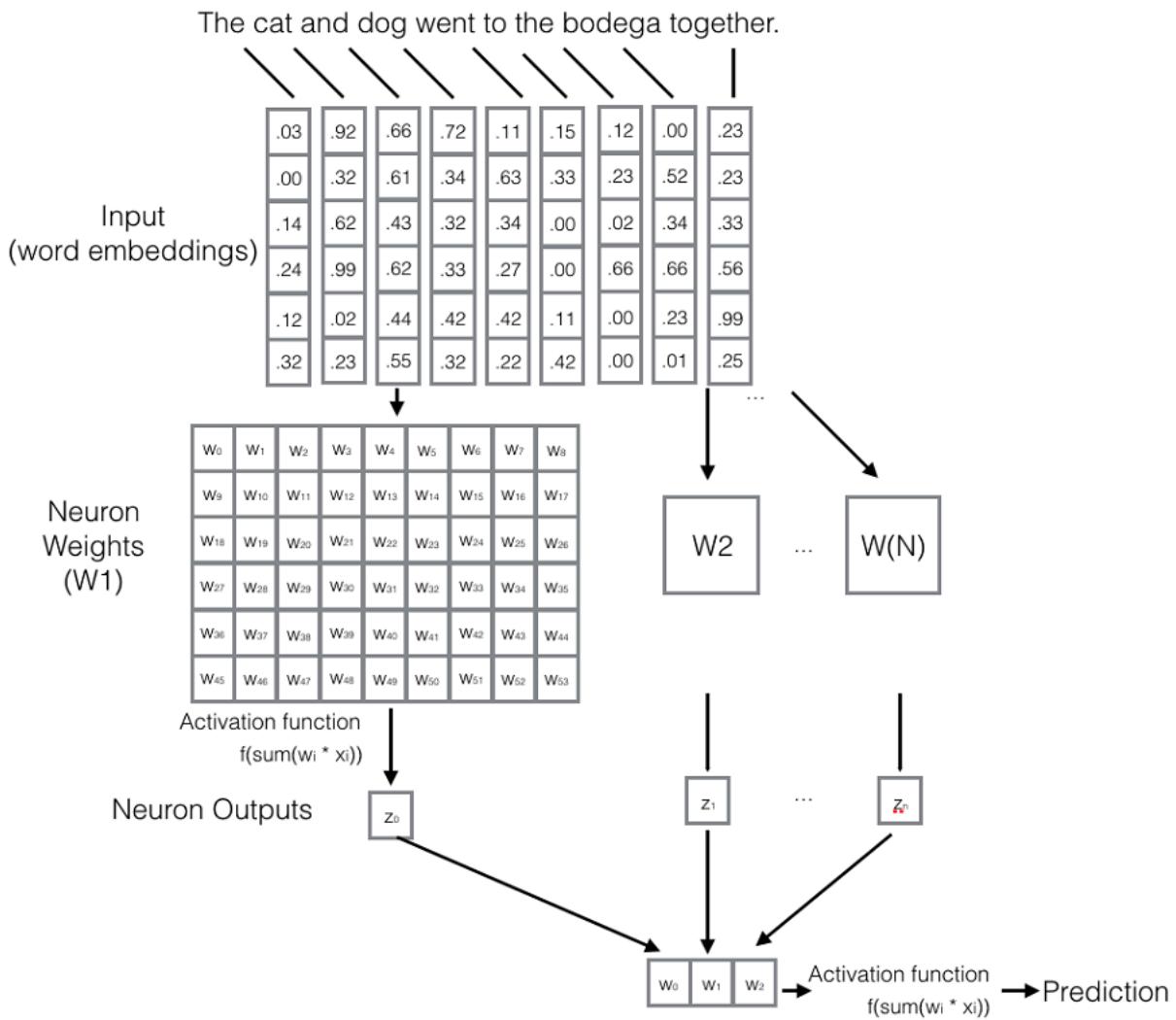


Figure 8.2 Text into a Feed Forward Network

Sure, this is absolutely a viable model, but while a Feed Forward network may make notice of co-occurrences of tokens when they are passed in this way, the path for the network to learn about relationships of words based on their order is much more tenuous. A Feed Forward network's main strength is to model the relationships of a the members of a particular data-point as whole to its associated label.

The 1-dimensional convolutions gave us a path into this inter-token relationships by looking at *windows* of words together and the pooling layers discussed in Chapter 7 where specifically designed to handle slight variations in word order. In this chapter, we will look at a different approach. And through this, we will take a first step toward the concept of *memory* in a neural network. Instead of thinking about language as large dump of data, we can begin to look at it as it is created, token by token, over *time*.

8.1 Remembering with Recurrent Networks

Of course, the words in a document are rarely completely independent of each other, their occurrence predicates or is predicated by occurrences of other words in the document.

The stolen car sped into the arena.

The clown car sped into the arena.

Two very different emotions may arise in the reader of the two sentences above, specifically as the reader reaches the end of the sentence. The two sentences are identical in adjective, noun, verb, prepositional phrase construction. However, that "early" adjective swap, as in the reader encounters it early in the process of reading the sentence, has a profound effect on what the reader may infer is going on in general!

Can we find a way toward modeling that relationship? A way to understand that "arena" and even "sped" could take on slightly different connotations because of an adjective that does not directly modify either occurred earlier in the sentence?

If we can find a way to *remember* what just happened in the moment before (specifically what happened at time step t when we are looking at time step $t+1$) we would be on the way to capturing the patterns that emerge when certain tokens appear in specific relations to other tokens in a sequence. *Recurrent Neural Nets* (RNNs) are the first step toward being able to remember.

RNN

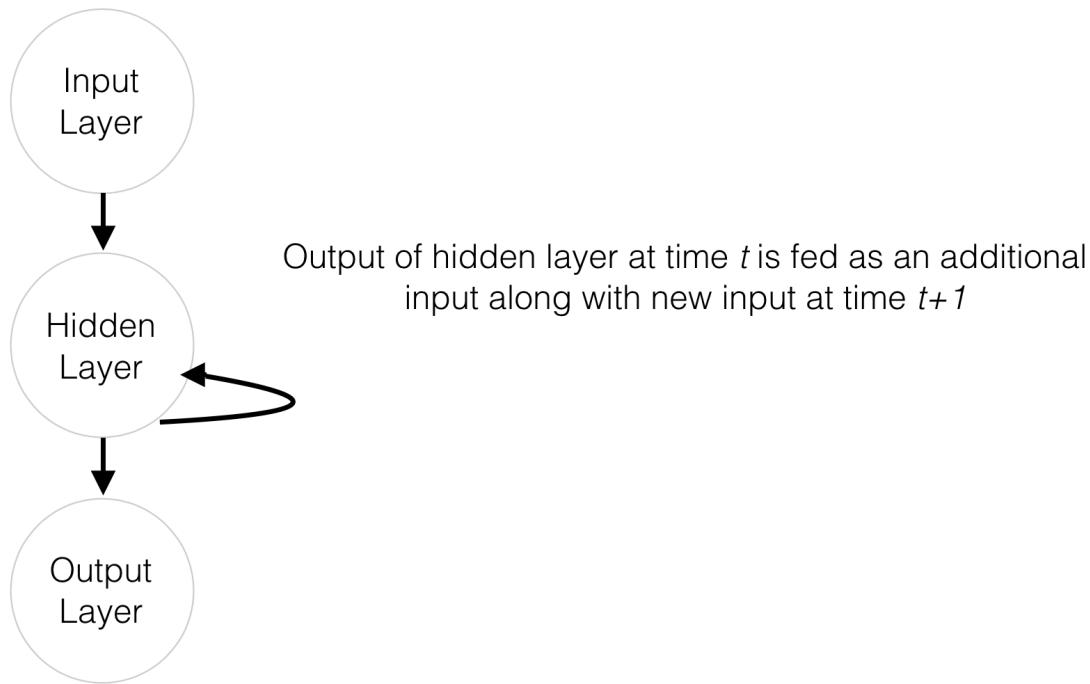


Figure 8.3 Recurrent Neural Net

While the idea of affecting state across time can be a little mind bending at first, the basic concept is simple. For each input we feed into a regular Feed Forward Net, we would like to take the output of the network at time step t and provide it as an additional input, along with the next piece of data being fed into the network at time step $t+1$. We tell the Feed Forward network what happened before along with what is happening "now".

TIP

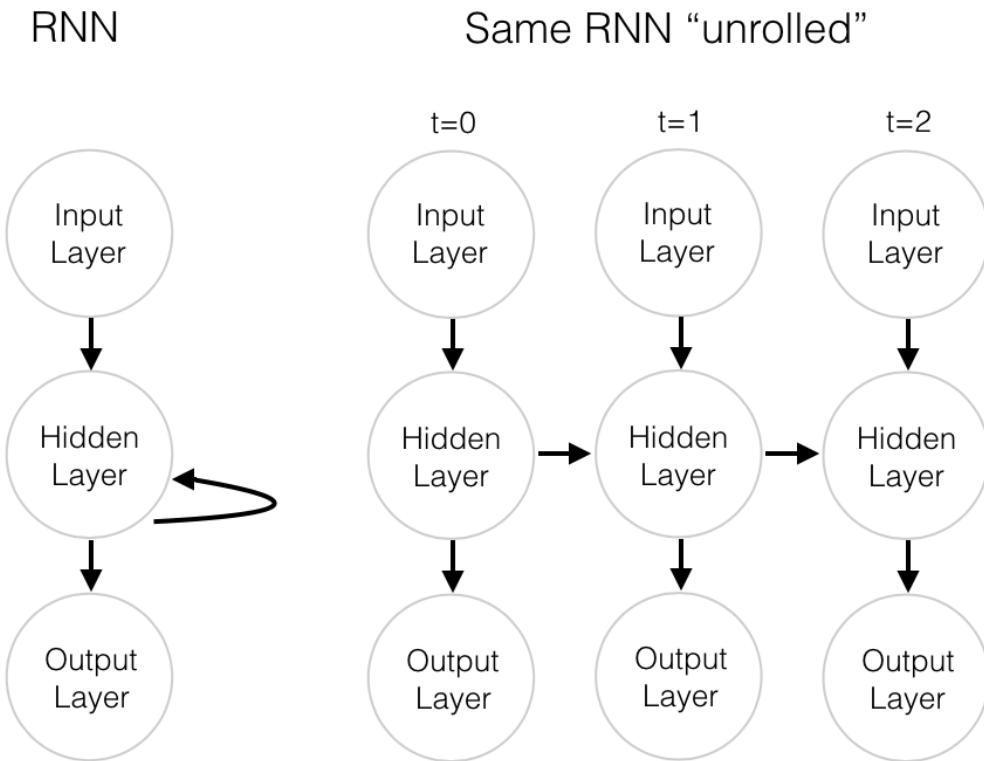
In this chapter and the next we will be discussing most things in terms of time steps. This is not the same thing as individual data samples from previous chapters. We are referring to single data samples split into smaller chunks that represent changes over time. For us the single data sample will still be a piece of text, say a short movie review or a tweet. As before we will tokenize the sentence. But where before we put those tokens into the network all at once, we will now put them in one at a time. *This is not the same as having multiple new samples.* The tokens are still part of *one* data sample with *one* associated label.

The tokens, in the order they appear in the sentence, will be the inputs at each *time step*. This segmentation (word level sentence segmentation at first and later at the character level) will occur for each individual data sample. The steps of putting each token in will be *sub-steps* of feeding the data sample into the network.

Throughout we will reference the current time step as t and the following time step as $t+1$.

You can visualize this as above: the circles are entire Feed Forward network *layers* composed of one or more neurons. The output of the *hidden layer* is output from the network as normal, but it is also set aside to be passed back in as an input to *itself* along with the the normal input from the next time step. This feedback is represented with an arc from the output of a layer back into it own inputs.

The easier way to see this, and its more commonly shown this way, is by *unrolling* the net.

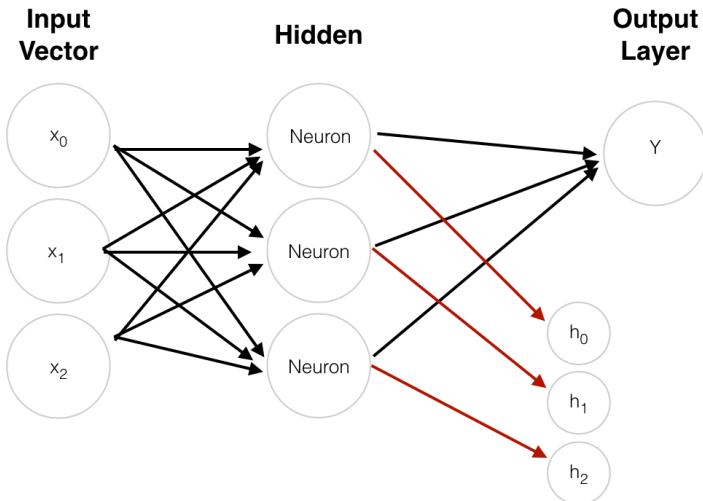


Same Network stood on its head and just the layers represented.
Output of hidden layer from input at one *time step* (t) is fed back into
the hidden layer along with input data from next *time step* ($t+1$).

Figure 8.4 Unrolled Recurrent Neural Net

Each time step represented in the unrolled version of the network is the *same* network. The network to the right is the *future* version of the network on the left. All of the vertical paths in this visualization are clones. They are just the single network represented on a time line. This visualization is indeed helpful, especially when we talk about how information flows through the network forward and *backward* during backpropagation, but it is easy to lose sight of the fact that this is just one network with one set of weights.

Let's zoom in on the original representation of a Recurrent Neural Network before we unrolled it and expose the input-weight relationships. We get something that looks like this:



In a Fully-Connected Network, all nodes feed into each node in the neuron in the next layer, but with a trick. Time step $t = 0$.

Figure 8.5 Detailed Recurrent Neural Net at Time Step $t = 0$

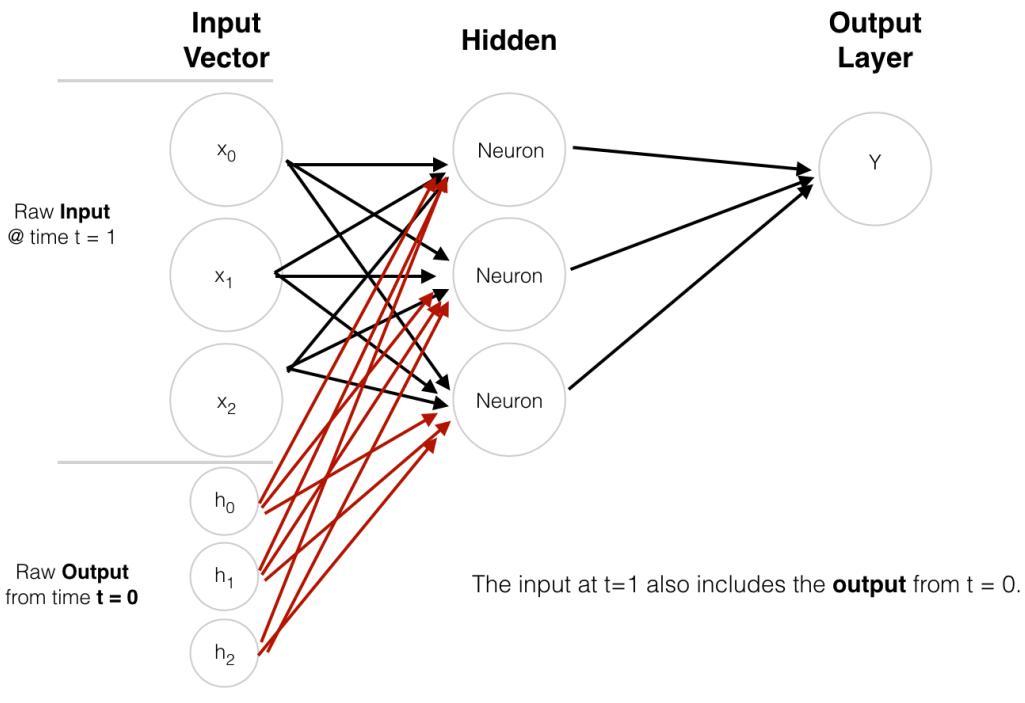


Figure 8.6 Detailed Recurrent Neural Net at Time Step $t = 1$

Each neuron in the hidden state has a set of weights that it applies to each of the elements of each input vector, as a normal Feed Forward network. But now we have an additional

set of trainable weights that is applied to the output of the hidden neurons from the previous time step. The network can learn how much weight or importance to give the events of the "past" as we input a sequence token by token.

TIP

The first input in a sequence will have no "past" so the hidden state at t=0 receives an input of 0 from its t-1 self. There are variations on this where related but separate samples that are input one after the other pass their final output back to t=0 of the next input, but we'll come back to that later when we discuss *statefulness*.

Let's turn our attention back to the data, imagine we have a set of documents, each a labeled example. For each example, instead of passing the collection of word vectors into a Convolutional Neural Net all at once as in the last chapter, we take the sample one token at a time.

Convolutional Neural Net Or Feed Forward Net

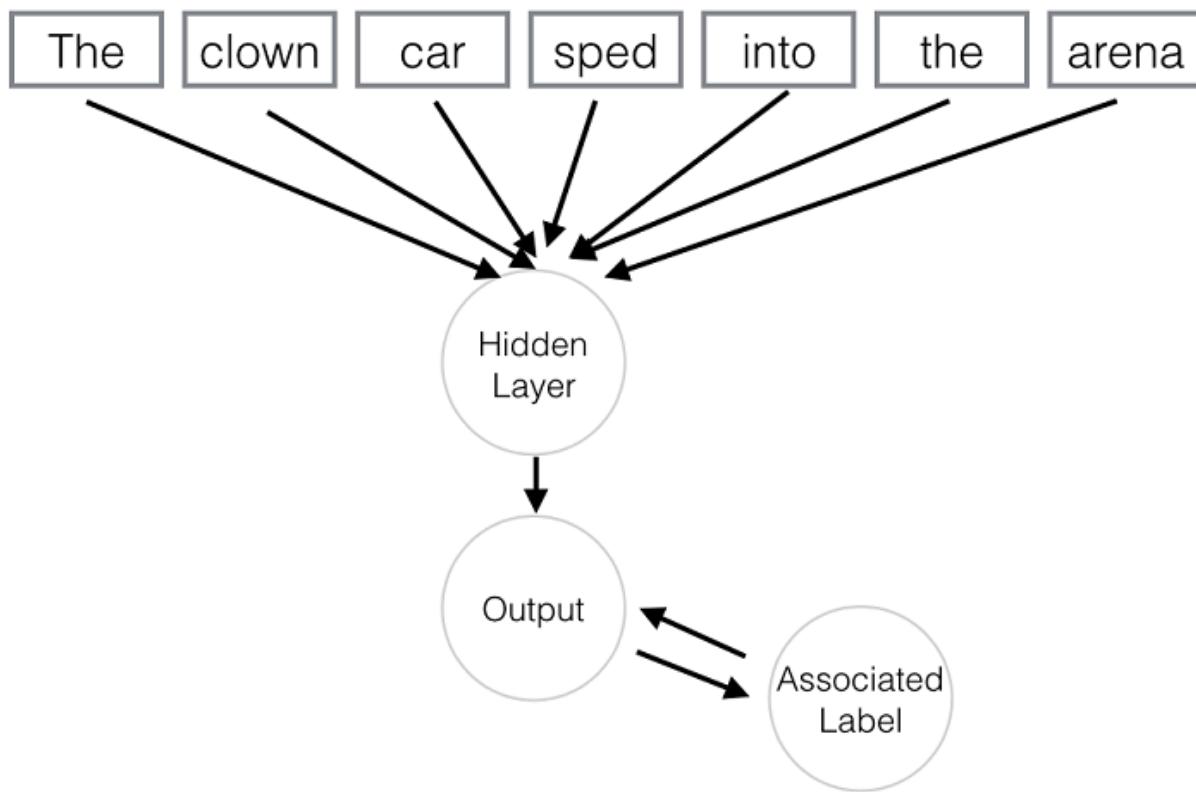


Figure 8.7 Data into Convolutional Network

In our Recurrent Neural Net, we pass in the word vector for the first token and get the network's output. We then pass in the second token, but we also pass in the output from

the first token! And then pass in the third token along with the output from the second token. And so on. The network has a concept of before and after, cause and effect, some vague notion of time.

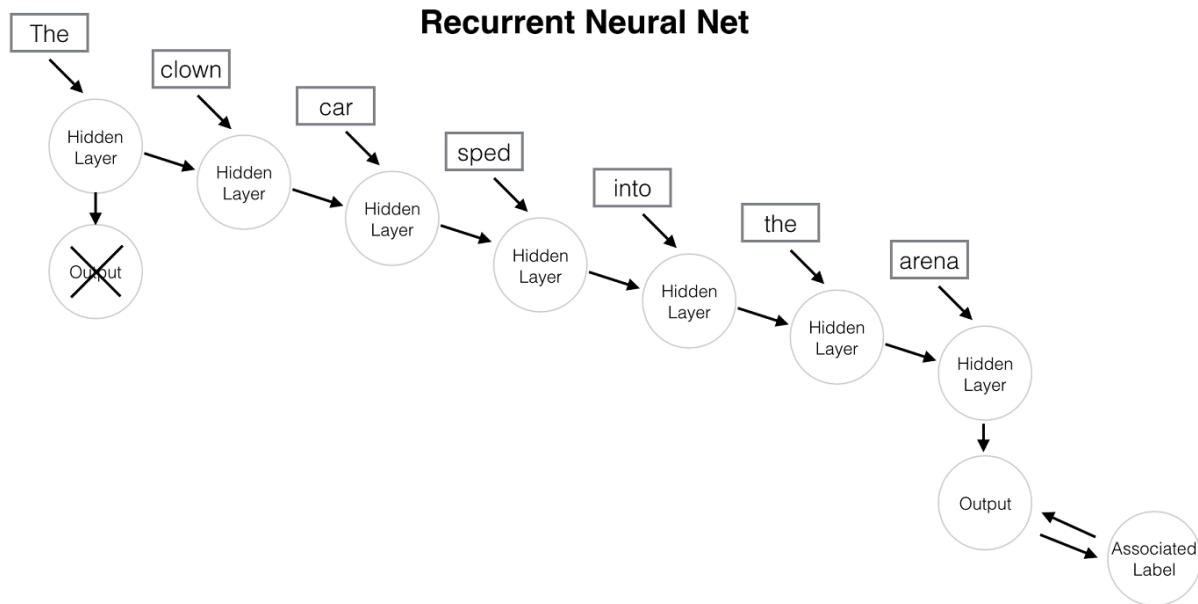


Figure 8.8 Data into Recurrent Network

So now we are remembering! Sort of. Well. There are a couple of things we need to figure out. For one, how does backpropagation even work in a structure like this?

8.1.1 Backpropagation Through Time

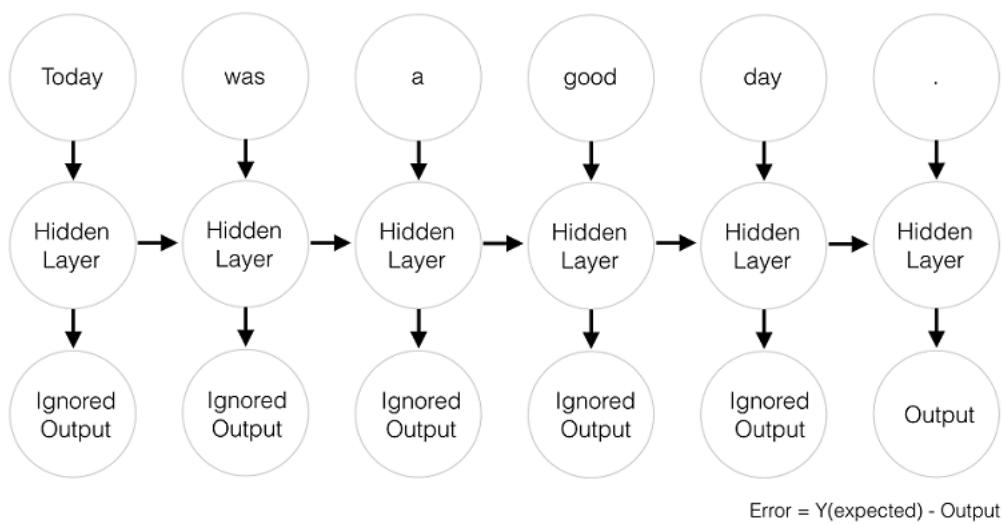
With all the networks we've talked about so far, we have a label, a target to aim for, and this is no exception. But we don't really have a concept of a label for each token. We only have a label for the sample, for all of the tokens taken together.

TIP

We are speaking about tokens as the input to each time step of the network, but Recurrent Neural Nets work identically with any sort of time series data. Readings from a given weather station, musical notes, characters in a sentence, you name it.

Here we will initially look at just the output of the network at the last time step and compare that to the label. That will be (for now) the definition of the *error*. And of course, the error is what ultimately our network will try to minimize. But here we now have something that is a shift from what we had in the earlier chapters. For a given data sample, we break it into smaller pieces that are fed into the network sequentially. But

instead of dealing with the output generated by any of these "sub-samples" directly, we feed it back into the network. We are only concerned with the final output, at least for now.



We input each token from the sequence and calculate the error/loss based on the output of the last iteration of the sequence.

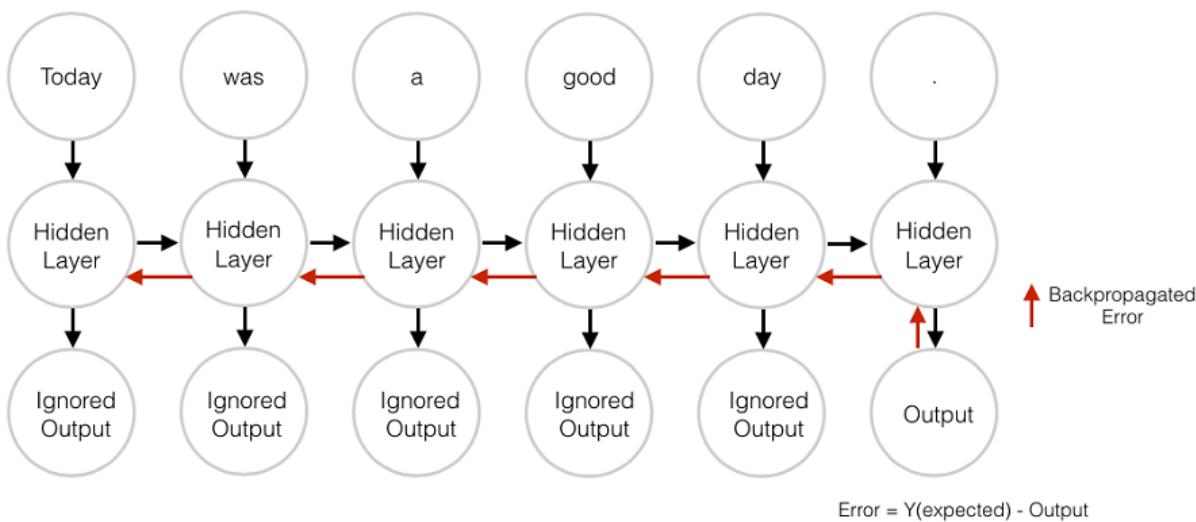
Figure 8.9 Only Last Output Matters Here

With an error for a given sample, we just need to figure out what weights to update and by how much. In Chapter 5, we learned how to backpropagate the error through a standard network, and we know that the correction to the weight is dependent on how much that weight contributed to the error. But here the concept of applying this over time seems to throw a wrench into this whole process.

One way to think about it is, go ahead and consider the process to be time based as you enter each "sub-sample" into the next. One column of the image above appearing at a time. But once you get to the end: after all of the pieces of the sample fed in and have the final output in hand and you've calculated the error, you have this picture, this graph if you will, the *unrolled net*.

At this point, you can consider the whole of the input as static. You can see which neuron fed which input all the way through the graph. And once you know how each neuron fired, you can just go back through the chain along the same path and *backpropagate* as with the standard Feed Forward network.

We use the chain-rule to backpropagate to the next layer below. But instead of going to the layer below we go to the layer in the *past*, as if each unrolled version of the network were different. The math is the same.



The error from the last step is backpropagated back in time. Each “older” step takes the gradient with respect to the newer step. The changes are aggregated and applied to the single set of weights after all have been calculated back to $t=0$

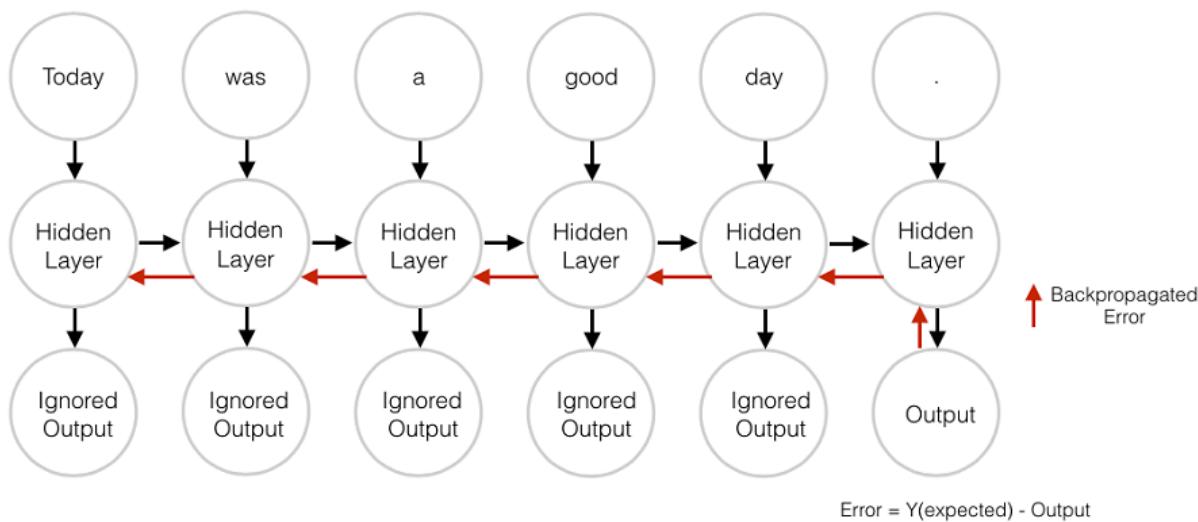
Figure 8.10 Backpropagation Through Time

TL;DR RECAP

- Break each data sample into tokens
- Pass each token into a Feed Forward net
- Pass the output of each time step as input into the same layer with the input from the next time step
- Collect the output of the last time step and compare it to the label
- Backpropagate the error through the whole graph, all the way back to the first input at time step 0.

8.1.2 When Do We Update What?

We have converted our strange Recurrent Neural Network into something that looks like a standard Feed Forward network, so updating the weights should be fairly straightforward. There is one catch though. The tricky part of the update process is the weights we are updating are not just each a different branch of a neural network. Each leg is the *same* network at different time steps. The weights are the *same* for each time step.



The error from the last step is backpropagated back in time. Each "older" step takes the gradient with respect to the newer step. The changes are aggregated and applied to the single set of weights after all have been calculated back to $t=0$

Figure 8.11 Backpropagation Through Time

The simple solution to this is, the weight corrections are calculated at each time step, but not immediately updated. In a Feed Forward network all of the weight updates would be calculated once all of the gradients had been calculated for that input. Here the same holds, but we have to hold the updates until we go all the way back in time, to time step 0 for that particular input sample.

The gradient calculations need to be based on the values that the weights had when they contributed that much to the error. Now the mind-bending part of this. A weight at time step t contributed something to the error when it was initially calculated. That *same* weight received a different input at time step $t+1$ and therefore contributed a different amount to the error then.

So we can figure out the various changes to the weights (as if they were in a bubble) at each time step and then sum up the changes and apply the aggregated changes to each of the weights of the hidden layer as the very last step of the learning phase.

TIP

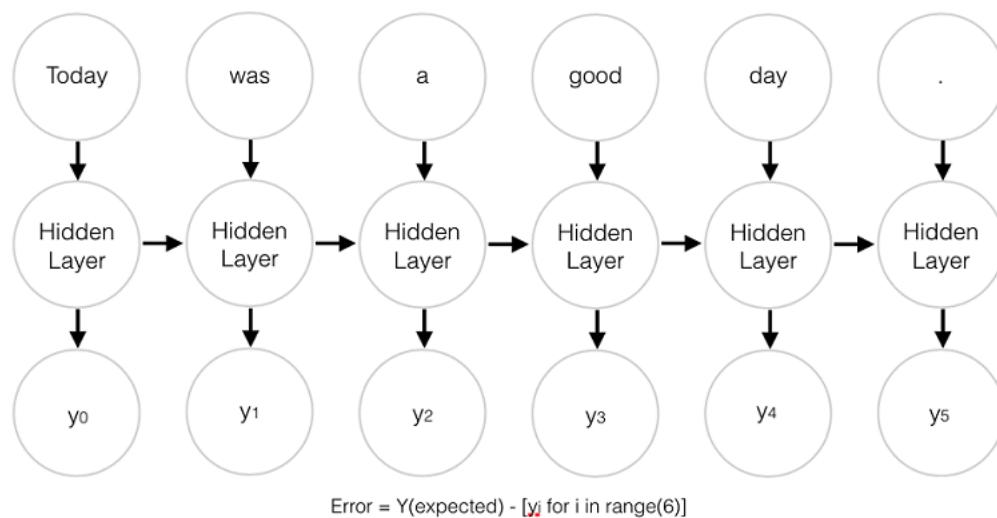
In all of these examples, we have been passing in a single training example, the *forward pass*, and then backpropagating the error. As with any Neural Network this can happen after each training sample or in batches. And it turns out there are benefits other than just speed to batching. But for now, just think of these processes in terms of single data samples.

That seems like quite a bit of magic. As we backpropagate through time, a single weight may be pushed up at one time step t (given how it reacted to the input at time step t) and

be pushed down at time step for $t-1$ (given how it reacted to the input at time step $t-1$), for a single data sample! But remember, neural networks in general work by minimizing a loss function, regardless of how complex the intermediary steps are. In aggregate it will optimize across this complex function. As the weight update is applied just once per data sample, the network will settle (assuming it converges) on the weight for that given position in the given neuron that best handles this task.

BUT I DO CARE WHAT CAME OUT OF THE EARLIER STEPS

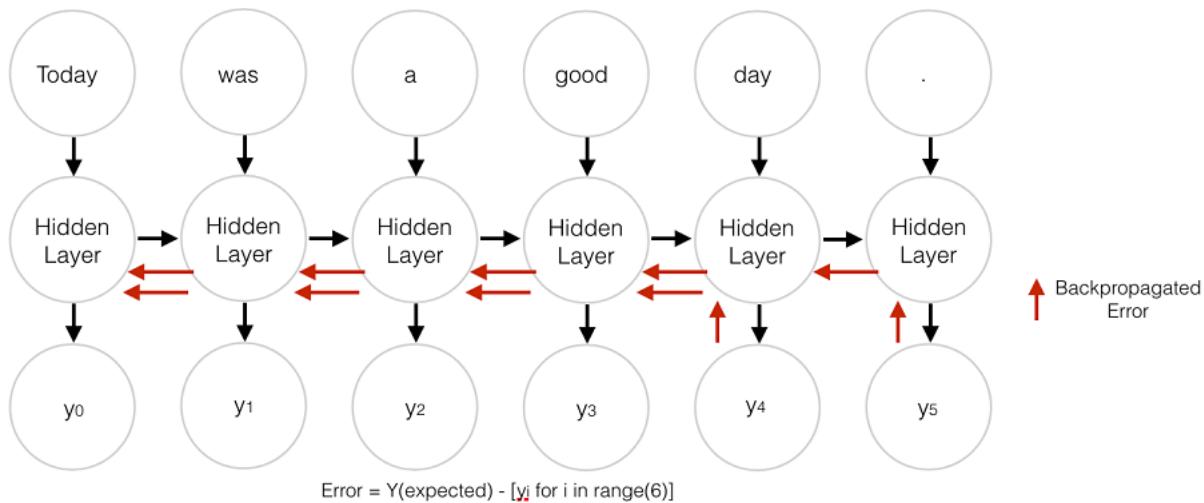
There are times where we may care about the entire sequence that is generated by each of the intermediary time steps as well. We will see later in Chapter 9 examples where the output at a given time step t is just as import as the output at the final time step. There is a path to capturing the error at any given time step and carrying that backwards in the backpropagation as well.



Or we can collect the sequence and calculate the error from an expected sequence.
The error at each timestep is separately backpropagated through time.

Figure 8.12 All Outputs Matter Here

This process is just like the normal backpropagation through time for n time steps. In this case, we are now backpropagating the error from multiple sources at the same time. But as with the first example, the weight corrections are additive. So we backpropagate from the last time step all the way to the first, summing up what we will change for each weight. Then we do the same with the error calculated at the second to last time step and sum up all the changes all the way back to $t=0$. We repeat this process until we get all the way back down to time step 0 and then just backpropagate it as if it were the only one in the world. We then apply the grand total of the updates to the corresponding weights of the hidden layer all at once as before.



The error is backpropagated from each output all the way back to $t=0$ and aggregated before finally applying changes to the weights.

Figure 8.13 Multiple Outputs and Backpropagation Through Time

This is the most import take away of this section. As with a standard Feed Forward network: you always update the weights *only after* you have calculated the proposed change in the weights for the entire backpropagation step for that input (or set of inputs). In the case of Recurrent Neural Net, this includes the updates all the way back time $t=0$.

Updating the weights earlier would "pollute" the calculations of the gradient in the backpropagations earlier in time. Remember the gradient is calculated with respect to a particular weight. If you were to update it early, say at time step t , when you go to calculate the gradient at time step $t-1$, the value of that weight (remember it is the *same* weight position in the network) would have changed. Computing the gradient based on the input from time step $t-1$, the calculation would be off. You would be punishing (or rewarding) a weight for something it did not actually do!

8.1.3 Recap

So were are we? We've segmented each data sample into tokens. Then one by one we fed them into a Feed Forward network. With each token, we input the token itself and the output from the previous time step along side each other (at time step 0 we input the initial token and 0 or a 0 vector, since there is no previous output). We get our error from the difference between the output of the network from the final token and the expected label. We then backpropagate that error to the weights of the network, backward through time. We aggregate the the proposed updates and apply them all all at once to the network.

We now have a Feed Forward Network that has some concept of time and a rudimentary

tool for memory about occurrences that happen in that time line.

8.1.4 There's Always a Catch

While a Recurrent Neural Net may have relatively fewer weights (parameters) to learn, you can see from the above how a recurrent neural can quickly get very expensive to train, especially for sequences of any length, say greater than 10 tokens. The more tokens you have, the further back in time each error must be backpropagated. For each step back in time, there are ever more derivatives to calculate. This doesn't mean Recurrent Neural Nets are any less effective, just get ready to heat your house with your computer's exhaust fan.

Great, new heat sources aside, we have given our neural network a rudimentary memory. But there is another, more troublesome, problem. It is a problem that you also see in regular Feed Forward networks as they get deeper. It is known as the *vanishing gradient problem* and it has a corollary the *exploding gradient problem*. The idea is that as a network gets deeper (more layers) the error signal has a hard time reaching the deepest layers.

This same problem applies to Recurrent Neural Nets as each time step back in time is the mathematical equivalent of backpropagating an error "down" a layer in a Feed Forward network. But its worse here! While most Feed Forward networks tend just to be a few layers deep for this very reason, we are dealing with sequences of tokens five, ten, or even hundreds long. Getting to the bottom of a hundred layer deep network is going to be difficult. There is a mitigating factor that keeps us in the game though. While the gradient may vanish on the way to the last weight set, we are really only updating one weight set, which is the same at every "layer" or time step. So some information is going to get through, it just might not be the ideal memory state we thought we had created. But fear not, research is on the case, and we have new answers for that in the next chapter.

Enough doom and gloom, let's see some magic.

8.1.5 Recurrent Neural Net with Keras

We'll start with the same dataset and preprocessing as we had in the previous chapter. First, we load the dataset and grab the labels and shuffle the examples. Then we will tokenize it and vectorize it again using the Google word2vec model. Next, we grab the labels in an ordered set. And finally we split it 80/20 into the training and test sets.

```
import glob
import os
```

```

from random import shuffle

from nltk.tokenize import TreebankWordTokenizer
from gensim.models.keyedvectors import KeyedVectors
word_vectors = KeyedVectors.load_word2vec_format('<path to your download
    word2vec file>/GoogleNews-vectors-negative300.bin.gz', binary=True, limit=200000)

def pre_process_data(filepath):
    """
    This is dependent on your training data source but we will try to generalize it as
    best as possible.
    """
    positive_path = os.path.join(filepath, 'pos')
    negative_path = os.path.join(filepath, 'neg')

    pos_label = 1
    neg_label = 0

    dataset = []

    for filename in glob.glob(os.path.join(positive_path, '*.txt')):
        with open(filename, 'r') as f:
            dataset.append((pos_label, f.read()))

    for filename in glob.glob(os.path.join(negative_path, '*.txt')):
        with open(filename, 'r') as f:
            dataset.append((neg_label, f.read()))

    shuffle(dataset)

    return dataset

def tokenize_and_vectorize(dataset):
    tokenizer = TreebankWordTokenizer()
    vectorized_data = []
    expected = []
    for sample in dataset:
        tokens = tokenizer.tokenize(sample[1])
        sample_vecs = []
        for token in tokens:
            try:
                sample_vecs.append(word_vectors[token])
            except KeyError:
                pass # No matching token in the Google w2v vocab
        vectorized_data.append(sample_vecs)

    return vectorized_data

def collect_expected(dataset):
    """
    Peel off the target values from the dataset """
    expected = []
    for sample in dataset:
        expected.append(sample[0])
    return expected

dataset = pre_process_data('./aclimdb/train')
vectorized_data = tokenize_and_vectorize(dataset)
expected = collect_expected(dataset)

# Divide up the train and test sets
split_point = int(len(vectorized_data)*.8)

x_train = vectorized_data[:split_point]
y_train = expected[:split_point]

```

```
x_test = vectorized_data[split_point:]
y_test = expected[split_point:]
```

We'll basically use the same hyperparameters for this model. 400 tokens per example, batches of 32. Our word vectors are 300 elements long. And we'll let it run for 2 epochs again.

```
maxlen = 400
batch_size = 32
embedding_dims = 300
epochs = 2
```

Next we'll need to pad/truncate the samples again. We won't always need to do this with Recurrent Neural Nets as they can take multivariate length input, but you will see in the next few steps that this case does require our sequences to be of matching length.

```
import numpy as np

x_train = pad_trunc(x_train, maxlen)
x_test = pad_trunc(x_test, maxlen)

x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims))
y_train = np.array(y_train)
x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
y_test = np.array(y_test)
```

Now that we've got our data back. It's time to build a model. We'll start again with a Sequential() Keras model.

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, SimpleRNN

num_neurons = 50

print('Build model...')
model = Sequential()
```

And then, because Keras is magic, all of the explanation from above and more happens in:

```
model.add(SimpleRNN(num_neurons, return_sequences=True, input_shape=(maxlen,
embedding_dims)))
```

This sets up the infrastructure to take each input and pass it into a simple Recurrent Neural Net (RNN) (the not-simple version is in the next chapter) and for each token, gathers the output into a vector. Since our sequences are 400 tokens long and we are

using 50 hidden neurons, our output from this layer will be a vector 400 elements long. Each of those elements a vector 50 elements long, one output for each of the neurons.

Notice here the keyword argument *return_sequences*. This is going to tell the network to return the value of the network at each time step, hence the 400 values each 50 long. If *return_sequences* was set to False (the Keras default behavior) only a single 50 dimensional vector would be returned.

The choice of 50 neurons was arbitrary for this example, mostly to keep computation time down, but do experiment with this number to see how it affects computation time and how it affects the accuracy of the model.

TIP

It is a good rule of thumb to try and make your model no more complex than the data you are training on. That, of course, is easier said than done, but holding that in your head gives you a rationale for the direction you choose to adjust your parameters as you experiment with your dataset. A more complex model will *overfit* training data and not generalize well, while a model that is too simple will *underfit* the data and also not have much nice to say about novel data. You will see this discussion referred to as the *bias vs. variance* trade off. A model that is overfit to the data is said to have high variance and low bias. And underfit model is the opposite, low variance and high bias.

The other thing to note is we truncated and padded the data again. We did this to provide a comparison with the CNN example from last chapter. However, when using a Recurrent Neural Net this is not always necessary. You can provide training data of varying lengths and simply unroll the net until you hit the end of the input. Keras will handle this automatically. The catch is your output of the RNN layer will vary time step for time step with the input. A 4 token input will output a sequence 4 elements long. A 100 token sequence will get you sequence of 100 elements. So if you need to pass this into another layer that expects a uniform input it will not work. But there are cases where that is acceptable, and even preferred. But back to our classifier.

```
model.add(Dropout(.2))

model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
```

We requested the simple RNN return full sequences but to prevent overfitting we add a Dropout layer to zero out 20% of those inputs. Randomly chosen on each input example. And then finally we add a classifier. In this case, we just have one class "Yes - Positive

Sentiment - 1" or "No - Negative Sentiment - 0" so we chose a layer with one neuron (Dense(1)) and a sigmoid activation function. But a Dense layer expects a "flat" vector of n elements (each element a float) as input. And the data coming out of the SimpleRNN is a tensor 400 elements long and each of those are 50 elements long. But a Feed Forward network also no longer cares about order of elements as long as we are consistent with the order. So we use the convenience layer, Flatten(), that Keras provides to flatten the input from a tensor 400 x 50 to a vector 20,000 elements long. And that is what we pass into the final layer that will make the classification. In reality, the Flatten layer is a mapping so that as the error is backpropagated from the last layer back to the appropriate output in the RNN layer and each of those backpropagated errors are then backpropagated through time from the right point of the output, as discussed above.

Passing the "thought vector" that is produced by the Recurrent Neural Network (RNN) layer into a Feed Forward network will of course no longer keep the order of the input that we tried so hard to incorporate, but the important takeaway is to notice the "learning" related to sequence of tokens happens in the RNN layer itself, the aggregation of errors via backpropagation through time is encoding that relationship in the network and expressing it in the "thought vector" itself. Our judgment on the thought vector via the classifier is providing feedback to the "quality" of that thought vector with respect to our specific classification problem. There are other ways to "judge" our thought vector and work with the RNN itself, but more on that in the next chapter (can you tell we're excited for the next chapter? Hang in there, all of this is critical to understanding the next part).

8.2 Putting Things Together

We compile just as we did with the Convolutional Neural Net in last chapter.

Keras also comes with several tools for introspection of your model. `model.summary()` is one of those tools. As the models grow more and more complicated it can become taxing to keep track of just how things inside change as you adjust the hyperparameters. `model.summary` gives a handy overview of the current model architecture.

```
model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
print(model.summary())
```

```
Using Tensorflow backend.
Build model...
```

| Layer (type) | Output Shape | Param # |
|--------------------------|-----------------|---------|
| <hr/> | | |
| simple_rnn_1 (SimpleRNN) | (None, 400, 50) | 17550 |
| dropout_1 (Dropout) | (None, 400, 50) | 0 |

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

| | | |
|----------------------------------|---------------|-------|
| <code>flatten_1 (Flatten)</code> | (None, 20000) | 0 |
| <code>dense_1 (Dense)</code> | (None, 1) | 20001 |
| <hr/> | | |
| Total params: 37,551.0 | | |
| Trainable params: 37,551.0 | | |
| Non-trainable params: 0.0 | | |
| <hr/> | | |
| None | | |

Here we are going to pause and look at the number of parameters we are working with. This is a relatively small recurrent neural network, but you can see we are learning 37,551 parameters! That's a lot of weights to update based on 20,000 training samples (not to be confused with the 20,000 elements in the last layer, that is just coincidence). Let's look at those numbers and see specifically where they come from.

In the SimpleRNN layer, we requested 50 neurons. Each of those neurons will receive input (and apply a weight to) each input sample. In an RNN the input at each time step is one token and our tokens are represented by word vectors in this case, each 300 elements long (300-dimensional). So each neuron will need 300 weights. That gives us:

$$50 * 300 = 15000$$

Each neuron also has the *bias* term which always has an input of 1 but has a trainable weight on it. So that adds:

$$15000 + 50 \text{ (bias weights)} = 15050$$

15,050 weights in the first time step of the first layer. Now each of those 50 neurons will feed its output into the next time step of the network. Each neuron accepts the full input vector as well as the full output vector. In the first time step the feedback from the output doesn't exist yet, it is initiated as a vector of zeros, its length the same as the length of the output.

So each neuron in the hidden layer now has weights corresponding to each element of the input, 300 in this case, plus one for the bias input, plus 50 for the results of the output in the previous time step (or zeros if $t=0$). *The key feedback step in a Recurrent Neural Network.* That gives us:

$$300 + 1 + 50 = 351$$

And times 50 neurons, we have:

$$351 * 50 = 17550$$

17550 parameters to train. We are *unrolling* this net 400 time steps (probably too much given the problems associate with vanishing gradients, but even so this network turns out to still be effective). But those 17550 parameters are the same in each of the unrollings, and remain the same until all the backpropagations have been calculated. The updates to the weights occur at once at the end of the sequence forward propagation and subsequent backpropagation. While we are adding complexity to the backpropagation algorithm we are saved by the fact we are not training a net with 7.02 million parameters ($17550 * 400$) which is what it would look like if the unrollings each had their own weight sets.

The final layer in the summary is reporting 20,001 parameters to train. And this relatively straightforward. After the Flatten() layer the input is a 20,000 dimensional vector plus the one bias input. And sense we only have one neuron in the output layer, the total number of parameters is:

$$(20,000 \text{ input elements} + 1 \text{ bias unit}) * 1 \text{ neuron} = 20,001 \text{ parameters.}$$

Now those numbers can be a little misleading in computational time as there are so many extra steps to backpropagation through time, compared to something like Convolutional Neural Networks or standard Feed Forward networks. This is of course shouldn't be a deal killer by any stretch as Recurrent Nets' special talent at *memory* is just the start of a bigger world in NLP or any other time-series data as we'll see in the next chapter. But back to our classifier for now.

8.3 Let's Get to Learning Our Past Selves

```
model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           validation_data=(x_test, y_test))
model_structure = model.to_json()
with open("simplernn_modell.json", "w") as json_file:
    json_file.write(model_structure)

model.save_weights("simplernn_weights1.h5")
print('Model saved.')
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [=====] - 215s - loss: 0.5723 - acc: 0.7138 - val_loss: 0.5011 - val_acc: 0.7676
Epoch 2/2
20000/20000 [=====] - 183s - loss: 0.4196 - acc: 0.8144 - val_loss: 0.4763 - val_acc: 0.7820
Model saved.
```

Not horrible, but also not something we'll probably write home about. So where can we

look to improve ...

8.4 Hyperparameters

In all of the models we have listed in the book, there are various ways to tune them toward your data, toward your samples, they all have their benefits and associated trade offs. There are also enough choices and combinations of choices that finding the "perfect" set of hyperparameters is usually an intractable problem. But human intuition and experience can at least provide approaches the problem. Let's look at the last example. What are some of the choices we made?

```
maxlen = 400          ①
embedding_dims = 300  ②
batch_size = 32        ③
epochs = 2
num_nuerons = 50      ④
```

- ① Arbitrary sequence length based on perusing the data.
- ② This came from the pre-trained word2vec model
- ③ Number of sample sequences to pass through (and aggregate the error) before backpropagating
- ④ Hidden layer complexity

maxlen This is most likely the biggest question mark in the bunch. The training set varies widely in sample length and forcing samples that are less than 100 tokens long up to 400 and conversely chopping down 1000 token samples to 400, introduces an enormous amount of noise. Changing this number will impact training time more than any other parameter in this model. As the length of the individual samples dictates how many and how far back in time the error must backpropagate. It will turn out this isn't strictly necessary with Recurrent Neural Networks. You can simple unroll the network as far or as little as you need to for the sample. It is necessary in our example as we are passing the output, itself a sequence, into a Feed Forward layer; and Feed Forward layers require uniformly sized input.

embedding_dims This value was dictated by the word2vec model we chose, but this could easily be anything that adequately represents the dataset. Even something as simple as a one-hot encoding of the top 50 most commons tokens in the corpus may be enough to get accurate predictions on.

batch_size As with any net, increasing batch size will speed training as it reduces the

number of times backpropagation (the computationally expensive part) needs to happen. The trade-off is the larger the batch the greater the chance of settling in a local minimum.

epochs This one is easy to test, it just requires patience. Keras models can restart training. If you have saved the model simply reload it and your dataset and call `model.fit()` again passing in your data, or additional data. It will not reinitialize the weights but continue training as if nothing else changed. The other alternative is to add a Keras *callback* called `EarlyStopping`. By providing this method to the model the model will continue to train up until the number of epochs provided, *unless* a metric passed to `EarlyStopping` is met. Such as validation accuracy doesn't improve for a certain number of epochs in a row. This allows you to set it and forget it, as the model will stop training when it hits your metric, and you don't have to worry too much about investing a bunch of time only to find it started overfitting your training data 32 epochs ago.

num_neurons This number we chose arbitrarily lets do a test run with 100 instead of 50.

```
num_neurons = 100

print('Build model...')
model = Sequential()

model.add(SimpleRNN(num_neurons, return_sequences=True, input_shape=( maxlen, embedding_dims )))
model.add(Dropout(.2))

model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))

model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
print(model.summary())

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test))
model_structure = model.to_json()
with open("simplernn_model2.json", "w") as json_file:
    json_file.write(model_structure)

model.save_weights("simplernn_weights2.h5")
print('Model saved.')
```

```
Using Tensorflow backend.
Build model...

Layer (type)           Output Shape        Param #
=====
simple_rnn_1 (SimpleRNN)   (None, 400, 100)      40100
dropout_1 (Dropout)       (None, 400, 100)      0
flatten_1 (Flatten)       (None, 40000)         0
dense_1 (Dense)          (None, 1)             40001
=====
Total params: 80,101.0
```

```

Trainable params: 80,101.0
Non-trainable params: 0.0

None

Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [=====] - 287s - loss: 0.9063 - acc:
0.6529 - val_loss: 0.5445 - val_acc: 0.7486
Epoch 2/2
20000/20000 [=====] - 240s - loss: 0.4760 - acc:
0.7951 - val_loss: 0.5165 - val_acc: 0.7824
Model saved.

```

Almost no difference with twice the complexity in one of the layers. This would lead one to think the model (at this point in the network) is too complex for the data. Let's try shrinking num_neurons to 25.

```

...
20000/20000 [=====] - 240s - loss: 0.5394 - acc:
0.8084 - val_loss: 0.4490 - val_acc: 0.7970
...

```

So a little better, but not noticeably. These kinds of tests can take quite a while to get used to, especially as the training time precludes the instant feedback/gratification of many coding tasks. And there are often cases where changing one parameter at a time can mask benefits you would get from adjusting two at a time, but of course this sends the complexity of the task through the roof.

TIP

Best advice, is to experiment often, and always document how the model responds to your manipulations. This kind of hands on work provides the quickest path toward an intuition that will speed your model building.

Lastly, *Dropout(percentage)* If you feel the model is overfitting the training data but shouldn't be or can't be made simpler, increasing the dropout percentage can alleviate some of that problem. Much over 50 percent and the model will start to have a difficult time learning anything, but 20% - 50% is a pretty safe realm to work in.

8.5 Predicting

Now that we have a trained model, such as it is, we can predict just as we did with the CNN in the last chapter.

```

sample_1 = "I'm hate that the dismal weather that had me down for so long, when will it
break! Ugh, when does happiness return? The sun is blinding and the puffy
clouds are too thin. I can't wait for the weekend."
from keras.models import model_from_json

```

```

with open("simplernn_model1.json", "r") as json_file:
    json_string = json_file.read()
model = model_from_json(json_string)

model.load_weights('simplernn_weights1.h5')

vec_list = tokenize_and_vectorize([(1, sample_1)]) ①

test_vec_list = pad_trunc(vec_list, maxlen) ②

test_vec = np.reshape(test_vec_list, (len(test_vec_list), maxlen, embedding_dims))

model.predict_classes(test_vec)

```

- ① We pass a dummy value in the first element of the tuple just because our helper expects it from the way processed the initial data. That value won't ever see the network, so it can be whatever.
- ② Tokenize returns a list of the data (length 1 here)

```
array([[0]], dtype=int32)
```

Negative again.

So we have another tool add to the pipeline in classifying our possible responses, and the incoming questions or searches that a user may enter. But why choose an Recurrent Neural Network (RNN)? The short answer is: don't. Well not a SimpleRNN as we have implemented here. For the simple reason it is relatively expensive to train and pass new samples through compared to a Feed Forward net or a Convolutional Neural Net and at least in this example, the results aren't appreciably better, or even better at all. So why bother with an RNN at all? Well it turns out the concept of remembering bits of input that have already occurred is absolutely crucial in Natural Language Processing. The problems of vanishing gradients are usually too much for a Recurrent Neural Net to overcome, especially in an example with so many time steps such as ours. In the next chapter we will begin to examine alternative ways *remembering*, ways that turn out to be as Andrej Karpathy puts it "unreasonably effective".¹⁴⁴

Footnote 144 Karpathy, Andrew, The Unreasonable Effectiveness of Recurrent Neural Networks.
karpathy.github.io/2015/05/21/rnn-effectiveness/

There are a few last things to mention about Recurrent Neural Networks that we did not mention in the example but are important nonetheless.

8.5.1 Statefulness

The are times when you want to remember from one input *sample* to the next, not just one time step to the next within a single sample. So what happens to that information at the end of the training step? Other than what is encoded in the weights via backpropagation, the final output has no effect and the next input will start fresh. Keras provides a keyword argument in the base RNN layer (therefore in the SimpleRNN as well) called *stateful*. It defaults to False, but if you flip this to True when adding the SimpleRNN layer to your model, the last output of the last sample will be passed into itself at the next time step along with the first token input, just as would happen in the middle of the sample.

This can be useful when we try to model a larger document or the nature of the entire corpus itself. You would not use it in an example such as the one above. As the samples are each unrelated, the order of the samples themselves here provides no new information to the system.

If the fit method is passed a *batch_size* then the statefulness of the model will hold the output of each sample in the batch and then for the 1st sample in the next batch it will pass the output of the first sample in the previous batch. 2nd to 2nd. i-th to i-th. So if you are trying to model a larger single corpus on smaller bits of the whole, paying attention to the order of the dataset will become important.

8.5.2 Two Way Street

So far we have only discussed relationships between words and what has come before. But isn't there a case to be made for flipping those word dependencies?

They wanted the pet the dog whose fur was brown.

As we get to the token fur we have encountered dog already and know something about it. But there is information in the sentence that the dog has fur and that that fur is brown, which in turn provides information back to the action of petting and the fact that it was "they" who wanted to do the petting. Humans read the sentence in one direction, but are capable of flitting back to earlier parts of the text as new information is revealed, sometimes in an inverted fashion. So it would be nice to allow our model to flit back across the input as well. That is where *bi-directional* Recurrent Neural Nets come in. Keras has recently added a layer wrapper that will automatically flip and the necessary inputs and outputs and simultaneously calculate a bi-directional RNN for us.

```

from keras.models import Sequential
from keras.layers import SimpleRNN
from keras.layers.wrappers import Bidirectional

num_neurons = 10
maxlen = 100
embedding_dims = 300

model = Sequential()
model.add(Bidirectional(SimpleRNN(num_neurons, return_sequences=True),
    input_shape=(maxlen, embedding_dims)))

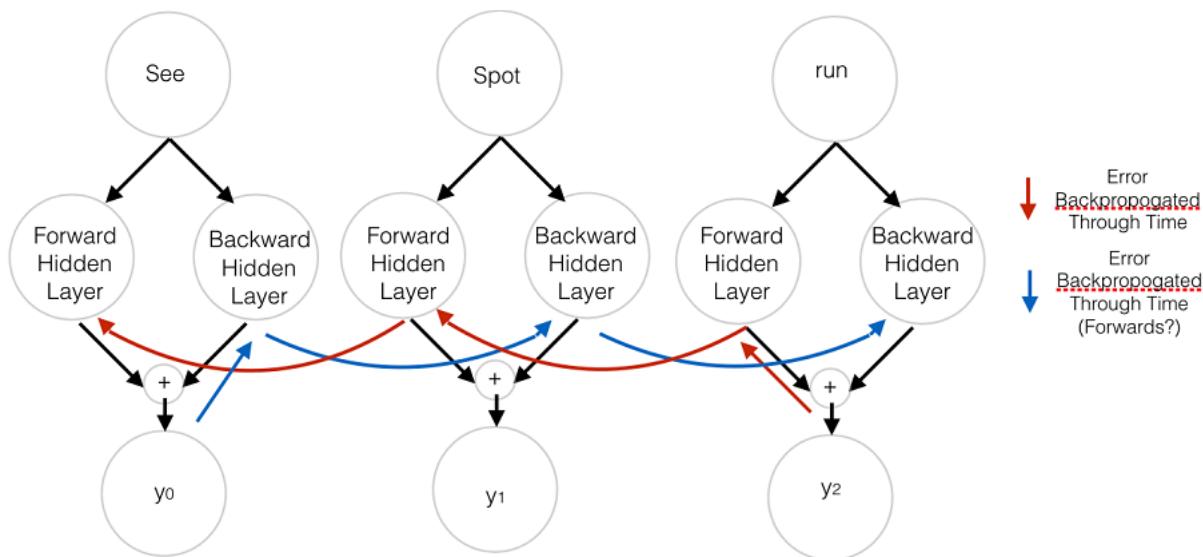
```

The basic idea is you take two RNN's right next to each other and pass the input into one as normal and pass the same input *backwards* into other net. The output of those two are then concatenated at each time step to the related time step in the other network, related being same input token. So we take the output the final time step in the input and concatenate it with the output that is generated by the same input token at the first time step of the backward net.

TIP

Keras also has a `go_backwards` keyword argument. If this is set to True, Keras will automatically flip the input sequences and input them into the network in reverse order. Basically this is just the second half of the Bidirectional layer described above.

If you are not using a Bidirectional wrapper, this can be useful as a Recurrent Neural Network (due to the vanishing gradient problem) is more receptive to data at the end of the sample than at the beginning. If you have padded your samples with `<PAD>` tokens at the end, all of the good juicy stuff is buried deep in the input loop. `go_backwards` can be a quick way around this problem.



A Bidirectional Recurrent Net

Figure 8.14 Bidirectional Recurrent Neural Net

With these tools we are well on our way to not just predicting and classifying text, but actually modeling language itself and how it is used. And with that deeper algorithmic understanding, instead of just parroting text our model has seen before, we can generate completely new text!

8.5.3 What is this thing?

Ahead of the Dense layer we have a vector that is of shape (number of neurons x 1) coming out of the last time step of the Recurrent Layer for a given input sequence. This is the parallel to the *thought vector* we got out of the Convolutional Neural Network in the previous chapter. It is an encoding of the sequence of tokens. Granted it is only going to be able to encode the thought of the sequences in relation to the labels the network is trained on. But in terms of Natural Language Processing, this is an amazing next step toward encoding higher order concepts into a vector computationally.

8.6 Summary

In this chapter we started remembering things:

- In sequence data like language, what came before is important
- Splitting single data samples along the dimension of time can lead to deeper understanding
- We can backpropagate error back in time
- Weights are adjusted adjusted in aggregate across time for a given sample
- Gradients are temperamental and may disappear or explode
- There are different ways to examine the output of Recurrent Neural Nets
- Tuning Neural Nets is hard
- We can go backward and forward in time simultaneously



Improving Retention with Long Short-Term Memory Networks (LSTMs)

In this chapter

- Adding deeper memory to Recurrent Neural Nets
- Gating information inside Neural Nets
- Classification vs. Generation
- Modeling Language Patterns

For all the benefits Recurrent Neural Nets provide for modeling relationships in time series data, and therefore *possibly* causal relationships, they suffer from one main deficiency: the *effect* of a token is almost completely lost by the time two time steps have passed. Any effect the first node will have on the third node (the 2nd time step thereafter), will be thoroughly stepped on by new data introduced in the intervening time step. This is, of course, important to the basic structure of the net, but it precludes the common case in human language that the tokens may be deeply interrelated but be separated greatly in the sentence.

Take the example:

The young woman went to the movies with her friends.

The noun "woman" immediately precedes the verb "went" and as we learn from this sentence a Recurrent Neural Net would have access to that relationship.

However, in a similar thought:

The young woman, having found a free ticket on the ground, went to the movies.

The noun and verb are no longer one time step apart in the sequence. A Recurrent Neural Net is going to have difficulty picking up on this relationship as *having* or the comma, depending on your tokenization would be the most related in the time series.

Here we ideally want to pick up on the same *thought* in imparting it to our model, but now *woman* and *went* are so far apart, the effect of *woman* in terms of the output of the time step will be almost completely washed away by the output of the net for time steps in the middle. That's not to say a net would not necessarily pick up the important information, but there is an "easier" way. What we need is a way to remember the past across the entire input. The advent of *Long Short-Term Memory* (and the closely related *Gated Recurrent Unit*) is the tool developed for just this sort of case.¹⁴⁵

Footnote 145 Hochreiter, Schmidhuber; Neural Computation 9(8):1735-80, 12/1997

9.1 LSTM

Long Short-Term Memory networks or LSTM's introduce the concept of a *state* for each layer in the Recurrent Network. The state acts as its *memory*. You can think of it as adding attributes to a class in Object-Oriented Programming. The attributes of the memory state are updated with each training example.

The magic of LSTMs lies in the fact that the rules that govern what information is stored in the state are trained neural nets themselves. They learn along with the rest of the Recurrent Net! With the introduction of a *short term memory* we can begin to learn dependencies that stretch not just one or two tokens away, but across the entirety of each data sample. With those long-term dependencies in hand we can start to see beyond the words themselves and into something deeper about language.

Patterns that humans take for granted and often process on a subconscious layer begin to be available to our model. And with that we can not only more accurately predict classifications of samples, we can actually start to generate novel text around those patterns. State of the art in this field is still far from perfect, but the results we'll see, even in our toy examples, are striking.

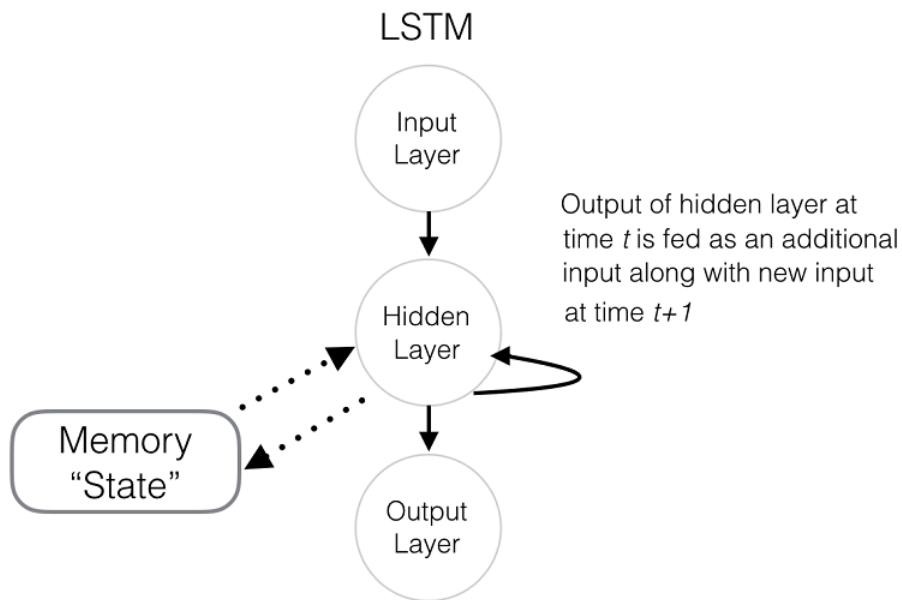
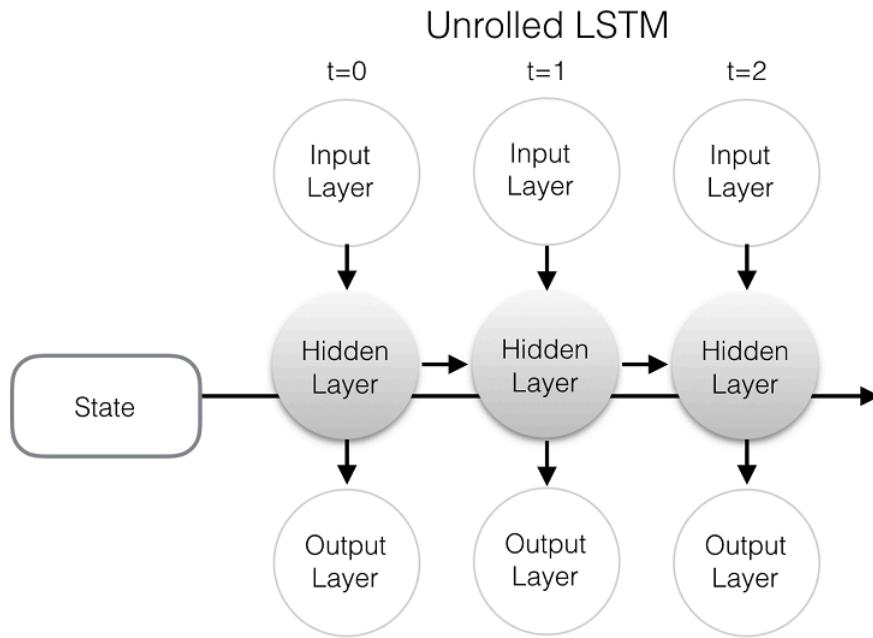


Figure 9.1 LSTM Network and its Memory

So how does this thing work?

The state is affected by the input and affects the output of the layer just like happens in a normal recurrent net but that state persists across all time steps of the time series. So each input can have an effect on the state as well as the output of the cell. The magic of the memory state comes from what the network decides to remember is *learned* along with what to output through standard backpropagation! So what does this look like?

First let's unroll a standard Recurrent Neural Net and add our memory unit.



Just like an RNN, but the hidden cells have access to read and update the state.

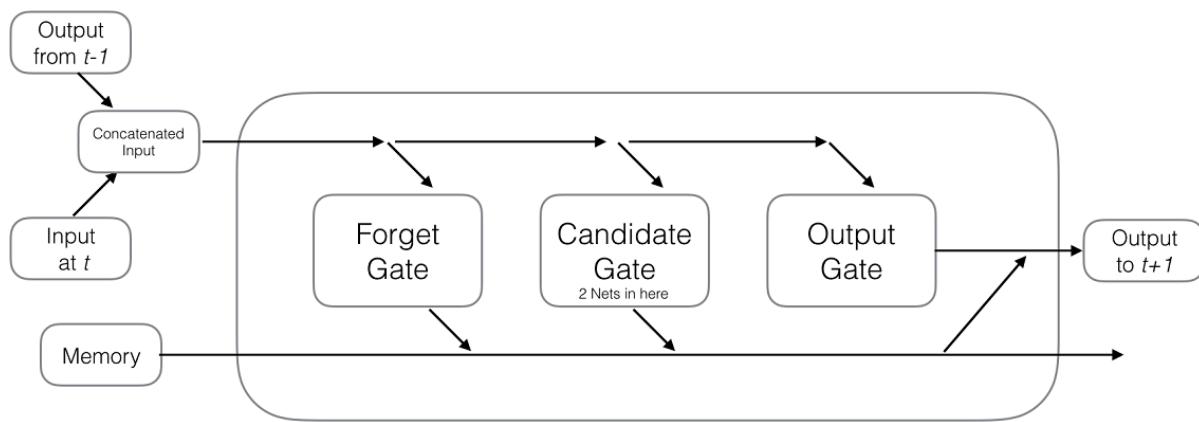
Figure 9.2 Unrolled LSTM Network and its Memory

This looks very similar to a normal Recurrent Neural Net, but in addition to the activation output feeding into next time-step version of the layer, let's add a memory state that passes through time steps of the network, so that each time step iteration has access to it. The addition of this memory unit and the mechanisms that interact with it make this a step removed from a traditional Neural Network layer, though in reality it is just a highly specialized example of one.

TIP

In much of the literature, including here, you will find this structure referred to as an **LSTM cell**. A **cell** can be composed of one or more output neurons, so the corollary to an **LSTM cell** is definitely closer to a Neural Network layer, rather than the neurons themselves.

So let's take a closer look at one of these *cells*



LSTM Layer at Time Step t

Figure 9.3 LSTM Cell

Each cell instead of being a series of weights on the input and an activation function on those weights, is now somewhat more complicated. As before the input to the layer (or cell) is a combination of the input sample and output from the previous time step. As information flows into the cell instead of a vector of weights, it is now greeted by three *gates*. A Forget gate, an Input/Candidate Gate, and a Output gate.

Each of these gates is a feed-forward network layer composed of a series of weights that the network will learn and an activation function. Technically one of the gates is composed of two feed-forward networks, so there will actually be four sets of weights to learn in this layer. The weights and activations will aim to *allow* information to flow through the cell in different amounts all in relation to the state (or memory) of the cell.

Before we get too deep in the weeds lets look at this in Python. We'll use our example from the previous chapter and swap out the SimpleRNN layer for an LSTM. You can use the same vectorized, padded/truncated processed data from the last chapter, `x_train`, `y_train`, `x_test`, `y_test`.

```
maxlen = 400
batch_size = 32
embedding_dims = 300
epochs = 2

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, LSTM

num_neurons = 50

print('Build model...')
model = Sequential()

model.add(LSTM(num_neurons, return_sequences=True, input_shape=(maxlen, embedding_dims)))
model.add(Dropout(.2))

model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))

model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
print(model.summary())
```

```
Build model...

Layer (type)          Output Shape         Param #
=====
lstm_1 (LSTM)        (None, 400, 50)      70200
dropout_1 (Dropout)   (None, 400, 50)      0
flatten_1 (Flatten)   (None, 20000)        0
dense_1 (Dense)       (None, 1)            20001
=====
Total params: 90,201.0
Trainable params: 90,201.0
Non-trainable params: 0.0
```

One import and one line of Keras code changed. But a great deal more is going on under the surface. From the summary, you can see we have many more parameters to train than we did in the SimpleRNN from last chapter for the same number of neurons (50). Recall the simple RNN had the weights:

300 (one for each element of the input vector)

1 (one for the bias term)

50 (one for each neuron's output from the previous time step)

351 total/per neuron

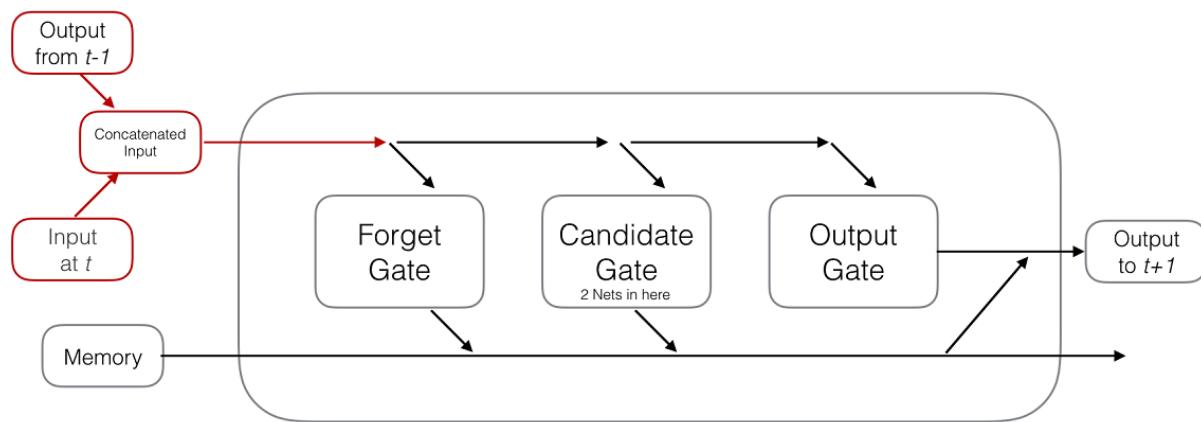
$351 * 50 = 17,550$ for the layer

As we said above, the cells have 3 gates (a total of 4 nets) which is:

$17,550 * 4 = 70,200$

But what is the memory? The memory is going to be represented by a vector that is the same number of elements as neurons in the cell. Our example has a relatively simple 50 neurons, so the memory unit will be a vector of floats that is 50 elements long.

Now what are these gates? Let's follow the first sample on its journey through the net and get an idea.



LSTM Layer at Time Step t

Figure 9.4 LSTM Cell Entry

Our "journey" through the cell is not a single road, there are branches and we will follow each for a while then back up, progress, branch, and finally come back together for the grand finale of the cell's output.

We take the first token from the first sample and pass its 300 element vector representation into the first LSTM cell. On the way into the cell, the vector representation of the data is concatenated with the vector output from the previous time step (which is a 0 vector in the first time step). In this example we will have a vector that is $300 + 50$ elements long. Sometimes you will see a 1 appended to the vector, this would correspond to the bias term. As the bias term always multiplies its associated weight by one before passing to the activation function, that input is just assumed and occasionally omitted from the input vector representation to keep the diagrams more digestible.

At the first fork in the road, we hand off a copy of the combined input vector to the ominous sounding *forget gate*.

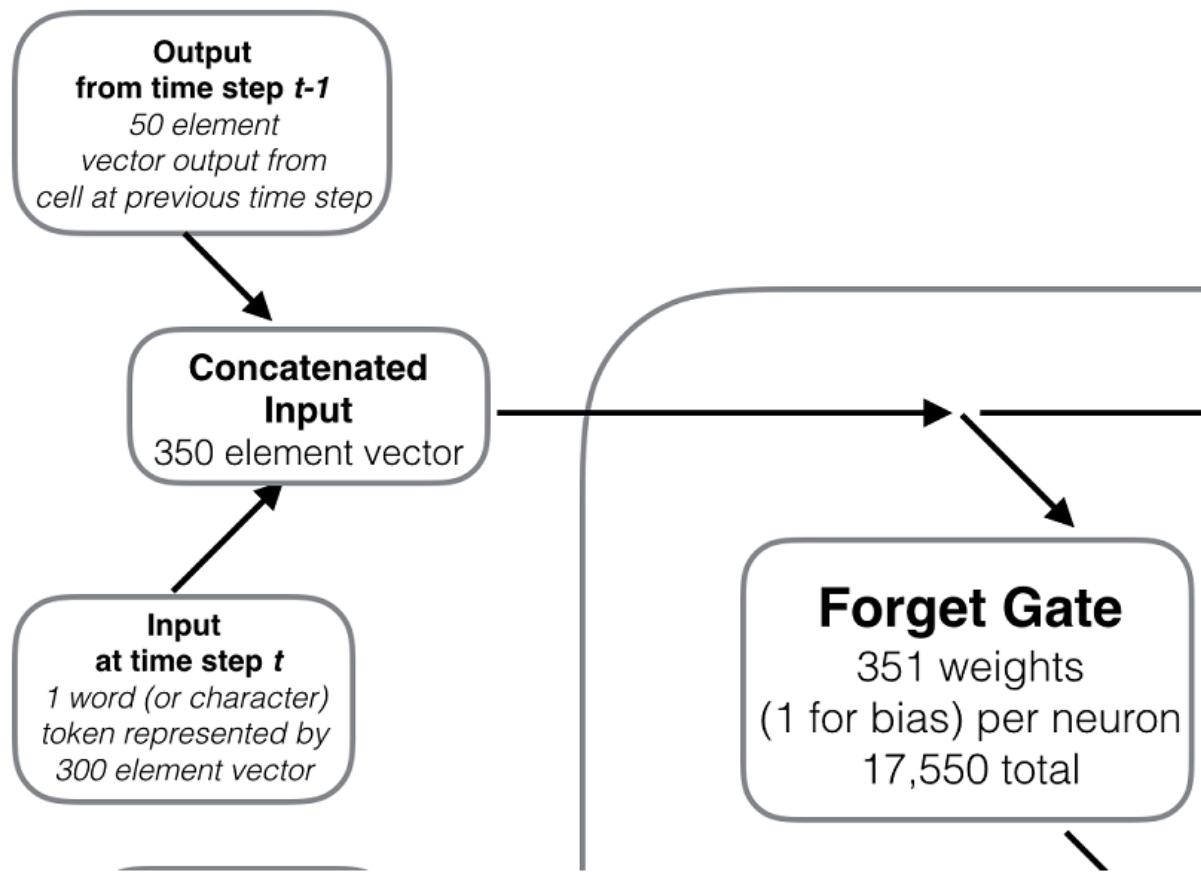


Figure 9.5 First Stop - The Forget Gate

The goal of the *forget gate* is to learn, based on a given input, how much of the cell's memory do we want erase. Whoa, wait a minute we just got this memory thing plugged in and the first thing we want to do is start erasing things? Sheesh.

The idea behind wanting to forget is just as important as wanting to remember. As we pick up certain bits of information, say whether the noun is singular or plural, or in Romance languages the gender of the noun encountered, we want to retain that information so that it can be correctly associated with the conjugation of the associated verb or constructing the proper adjective later in the sequence. But an input sequence can very easily switch from one noun to another, as an input sequence can be composed of multiple phrases/sentences/thoughts. As that happens we may want to no longer focus on the information around the first noun we came across:

A thinker sees his own actions as experiments and questions—as attempts to find out something. Success and failure are for him answers above all.

-- Friedrich Nietzsche

In this example the verb *see* is conjugated to fit with the noun *thinker*. The next active verb we come across is *to be* in the second sentence. There *be* is conjugated into are to agree with *Success and failure*. If we were to conjugate it to match the first noun we came across, *thinker*, it comes out as *is*. So we are trying to model not just long-term dependencies within a sequence but crucially forget long-term dependencies as new ones arise, hence the importance of the *forget gate*.

Now, of course, the network isn't working in these kind of explicit representations, it is only trying learn a set of weights that when applied to all of the input from a certain sequence it updates the memory and hence the output appropriately. It is remarkable that they work at all, let alone how well they work. But enough marveling, back to forgetting.

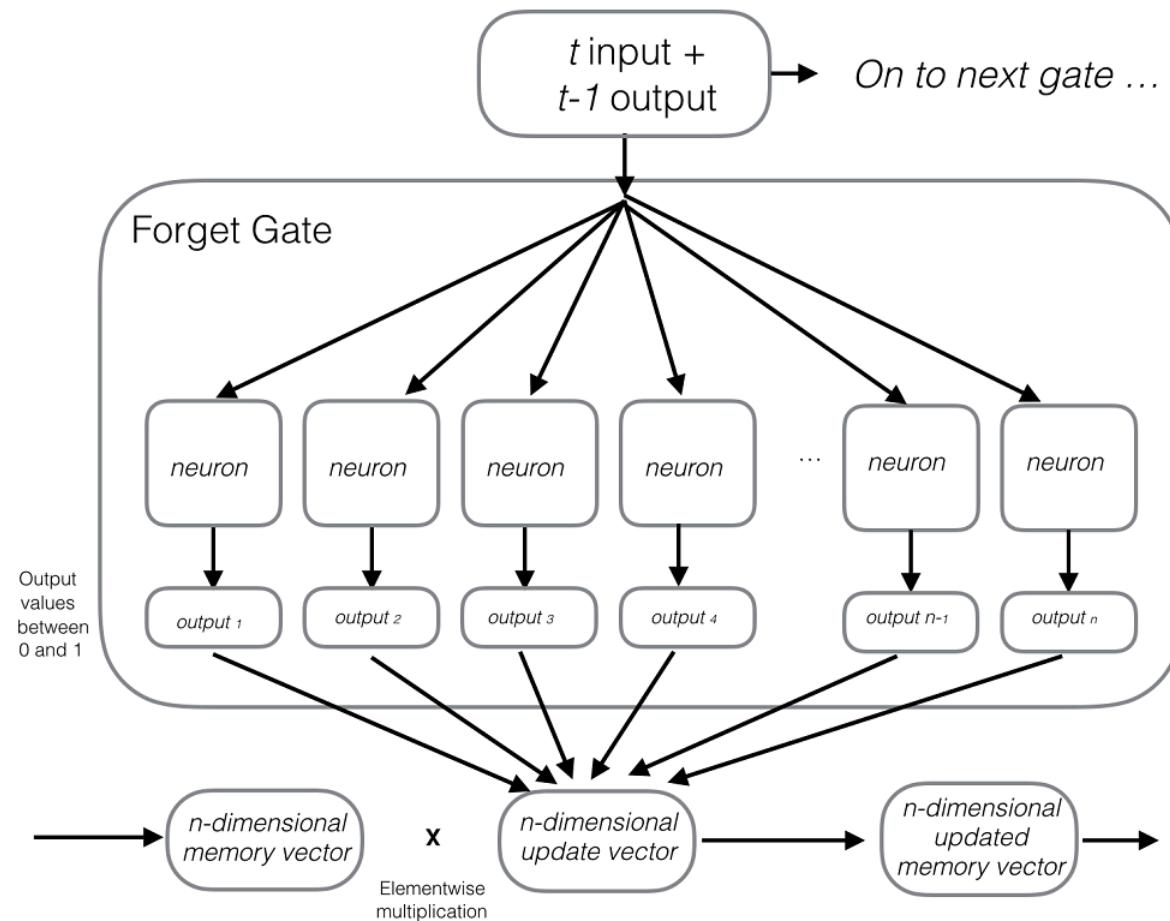


Figure 9.6 Forget Gate

The *forget gate* itself, is as we said just a feed-forward network. It consists of n neurons each with $m + n + 1$ weights. So our example *forget gate* would have 50 neurons each with 351 ($300+50+1$) weights. The activation function for a *forget gate* is the sigmoid function, as we want the output for each neuron in the gate to be between 0 and 1.

The output vector of the *forget gate* is then a "mask" of sorts, albeit a porous one that "erases" elements of the memory vector. As the *forget gate* outputs values closer to 1, more of the memory's knowledge in the associated element is retained for that time step. Closer to 0 and more of the value of that element is erased.

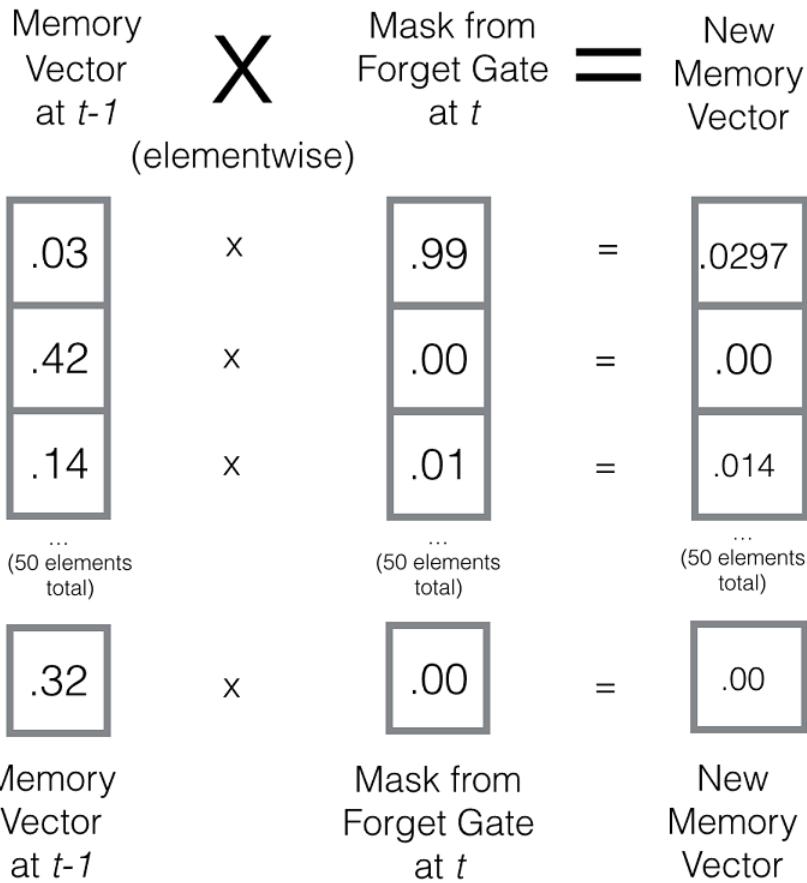


Figure 9.7 Forget Gate Application

Actively forgetting things, check. We better learn how to remember something new, or this is going to go south pretty quickly. Just like in the forget gate we use a small network to learn how much to augment the memory based on two things: the input so far and the output from the last time step. This is what happens in the next gate we branch into, the *candidate gate*.

The *candidate gate* is the gate we mentioned above that has 2 separate networks inside. The first is a net with a sigmoid activation function whose goal is to learn which values of the memory register to update. This very closely resembles the mask generated in the forget gate.

The second half of this gate determines what values we are going to update the memory with. This second part has a *tanh* activation function so each output value will be

between -1 and 1. The output of these two are multiplied together element-wise. The resulting vector from this multiplication is then added, again element-wise, to the memory register, thus remembering new details.

This gate is learning simultaneously which values to extract and the magnitude of those particular values. The mask and magnitude become what is added to the memory state. As in the forget gate, the candidate gate is learning to mask off the "inappropriate" information before adding into the cell's memory.

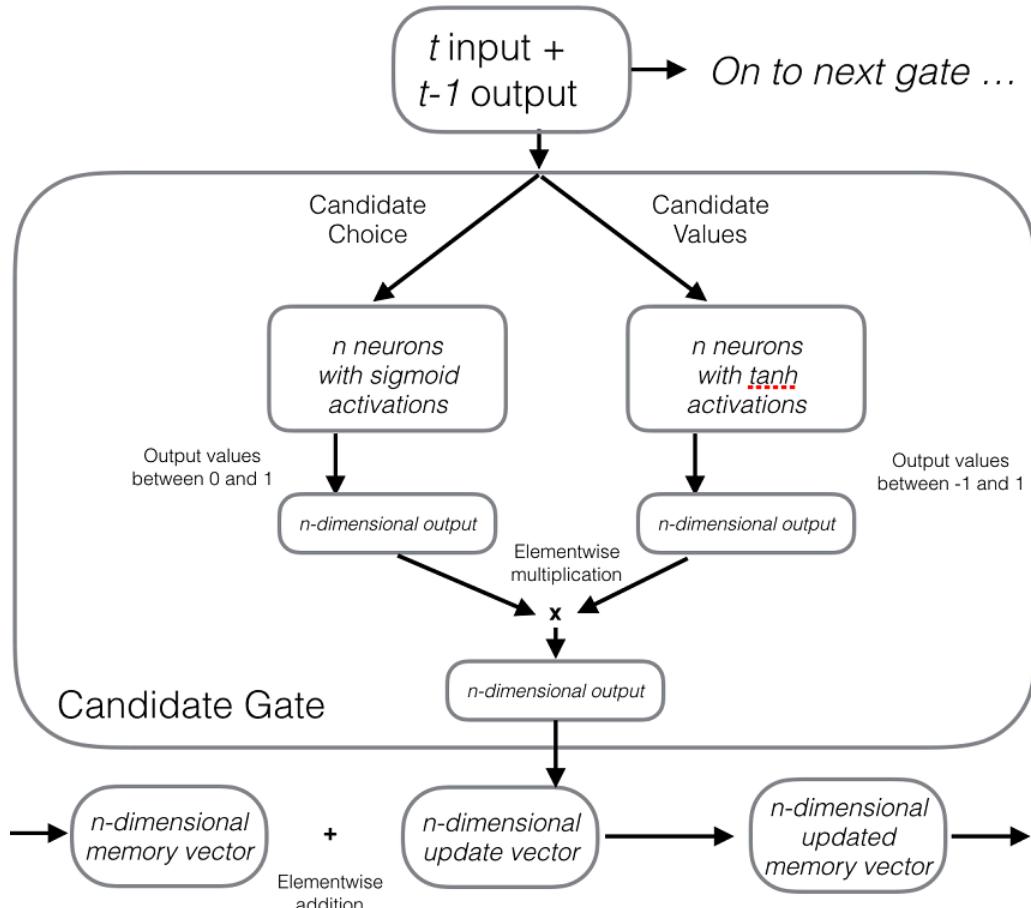


Figure 9.8 Candidate Gate

So old, seemingly irrelevant, things are forgotten and new things are remembered. Thus we arrive at the last gate of the cell; the *output gate*.

Up until this point in the journey through the cell, we have only written to the cell's memory. Now it is finally time to get some use out of this structure. The *output gate* takes the input (remember this is still the concatenated input a time step t and the output of the cell from time step $t-1$) and passes it into the output gate.

The concatenated input is passed into the weights of the n neurons then we apply a

sigmoid activation function outputting an n-dimensional vector of floats, just like the output of a simpleRNN. But instead of handing that information out through the cell wall, we pause.

The memory structure we have built up is now primed and it gets to weigh in on what we *should* output. This judgment is achieved by using the memory to create one last mask. This mask is a kind of gate as well, but we refrain from using that term explicitly as this doesn't have any learned parameters so it is important to differentiate it from the 3 previous gates described.

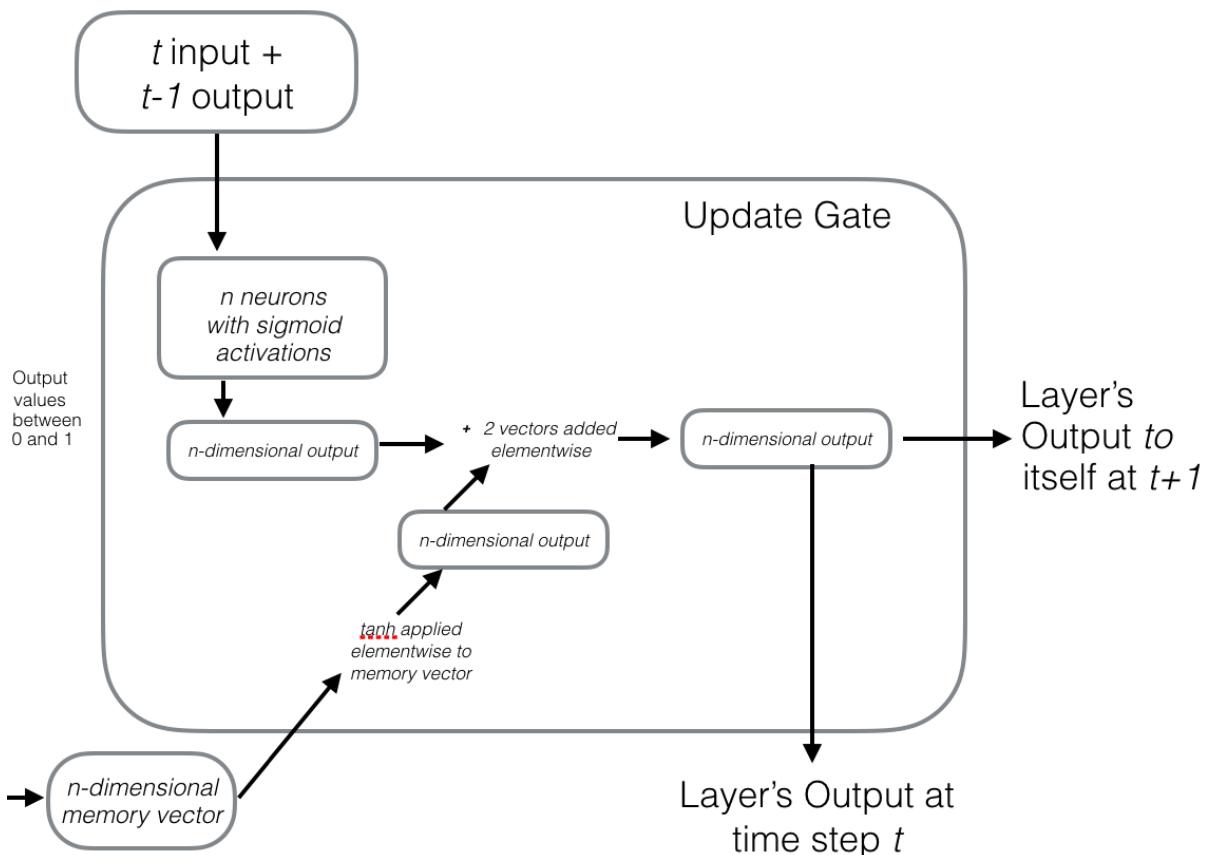


Figure 9.9 Update/Output Gate

The *mask* created from the memory is just the memory state with a tanh function applied element-wise. This gives an n-dimensional vector of floats between -1 and 1. That mask vector is then multiplied element-wise against the raw vector computed in the *output gate*'s first step. The resultant n-dimensional vector is finally passed out of the cell as the cell's official output at time step *t*.

TIP

Remember the output from an LSTM cell is just like the output from a Simple Recurrent Neural Network layer. It is passed out of the cell as the layer output (at time step *t*) and to itself as part of the input to time step *t+1*.

Thereby the memory of the cell gets the last word on what is important to output at time step t given what was input at time step t and output at $t-1$, and all the details it has gleaned up to this point in the input sequence.

9.1.1 Backpropagation Through Time

How does this thing learn then? As with any other neural net, *backpropagation*. For a moment let's step back and look at the problem we are trying to solve with this new complexity. A vanilla RNN is susceptible to a *vanishing gradient* as the derivative at any given time step is a factor of the weight themselves, so as we step back in time coalescing the various deltas, after a few iterations, the weights (and the learning rate) may shrink the gradient away to 0. The update to the weights at the end of the backpropagation (which would equate to the beginning of the sequence) are either minuscule or effectively 0. A similar problem occurs when the weights are somewhat large the gradient *explodes* and grows disproportionately to the network.

An LSTM avoids this problem via the memory state itself. The neurons in each of the gates are updated via derivatives of the functions they fed, namely those that update the Memory state on the forward pass. So at any given time step, as the normal chain rule is applied backwards to the forward propagation, the updates to the neurons are dependent on only the memory state at that time step and the previous one.

The error of the entire function is this way kept "nearer" to the neurons for each time step. This is known as the *error carousel*.

9.1.2 In Practice

How does this work in practice then? Exactly like the Simple RNN from last chapter, is the short answer. All we have changed is the inner workings of the black box that is a Recurrent Layer in the network. So we can just swap out the Keras SimpleRNN Layer for the Keras LSTM Layer and all the other pieces of our classifier will stay the same.

We'll use the same dataset, prepped the same way: Tokenize the text and embed those using word2vec. Then we will pad/truncate the sequences again to 400 tokens each using the functions we defined in the previous chapters.

```
import numpy as np

dataset = pre_process_data('./aclimdb/train')
vectorized_data = tokenize_and_vectorize(dataset)
expected = collect_expected(dataset)
```

1

```

split_point = int(len(vectorized_data)*.8)

x_train = vectorized_data[:split_point]
y_train = expected[:split_point]
x_test = vectorized_data[split_point:]
y_test = expected[split_point:]

maxlen = 400
batch_size = 32          # How many samples to show the net before backpropogating the
                        error and updating the weights
embedding_dims = 300     # Length of the token vectors we will create for passing into
                        the Convnet
epochs = 2

x_train = pad_trunc(x_train, maxlen)
x_test = pad_trunc(x_test, maxlen)

x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims))
y_train = np.array(y_train)
x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
y_test = np.array(y_test)

```

- ➊ Gather the data and prep it
- ➋ Split the data into training and testing sets
- ➌ Declare the hyperparameters
- ➍ Further prep the data by making each point of uniform length
- ➎ Reshape into a numpy data structure

Then we can build our model using the new LSTM layer.

```

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, LSTM

num_neurons = 50

print('Build model...')
model = Sequential()

model.add(LSTM(num_neurons, return_sequences=True, input_shape=(maxlen,
                                                               embedding_dims)))           ➊
model.add(Dropout(.2))

model.add(Flatten())                ➋
model.add(Dense(1, activation='sigmoid'))  ➌

model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
print(model.summary())

```

- ➊ Keras makes the implementation easy
- ➋ Stretch the output of the LSTM
- ➌ A one neuron layer that will output a float between 0 and 1

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

```
Build model...

Layer (type)          Output Shape         Param #
=====
lstm_2 (LSTM)         (None, 400, 50)      70200
dropout_2 (Dropout)   (None, 400, 50)      0
flatten_2 (Flatten)   (None, 20000)        0
dense_2 (Dense)       (None, 1)            20001
=====
Total params: 90,201.0
Trainable params: 90,201.0
Non-trainable params: 0.0
```

And train the model just as before as well.

```
model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           validation_data=(x_test, y_test))  

①  

model_structure = model.to_json() ②
with open("lstm_modell.json", "w") as json_file:
    json_file.write(model_structure)

model.save_weights("lstm_weights1.h5")
print('Model saved.')
```

① Train the model

② Save its structure so we don't have to do this part again

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [=====] - 548s - loss: 0.4772 - acc: 0.7736 - val_loss: 0.3694 - val_acc: 0.8412
Epoch 2/2
20000/20000 [=====] - 583s - loss: 0.3477 - acc: 0.8532 - val_loss: 0.3451 - val_acc: 0.8516
Model saved.
```

That is an enormous leap in the validation accuracy compared to simple RNN we implemented in the last chapter with the same dataset. You can start to see how much gain there is to be had in providing the model with a memory when the relationship of tokens is so important. And it is not just learning to remember tokens themselves, but the beauty of the algorithm is that it learns the *relationships* of the tokens it sees. The network is now able to model those relationships, specifically in the context of the cost function we provide.

In this case, how close we are to correctly identifying positive or negative sentiment. Granted this is a very narrow focus of a much grander problem within Natural Language Processing. How do you model humor, sarcasm, angst? Can they be modeled together? Definitely a field of active research for sure. But working on them separately, while demanding a lot of hand labeled data (and there is more of this out there every day) is certainly a viable path, and stacking these kinds of discrete classifiers in your pipeline is a perfectly legitimate path to pursue in a focused problem space.

9.1.3 Where does the rubber hit the road?

This is the fun part. With a trained model you can begin trying out various sample phrases and seeing how well the model performs. Try to trick it. Use happy words in a negative context. Try long phrases, short ones. Contradictory ones.

```
from keras.models import model_from_json
with open("lstm_modell.json", "r") as json_file:
    json_string = json_file.read()
model = model_from_json(json_string)

model.load_weights('lstm_weights1.h5')
```

```
sample_1 = """I'm hate that the dismal weather that had me down for so long, when will it break! Ugh, wh
vec_list = tokenize_and_vectorize([(1, sample_1)]) ①
test_vec_list = pad_trunc(vec_list, maxlen) ②
test_vec = np.reshape(test_vec_list, (len(test_vec_list), maxlen, embedding_dims))

print("Sample's sentiment, 1 - pos, 2 - neg : {}".format(model.predict_classes(test_vec)))
print("Raw output of sigmoid function: {}".format(model.predict(test_vec)))
```

- ① We pass a dummy value in the first element of the tuple just because our helper expects it from the way processed the initial data. That value won't ever see the network, so it can be whatever.
- ② Tokenize returns a list of the data (length 1 here)

```
1/1 [=====] - 0s
Sample's sentiment, 1 - pos, 2 - neg : [[0]]
Raw output of sigmoid function: [[ 0.2192785]]
```

As you play with the possibilities watch the raw output of the sigmoid in addition to the specific sentiment classification. Unlike the binary classification, the *predict* method, using the sigmoid activation, is a continuous function and all outputs will be between 0 and 1. Anything above 0.5 will be classified as positive, below 0.5 will be negative. As

you try your samples you will get a sense of how confident the model is in its prediction. This can be helpful in parsing results of your spot checks. It may mis-classify something that intuitively seems obvious, but if it is hovering around 0.5 it is closer to random to chance that the prediction comes out one way or the other and you can then look at why that phrase is ambiguous to the model but not to you as a human. This in itself is good feedback about personal judgments in the context of such a large problem space as NLP.

9.1.4 Dirty Data

With this more powerful model there are still a great number of hyperparameters to toy with. But this is a good time to pause and look back to the beginning to our data. We have been using the same data, processed in exactly the same way since we started with Convolutional Neural Nets. This was done specifically to see the variations in the types of models and their performance on a given dataset. But we did make some choices that compromised the integrity of the data, *dirtied* it if you will.

Padding or truncating each sample to 400 tokens was important for Convolutional Nets so that the filters would be "scanning" the same amount of data and therefore outputting the same amount of data. This is important as it goes into a feed-forward layer at the end of the chain and those require vectors of fixed size. Similarly, our implementations of Recurrent Neural Nets, both simple and LSTM, are striving toward a *thought vector* that we can pass into a feed-forward layer for classification. So that the *thought vector* is of consistent size we have to *unroll* the net a consistent number of time steps. Let's look at our choice of 400 as then number of time steps to unroll the net.

```
def test_len(data, maxlen):
    total_len = truncated = exact = padded = 0
    for sample in data:
        total_len += len(sample)
        if len(sample) > maxlen:
            truncated += 1
        elif len(sample) < maxlen:
            padded += 1
        else:
            exact += 1
    print('Padded: {}'.format(padded))
    print('Equal: {}'.format(exact))
    print('Truncated: {}'.format(truncated))
    print('Avg length: {}'.format(total_len/len(data)))

dataset = pre_process_data('./aclimdb/train')
vectorized_data = tokenize_and_vectorize(dataset)
test_len(vectorized_data, 400)
```

```
Padded: 22559
Equal: 12
Truncated: 2429
Avg length: 202.4424
```

Whoa. Okay, 400 was a bit on the high side. Probably should have done this analysis earlier. Let's dial the *maxlen* back to something closer to average the sample size. Say 200 tokens, and give our LSTM another crack at it.

```

import numpy as np

from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, LSTM

maxlen = 200
batch_size = 32
embedding_dims = 300
①
epochs = 2

dataset = pre_process_data('./aclimdb/train')
vectorized_data = tokenize_and_vectorize(dataset)
expected = collect_expected(dataset)

split_point = int(len(vectorized_data)*.8)

x_train = vectorized_data[:split_point]
y_train = expected[:split_point]
x_test = vectorized_data[split_point:]
y_test = expected[split_point:]

x_train = pad_trunc(x_train, maxlen)
x_test = pad_trunc(x_test, maxlen)

x_train = np.reshape(x_train, (len(x_train), maxlen, embedding_dims))
y_train = np.array(y_train)
x_test = np.reshape(x_test, (len(x_test), maxlen, embedding_dims))
y_test = np.array(y_test)

num_neurons = 50

print('Build model...')
model = Sequential()

model.add(LSTM(num_neurons, return_sequences=True, input_shape=(maxlen, embedding_dims)))
model.add(Dropout(.2))

model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))

model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
print(model.summary())

```

① All the same code as above but we are limiting the the max length to 200 tokens

```

Build model...

Layer (type)          Output Shape         Param #
=====
lstm_1 (LSTM)        (None, 200, 50)      70200
dropout_1 (Dropout)   (None, 200, 50)      0
flatten_1 (Flatten)   (None, 10000)        0

```

```

dense_1 (Dense)           (None, 1)           10001
=====
Total params: 80,201.0
Trainable params: 80,201.0
Non-trainable params: 0.0
=====
```

```

model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test))
model_structure = model.to_json()
with open("lstm_model7.json", "w") as json_file:
    json_file.write(model_structure)

model.save_weights("lstm_weights7.h5")
print('Model saved.')
```

```

Train on 20000 samples, validate on 5000 samples
Epoch 1/2
20000/20000 [=====] - 245s - loss: 0.4742 - acc: 0.7760 - val_loss: 0.4235 - val_acc: 0.8010
Epoch 2/2
20000/20000 [=====] - 203s - loss: 0.3718 - acc: 0.8386 - val_loss: 0.3499 - val_acc: 0.8450
Model saved.
```

Well that trained much faster. With samples that were half the number of tokens, we cut the training time in half! More than half. There were half the LSTM time steps to compute and half the weights in the feed-forward layer to learn. But most importantly the backpropagation only had to travel half the distance (half the time steps back into the past) each time. But we got almost no improvement in the accuracy scores. Why would that be? With all the power of neural nets and their ability to learn complex patterns, it is easy to forget a neural net is in most cases just as good at learning to discard noise. We had inadvertently introduced a lot of noise into the data by appending all of those zero vectors. The bias elements in each node will still give it some signal even if all of the input is zero. But the net will eventually learn to disregard those elements entirely (specifically by adjusting the weight on that bias element down to zero) and focus on the portions of the samples that contain meaningful information.

So we didn't learn that much more, but we cut computation time. The most important take away from this though is to be aware of the length of our test samples in relation to length of the training sample lengths. If your training set is composed of documents thousands of tokens long, you may not get a very accurate classification of something only 3 tokens long padded out to 1000. And vice versa, cutting a 1,000 token opus to 3 tokens will severely hinder your poor, little model. Not that an LSTM won't make a good go of it, just a note of caution as you experiment.

9.1.5 Back to Our Dirty Data

What is arguably our greater sin in data handling: dropping the "unknown" tokens on the floor. The list of "unknowns", which is basically just words we couldn't find in the pre-trained word2vec model is quite extensive. By dropping this much data on the floor, especially when attempting to model the sequence of words is very problematic.

Sentences like:

"I don't like this movie."

may become:

"I like this movie."

if our token "don't" doesn't appear in the word2vec model. This isn't actually the case of our specific word2vec model, but there are many tokens that are skipped and they may or may not be relevant. Dropping these unknown tokens is one strategy you can pursue, but there are others. You can of course use or train a word2vec model that has a vector for every last one of your tokens, but this is almost always prohibitively expensive.

There are two more commonplace approaches that provide decent results without exploding the computational requirements. Both involve replacing the unknown token with a new vector representation. The first approach is very counter-intuitive, and that is: for every token not modeled by a vector, randomly select a vector from the existing model and use that instead. You can easily see how this would flummox a human reader.

A sentence like:

"The man who was defenestrated, brushed himself off with a nonchalant glance back inside."

may become:

"The man who was duck, brushed himself off with a airplane glance back inside."

How on earth is a model to learn from nonsense like this? It turns out, the model does overcome these hiccups in much the same way our example did when we dropped them on the floor. Remember we are not trying model every statement in the training set explicitly. The goal is to create a generalized model of the language in the training set. So

outliers will exist, but hopefully not so much as to derail the model in describing the prevailing patterns.

The second and more common approach is to replace all tokens not in the word vector library with a specific token, usually referenced as "UNK" for unknown when reconstructing the original input. The vector itself is chosen either when modeling the original embedding or at random (and ideally far away from the known vectors in the space).

As with padding, the network can learn its way around these unknown tokens and come to its own conclusions around them.

9.1.6 Words are hard. Letters are easier.

Words have meaning, we can all agree on this. It only seems natural then to model natural language with these basic building blocks. As it also seems natural, to use these models to describe meaning, feeling, intent, and everything else in terms of these atomic structures. But, of course, words aren't atomic at all. As we saw earlier, they are made up of smaller words, stems, phonemes. But they are also even more fundamentally, a sequence of characters.

As we are modeling language, there is good deal that lies all the way back at the character level. Intonations in voice, alliteration, rhymes all of this can be modeled if we break things down all the way down to the character level. They can be modeled by humans without breaking things down that far too, but the definitions that would arise from that modeling are fraught with complexity and not easily imparted to a machine, which after all is why we are here. A great deal of those patterns are inherent in text when you examine it with an eye toward which character came after which, given the characters we've already seen.

A space or a comma or a period all become just another character in this paradigm. And as our network is learning meaning from sequences if we break it down all the way to the individual characters themselves, the model is forced to find these lower level patterns. To notice a repeated suffix after a certain number of syllables, which would quite probably rhyme may be a pattern that carries meaning, perhaps joviality or derision. With a large enough training set, these patterns begin to emerge.

And there's not so many letters to deal with! So we have less variety of input vectors to deal with.

Training a model at the character level is tricky though. The patterns and long term dependencies found at the character level can vary greatly across voices. These patterns can be found but they may not generalize as well. Let's try our LSTM at the character level on the same example dataset. First we need to process the data differently.

As before we grab the data and sort out the labels:

```
dataset = pre_process_data('./aclimdb/train')
expected = collect_expected(dataset)
```

We then need to decide how far to unroll the network, so we'll see how many characters on average are in the data samples.

```
def avg_len(data):
    total_len = 0
    for sample in data:
        total_len += len(sample[1])
    return total_len/len(data)

print(avg_len(dataset))
```

```
1325.06964
```

So immediately we can see the network is going to be unrolled much farther. And we are going to be waiting a significant amount of time for this model to finish. Spoiler: this model doesn't do much other than over-fit, but it provides an interesting example nonetheless.

Next we need to clean the data of tokens that aren't related to the natural language in the text. This function cheats in that there are html tags in the dataset, so really the data should be more thoroughly scrubbed.

```
def clean_data(data):
    """ Shift to lower case, replace unknowns with UNK, and listify """
    new_data = []
    VALID = 'abcdefghijklmnopqrstuvwxyz0123456789\'?!.,: '
    for sample in data:
        new_sample = []
        for char in sample[1].lower(): # Just grab the string, not the label
            if char in VALID:
                new_sample.append(char)
            else:
                new_sample.append('UNK')

        new_data.append(new_sample)
    return new_data

listified_data = clean_data(dataset)
```

We are using the 'UNK' as single character in the list for everything that doesn't match the VALID list.

Then, as before we pad or truncate the samples to a given *maxlen*. Here we introduce another "single character" for padding: "PAD"

```
def char_pad_trunc(data, maxlen):
    """ We truncate to maxlen or add in PAD tokens """
    new_dataset = []
    for sample in data:
        if len(sample) > maxlen:
            new_data = sample[:maxlen]
        elif len(sample) < maxlen:
            pads = maxlen - len(sample)
            new_data = sample + ['PAD'] * pads
        else:
            new_data = sample
        new_dataset.append(new_data)
    return new_dataset

maxlen = 1500
```

We chose *maxlen* of 1500 to capture slightly more data than was in the average sample, but we tried to avoid introducing too much noise with PADs. It can be helpful to think about these choices in the sizes of words. At a fixed character length, a sample with lots of long words could be under-sampled, compared to a sample composed entirely of simple one syllable words. As with any machine learning problem it is important to know your dataset and its ins and outs.

This time instead of using *word2vec* for our embeddings we are going to one-hot encode the characters. So we need to create a dictionary of the tokens (our characters) mapped to an integer index. We'll also create a dictionary to map the reverse as well, but more on that later.

```
def create_dicts(data):
    """ Modified from Keras LSTM example"""
    chars = set()
    for sample in data:
        chars.update(set(sample))
    char_indices = dict((c, i) for i, c in enumerate(chars))
    indices_char = dict((i, c) for i, c in enumerate(chars))
    return char_indices, indices_char
```

And then we can use that dictionary to create input vectors of the indices instead of the tokens themselves.

```
import numpy as np

def onehot_encode(dataset, char_indices, maxlen):
```

```

"""
One hot encode the tokens

Args:
    dataset  list of lists of tokens
    char_indices  dictionary of {key=character, value=index to use encoding vector}
    maxlen  int  Length of each sample
Return:
    np array of shape (samples, tokens, encoding length)
"""

X = np.zeros((len(dataset), maxlen, len(char_indices.keys())))
for i, sentence in enumerate(dataset):
    for t, char in enumerate(sentence):
        X[i, t, char_indices[char]] = 1
    return X
1

```

- ① A numpy array of length equal to number of data samples, each sample will be a number of tokens equal to max_len, and each token will be a one hot encoded vector of length equal to the number of characters

```

dataset = pre_process_data('./aclimdb/train')
expected = collect_expected(dataset)
listified_data = clean_data(dataset)

maxlen = 1500
common_length_data = char_pad_trunc(listified_data, maxlen)

char_indices, indices_char = create_dicts(common_length_data)
encoded_data = onehot_encode(common_length_data, char_indices, maxlen)

```

And then we split up our data just like before.

```

split_point = int(len(encoded_data)*.8)

x_train = encoded_data[:split_point]
y_train = expected[:split_point]
x_test = encoded_data[split_point:]
y_test = expected[split_point:]

```

```

from keras.models import Sequential
from keras.layers import Dense, Dropout, Embedding, Flatten, LSTM

num_neurons = 40

print('Build model...')
model = Sequential()

model.add(LSTM(num_neurons, return_sequences=True, input_shape=(maxlen,
    len(char_indices.keys()))))
model.add(Dropout(.2))

model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))

model.compile('rmsprop', 'binary_crossentropy', metrics=['accuracy'])
print(model.summary())

```

```
Build model...
```

| Layer (type) | Output Shape | Param # |
|-----------------------|------------------|---------|
| <hr/> | | |
| lstm_2 (LSTM) | (None, 1500, 40) | 13920 |
| dropout_2 (Dropout) | (None, 1500, 40) | 0 |
| flatten_2 (Flatten) | (None, 60000) | 0 |
| dense_2 (Dense) | (None, 1) | 60001 |
| <hr/> | | |
| Total params: | 73,921.0 | |
| Trainable params: | 73,921.0 | |
| Non-trainable params: | 0.0 | |

```
batch_size = 32
epochs = 10
model.fit(x_train, y_train,
           batch_size=batch_size,
           epochs=epochs,
           validation_data=(x_test, y_test))
model_structure = model.to_json()
with open("char_lstm_model3.json", "w") as json_file:
    json_file.write(model_structure)

model.save_weights("char_lstm_weights3.h5")
print('Model saved.')
```

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [=====] - 634s - loss: 0.6949 - acc: 0.5388 - val_loss: 0.6775 - val_acc: 0.5738
Epoch 2/10
20000/20000 [=====] - 668s - loss: 0.6087 - acc: 0.6700 - val_loss: 0.6786 - val_acc: 0.5962
Epoch 3/10
20000/20000 [=====] - 695s - loss: 0.5358 - acc: 0.7356 - val_loss: 0.7182 - val_acc: 0.5786
Epoch 4/10
20000/20000 [=====] - 686s - loss: 0.4662 - acc: 0.7832 - val_loss: 0.7605 - val_acc: 0.5836
Epoch 5/10
20000/20000 [=====] - 694s - loss: 0.4062 - acc: 0.8206 - val_loss: 0.8099 - val_acc: 0.5852
Epoch 6/10
20000/20000 [=====] - 694s - loss: 0.3550 - acc: 0.8448 - val_loss: 0.8851 - val_acc: 0.5842
Epoch 7/10
20000/20000 [=====] - 645s - loss: 0.3058 - acc: 0.8705 - val_loss: 0.9598 - val_acc: 0.5930
Epoch 8/10
20000/20000 [=====] - 684s - loss: 0.2643 - acc: 0.8911 - val_loss: 1.0366 - val_acc: 0.5888
Epoch 9/10
20000/20000 [=====] - 671s - loss: 0.2304 - acc: 0.9055 - val_loss: 1.1323 - val_acc: 0.5914
Epoch 10/10
20000/20000 [=====] - 663s - loss: 0.2035 - acc: 0.9181 - val_loss: 1.2051 - val_acc: 0.5948
Model saved.
```

Clear evidence of overfitting. The model started to learn the sentiment of the training set slowly. Oh so slowly. This took over 1.5 hours on a modern laptop without a GPU. But the validation accuracy never improved much above a random guess and later in the epochs it started to get worse, which you can also see in the validation loss.

There are lots of things that could be going on here. The model could be too rich for the dataset. By that we mean it has enough parameters that it can begin to model patterns that are unique to the 20,000 samples in the training set, but aren't really applicable to a broader definition of language. One might alleviate this with a higher dropout percentage or less neurons in the LSTM layer. More labeled data would also help if you think the model is defined to "richly", but that is usually the hardest piece to come by.

But in the end this is creating a great deal of expense for both your hardware and your time for limited reward based on where we got with a word-level LSTM model and even the Convolutional Neural Nets in the previous chapters. So why bother with the character level at all? It turns out that the character-level model can be extremely good at modeling the language itself given a broad enough example. Or model a specific kind of language given a focused enough training set, say from one author instead of thousands. Either way this is a first step toward generating novel text with a Neural Net.

9.1.7 My Turn to Talk

If we could generate new text with a certain "style" or "attitude" we certainly have an entertaining chatbot indeed. Of course, being able to generate novel text with a given style doesn't guarantee your bot will talk about what you want it to talk about. But this approach can be used to generate lots of text within a given set of parameters, in response to a user's style for example, and this larger corpus of novel text could then be indexed and searched as possible responses to a given query.

Much like a Markov Chain which predicts the next word in a sequence based on the probability of any given word appearing after the 1-gram or 2-gram or n-gram that just occurred, our LSTM model can learn the probability of the next word based on what it just saw, but with the added benefit of *memory*! A Markov chain only has information about the n-gram it is using to search and the frequency of words that occur after that n-gram. The RNN model does something very similar in that it encodes information about the next term based on the few that preceded it. But with the LSTM memory state, the model has a greater context in which to judge the most appropriate next term. And most excitingly, we can predict the next character based on the characters that came before. This level of granularity is beyond a basic Markov Chain.

How do we train our model to do this magic trick? First we are going to abandon our classification task. The real core of what the LSTM is learned in in the LSTM cell itself. But as earlier we were using the models successes and failures around a specific classification task to learn that is not necessarily modeling a more general representation of language in general. So instead of using the sentiment label of the training set as the target for learning, we use the training samples themselves!

For each character in the sample we want to learn to *predict* the next character.

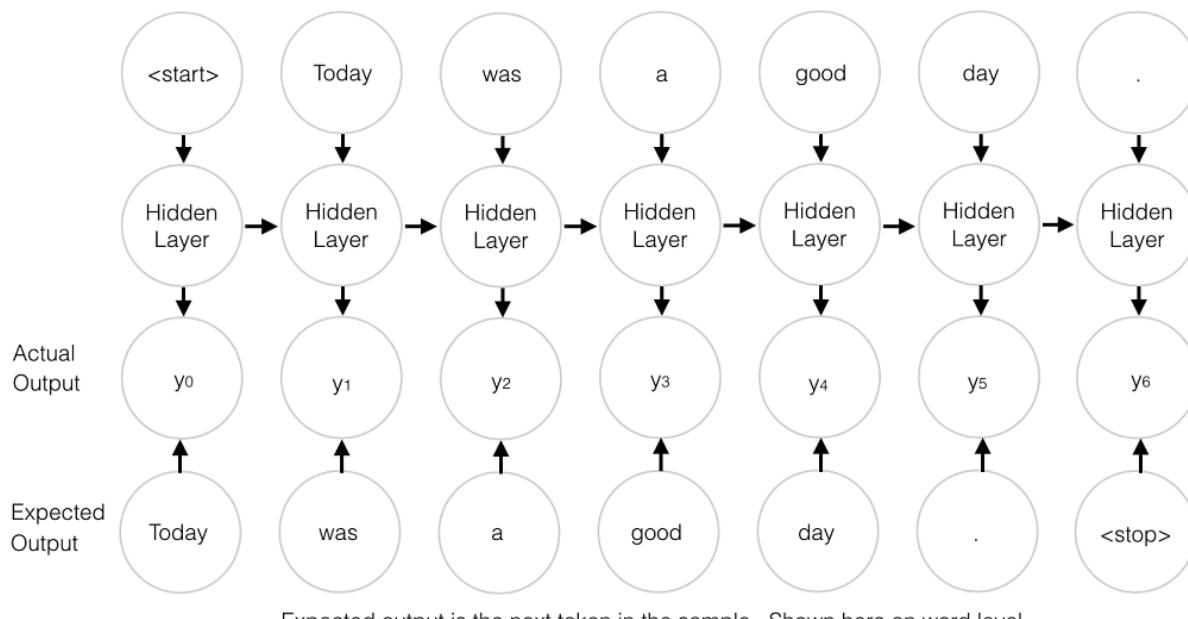


Figure 9.10 Next Word Prediction

This can work on the word level, but we are going to cut to the chase and go straight down to the character level with the same concept.

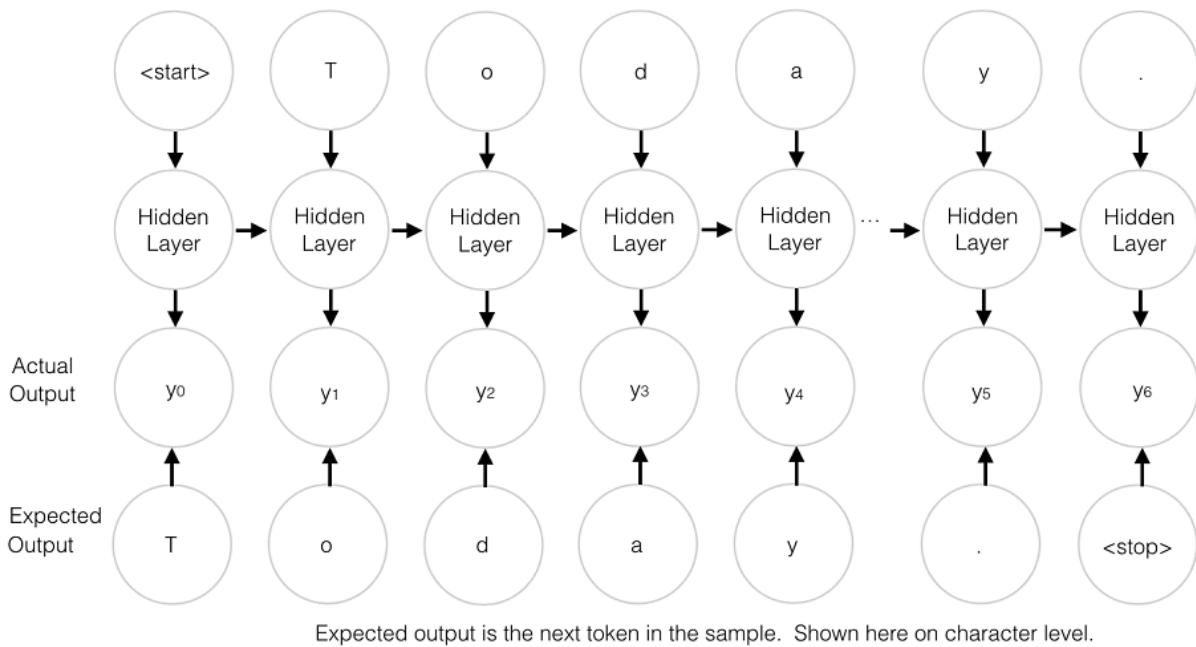


Figure 9.11 Next Character Prediction

Instead of a thought vector coming out of the last time step we are going to focus on the output of each time step individually. The error will still backpropagate through time from each time step back to the beginning but the error is determined specifically at the time step level. In a sense it was in our classifier above as well, but in the classifier the error wasn't determined until the end of the sequence and there was an aggregated output to feed into the feed-forward layer at the end of the chain. A minor point but can be helpful to make a richer mental model of what is going on, and it does not change how the backpropagation is aggregated and the error then applied to the weights only after the entire sequence has passed into the network.

So the first thing we need to do is adjust our training set labels. The output vector will be measured not against a given classification label but against the one-hot encoding of the next character in the sequence.

We can also fall back to a simpler model and instead of trying to predict every next character, just predict the next character for a given sequence. This is exactly the same as our classifier above if we drop the keyword argument `return_sequences=True` and just focus on the return value of the last time step in the sequence.

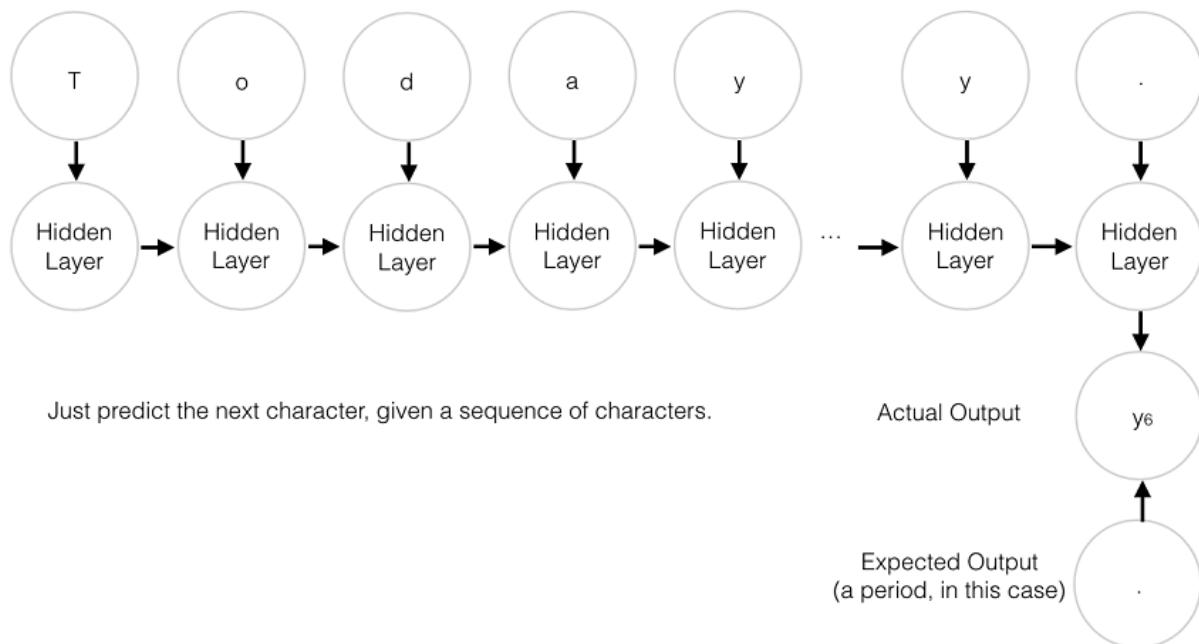


Figure 9.12 Last Character Prediction Only

9.1.8 My Turn to Speak More Clearly

Simple character level modeling like this is the gateway to more complex models ones that can pick up on not only details such as spelling, but grammar and punctuation. The real magic of these models comes we get to learn these details but start to pick up the rhythms and cadences of text as well. Let's take a tour on how we can start to generate some novel text with the tools we were using for classification so far.

The Keras documentation provides an excellent example of just this. For this project we are going to set aside the movie review dataset we have used up to this point. For finding concepts as deep as tone and word choice, that dataset has two attributes that are difficult to overcome. First of all, it is very diverse. As it is written by many writers each with their own patterns of speech finding commonalities across them all is difficult. Not to say such a task is intractable as a large enough dataset will reveal commonalities across large numbers of styles. But that leads us to the second attribute which is: it is an extremely small dataset for this kind of task. As either a dataset that is more consistent across samples in style an tone or a much larger dataset will overcome this problem, we'll choose the former. The Keras example provides a sample of the work of Friedrich Nietzsche. That's fun but we'll choose someone else with a singular style, William Shakespeare. He hasn't published anything in a while, so let's help him out.

```
from nltk.corpus import gutenberg
print(gutenberg.fileids())
```

```
[ 'austen-emma.txt', 'austen-persuasion.txt', 'austen-sense.txt', 'bible-kjv.txt',
  'blake-poems.txt', 'bryant-stories.txt', 'burgess-busterbrown.txt', 'carroll-alice.txt',
  'chesterton-ball.txt', 'chesterton-brown.txt', 'chesterton-thursday.txt',
  'edgeworth-parents.txt', 'melville-moby_dick.txt', 'milton-paradise.txt',
  'shakespeare-caesar.txt', 'shakespeare-hamlet.txt', 'shakespeare-macbeth.txt',
  'whitman-leaves.txt' ]
```

Ah, 3 plays by Shakespeare. We'll grab those and as in the Keras example concatenate them into a large string.

```
text = ''
for txt in gutenberg.fileids():
    if 'shakespeare' in txt:
        text += gutenberg.raw(txt).lower()

print('corpus length:', len(text))

chars = sorted(list(set(text)))
print('total chars:', len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars)) ②
indices_char = dict((i, c) for i, c in enumerate(chars)) ③
```

- ① Concatenate all Shakespeare plays in the Gutenberg corpus in NLTK
- ② Make a dictionary of characters to an index, for reference in the one-hot encoding
- ③ Make the opposite dictionary for lookup when interpreting the one-hot encoding back to the character

```
corpus length: 375542
total chars: 50
```

This is nicely formatted as well.

```
print(text[:500])

[the tragedie of julius caesar by william shakespeare 1599]

actus primus. scoena prima.

enter flauius, murellus, and certaine commoners ouer the stage.

    flauius. hence: home you idle creatures, get you home:
is this a holiday? what, know you not
(being mechanicall) you ought not walke
vpon a labouring day, without the signe
of your profession? speake, what trade art thou?
    car. why sir, a carpenter

    mur. where is thy leather apron, and thy rule?
what dost thou with thy best apparrell on
```

Next we are going to chop the source text up into data samples, each of a fixed, *maxlen*, set of characters. To increase our dataset size and focus on consistent patterns the Keras example *oversamples* the data into semi-redundant chunks. Take 40 characters from the beginning move to the third character from the beginning and take 40 from there, move to sixth ... and so on.

Remember the goal of this particular model is to learn to predict the 41st character given what came before. Specifically, given the 40 characters that came before.

```
# cut the text in semi-redundant sequences of maxlen characters
maxlen = 40
step = 3
sentences = []
next_chars = []

for i in range(0, len(text) - maxlen, step): ❶
    sentences.append(text[i: i + maxlen]) ❷
    next_chars.append(text[i + maxlen]) ❸
print('nb sequences:', len(sentences))
```

- ❶ Step by 3 characters, so the generated training samples will overlap, but not be identical
- ❷ Grab a slice of the text
- ❸ Collect the next expected character

```
nb sequences: 125168
```

So we have 125,168 training samples and the character that follows each of them.

```
print('Vectorization...')
X = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        X[i, t, char_indices[char]] = 1
        y[i, char_indices[next_chars[i]]] = 1
```

We then one-hot encode each character of each sample in the dataset and store it as the list *X*. We also store the list of one-hot encoded "answers" in the list *y*.

```
Vectorization...
```

We then construct the model ...

```

from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.layers import LSTM
from keras.optimizers import RMSprop

# build the model: a single LSTM
print('Build model...')
model = Sequential()

model.add(LSTM(128, input_shape=(maxlen, len(chars)))) ①
model.add(Dense(len(chars)))
model.add(Activation('softmax')) ②

optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)

print(model.summary())

```

- ① We use a much wider LSTM layer. 128, up from 50. And we don't return the sequence. We just want the last output character.
- ② This is a classification problem, so we want a probability distribution over all possible characters

```

Build model...

Layer (type)          Output Shape         Param #
=====
lstm_1 (LSTM)        (None, 128)          91648
dense_1 (Dense)      (None, 50)           6450
activation_1 (Activation) (None, 50)          0
=====
Total params: 98,098.0
Trainable params: 98,098.0
Non-trainable params: 0.0
=====
```

This looks slightly different from before, so let's look at the components. Sequential and LSTM layers we know, same as before with our classifier and in this case the num_neurons is 128 in the hidden layer of the LSTM cell. 128 is quite a few more than we used in the classifier, but we are trying to model much more complex behavior in reproducing the tone of a given text. Next the optimizer is defined in a variable, but this is the same one we have used up until this point. It is just broken out here for readability purposes as the learning rate parameter is being adjusted from its default (.001 normally). For what it's worth RMSProp works by updating each weight by adjusting the learning rate with "a running average of the magnitudes of recent gradients for that weight"¹⁴⁶. Reading up on optimizers can definitely save you some heartache in your experiments, but the details of each individual optimizer is beyond the scope of this book.

Footnote 146 Hinton, et al, www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

The next big difference is the loss function we want to minimize. Up until now it has been "binary_crossentropy". We were only trying to determine the level at which one single neuron was firing. But here we have swapped out Dense(1) for Dense(len(chars)) in the last layer. So the output of the network at each time step will be a 50 dimensional vector ($\text{len}(\text{char}) == 50$, from above). We are using *softmax* as the activation function, so the output vector will be the equivalent of a probability distribution over the entire 50-dim vector. i.e. the sum of the values in the vector will always add up to one. *categorical_crossentropy* will attempt to minimize the difference between the resultant probability distribution and the one-hot encoded expected character.

And the last major change is, no *dropout*. As we are looking to specifically model this dataset, we have no interest in generalizing to other problems so not only is over-fitting okay, it is ideal.

```
epochs = 6
batch_size = 128

model_structure = model.to_json()
with open("shakes_lstm_model.json", "w") as json_file:
    json_file.write(model_structure)

❶ for i in range(5):
    model.fit(X, y,
               batch_size=batch_size,
               epochs=epochs)

    model.save_weights("shakes_lstm_weights_{}.h5".format(i+1))
    print('Model saved.'')
```

- ❶ This is one way to train the model for a while, save its state, then continue training. Keras also has callback function built in that do similar tasks when called

```
Epoch 1/6
125168/125168 [=====] - 266s - loss: 2.0310
Epoch 2/6
125168/125168 [=====] - 257s - loss: 1.6851
...

```

This will save the model every 6 epochs and keep training. If it stops reducing the loss, further training is no longer worth the cycles, so you can safely stop the process and have a saved weight set within a few epochs. We found it takes 20 to 30 epochs to start to get something decent from this dataset. You can look to expanding the dataset, as Shakespeare's works are readily available in the public domain, just be sure and strive for consistency by appropriately preprocessing if you get them from disparate sources.

Lets make our own play! Since the output vectors are 50 dimensional vectors describing a probability distribution over the 50 possible output characters we can sample from that distribution. The Keras example has a helper function to do just that.

```
import random

def sample(preds, temperature=1.0):
    # helper function to sample an index from a probability array
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Since the last layer in the network is a *softmax*, the output vector will be probability distribution over all possible outputs of the network. By looking at the highest value in the output vector, we can see what the network thinks has the highest probability of being the next character. In explicit terms, the index of the output vector with the highest value (which will be between 0 and 1) will correlate with the index of the one-hot encoding of the expected token.

But here we aren't looking to exactly recreate what the input text was just what is likely to come next. Just as in a Markov chain, the next token is selected randomly based on the probability of the next token, not the most commonly occurring next token.

The effect of dividing the log by the temperature is flattening ($\text{temperature} > 1$) or sharpening ($\text{temperature} < 1$) the probability distribution. So a *temperature* (or *diversity* in the calling arguments) less than 1 will tend toward a more strict attempt to recreate the original text, while a temp greater than 1 will produce a more diverse outcome, but as the distribution flattens the patterns learned begin to wash away and we tend back toward nonsense. Higher diversities are fun to play with though.

The numpy random function *multinomial(num_samples, probabilities_list, size)* will make *num_samples* from the distribution whose possible outcomes are described by *probabilities_list* and output a list of length *size*, which is equal to the number of times the experiment is run. So in this case we will draw once from the probability distribution, 1 time.

When we go to predict below, the Keras example has us cycle through various different values for the temperature, for each prediction will see a range of different outputs based on the temperature used in the *sample* function to sample from the probability distribution.

```

import sys

start_index = random.randint(0, len(text) - maxlen - 1)

for diversity in [0.2, 0.5, 1.0]:
    print()
    print('----- diversity:', diversity)

    generated = ''
    sentence = text[start_index: start_index + maxlen] ①
    generated += sentence
    print('----- Generating with seed: "' + sentence + '"')
    sys.stdout.write(generated)

    for i in range(400):
        x = np.zeros((1, maxlen, len(chars)))
        for t, char in enumerate(sentence):
            x[0, t, char_indices[char]] = 1.

        preds = model.predict(x, verbose=0)[0] ②
        next_index = sample(preds, diversity)
        next_char = indices_char[next_index] ③

        generated += next_char
        sentence = sentence[1:] + next_char ④

        sys.stdout.write(next_char)
        sys.stdout.flush()

print()

```

- ① We seed the trained network and see what it spits out as the next character
- ② Model makes a prediction
- ③ Look up which character that index represents
- ④ Add it to the "seed" and drop the first character to keep the length the same. This is now the seed for the next pass

(Diversity 1.2 from the example was removed for brevity's sake, but feel free to add it back in and play with the output.)

We are taking a random chunk of 40 (*maxlen*) characters from the source and predicting what will come next character by character. We then append that to the input sentence, drop the first character and predict again on that new subset of 40. Each time we write out the console the predicted letter and flush() just so we don't get a newline.

And what do we get? Something like:

```

----- diversity: 0.2
----- Generating with seed: " them through & through
the most fond an"
them through & through
the most fond and stranger the straite to the straite
him a father the world, and the straite:

```

the straite is the straite to the common'd,
 and the truth, and the truth, and the capitoll,
 and stay the compurse of the true then the dead and the colours,
 and the companyed the straite the straite
 the mildiaus, and the straite of the bones,
 and what is the common the bell to the straite
 the straite in the commised and

----- diversity: 0.5
 ----- Generating with seed: " them through & through
 the most fond an"
 them through & through
 the most fond and the pindage it at them for
 that i shall proud-be be the house, not that we be not the selfe,
 and thri's the bate and the perpaine, to depart of the father now
 but ore night in a laid of the haid, and there is it

bru. what shall greefe vndernight of it

cassi. what shall the straite, and perfire the peace,
 and defear'd and soule me to me a ration,
 and we will steele the words them with th

----- diversity: 1.0
 ----- Generating with seed: " them through & through
 the most fond an"
 them through & through
 the most fond and boy'd report alone

yp. it best we will st of me at that come sleepe.
 but you yet it enemy wrong, 'twas sir

ham. the pirey too me, it let you?
 son. oh a do a sorrall you. that makino
 beendumons vp?x, let vs cassa,
 yet his miltrow addome knowlmy in his windher,
 a vertues. hoie sleepe, or strong a strong at it
 mades manish swill about a time shall trages,
 and follow. more. heere shall abo

Diversity 0.2 and 0.5 both give us something that looks much like Shakespeare at first glance. Diversity 1.0 (given this dataset) things start to go off the rails fairly quickly, but notable that some basic structures, such as the line break followed by a character's abbreviated name still show up. All and all not bad for a relatively simple model, and definitely something you can have fun with generating text for a given style.

9.1.9 Learned How to Say, but not yet What.

So we are actually generating novel text based solely on example text. And from that we are learning to pick up style. But, and this is somewhat counterintuitive, we have no control of what is being said. The context is limited to the source data, as that will limit its vocabulary if nothing else. But given an input, we can train toward what we think the original author or authors would say, but the best we can really hope for from this level of model is *how* they would say it. And specifically how they would finish saying what was started with a specific seed sentence. That sentence by no means has to come from the text itself. Since the model is trained on characters themselves, novel words can be used as the seed and interesting results still be produced in many cases. Now we have fodder for an entertaining chatbot. But to have our bot say something in a style *and* of substance we'll have to wait until the next chapter.

9.1.10 Other Kinds of Memory

LSTM's are an extension of the basic concepts of a Recurrent Neural Net and there are a variety of other extensions in the same vein. All of them are slight variations on the number or operations of the gates inside the cell. The *Gated Recurrent Unit* (GRU) for example combines the forget gate and the candidate choice branch from the candidate gate into a single *update* gate. This saves on the number of parameters to learn and has been shown to be comparable to a standard LSTM while being that much less computationally expensive. Keras provides a GRU layer abstraction that you can implement just as with LSTMs.

```
from keras.models import Sequential
from keras.layers import GRU

model = Sequential()
model.add(GRU(num_neurons, return_sequences=True, input_shape=x[0].shape))
```

Another technique is to use an LSTM with *peephole* connections. Keras does not have a direct implementation of this but there are several examples on the web of extending the Keras LSTM class to do this. The idea is that each gate in a standard LSTM cell has access to the current memory state directly, taken in as part of its input. As described in the paper Learning Precise Timing with LSTM Recurrent Networks¹⁴⁷, the gates contain additional weights of the same dimension as the memory state. The input to each gate is then a concatenation of the input to the cell at that time step and the output of the cell from the previous time step *and* the memory state itself. The authors found more precise

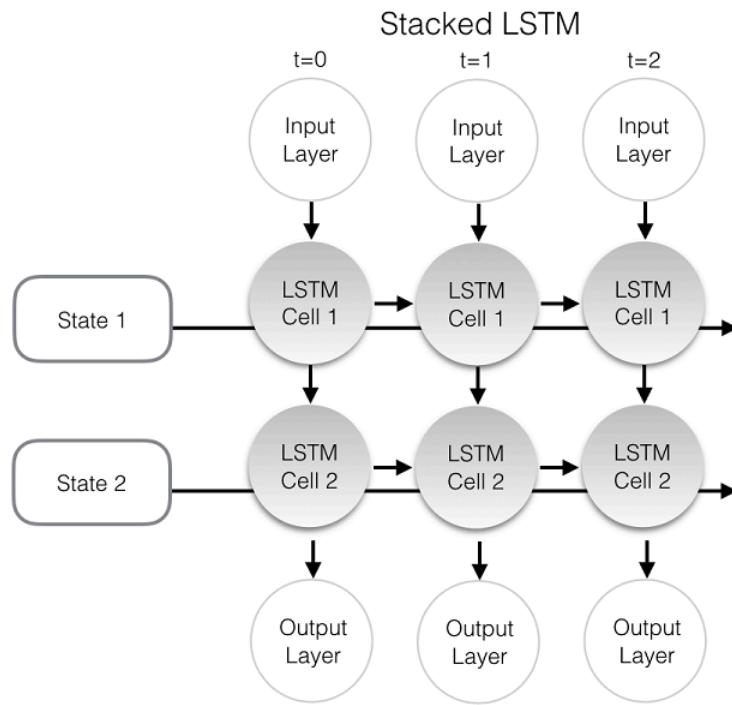
modeling of the timing of events in time series data. While they were not working specifically in the NLP domain, the concept has validity here as well, but we leave it to the reader to experiment with that.

Footnote 147 Gers, Schraudolph, Schmidhuber; www.jmlr.org/papers/volume3/gers02a/gers02a.pdf

Those are just two of the RNN/LSTM derivatives out there, experiments are ever ongoing, we encourage you join the fun. The tools are all readily available so finding the next newest greatest iteration is in the reach of all.

9.1.11 Going Deeper

While it is convenient to think of the memory unit as encoding a specific representations of noun/verb pairs or sentence to sentence verb tense references, that is not specifically what is going on. It is just a happy byproduct of the patterns that the network learns, assuming the training went well. Like any neural network, *layering* allows the model to form more complex representations of the patterns in the training data. And we can just as easily stack LSTM layers.



Each LSTM layer is a cell with its own gates and state vector.

Figure 9.13 Stacked LSTM

This is, of course, that much more computationally expensive. But easily implemented in Keras:

```
from keras.models import Sequential
from keras.layers import LSTM

model = Sequential()
model.add(LSTM(num_neurons, return_sequences=True, input_shape=X[0].shape))
model.add(LSTM(num_neurons_2, return_sequences=True))
```

Note the parameter `return_sequences=True` is required in the first and intermediary layers for the model to build correctly. This makes sense as the output at each time step is needed as the input for the time steps of the next layer.

Remember, however, that creating a model that is capable of representing more complex relationships than is actually present in the training data can lead to strange results. Simply piling layers onto model, while fun, is rarely the answer to building the most representative model.

9.1.12 Summary

In this chapter

- Remembering information in sequential inputs is possible
- It's important to forget information that is no longer relevant
- Only some the new information needs to be retained for the upcoming input
- All of the "rules" around remembering and forgetting can be learned
- If we can predict what comes next, we can generate novel text from probabilities

Sequence to Sequence Models and Attention (Generative Models)

10

In this chapter

- Mapping relationships amongst text
- Using NLP for translation
- Developing language models for chat
- Paying Attention to what we hear

Over the course of the book, we have introduced common Natural Language Processing tools which are key to understanding texts, or to classifying their sentiments or their content with a machine. But wouldn't it be nice to train a neural network to respond to a human's questions? Or to train a network to translate documents from English to German? The last few chapters have equipped you with the tools and concepts to develop applications to do just that. In this chapter, we want to tie all the previous chapters together and walk you through the process of developing your a chatbot using a neural network. But first, we need to introduce one last concept: *sequence-to-sequence networks*.

The ideas of pairing a question and a response and the pairing of a statement in one language with its equivalent translated into another language are very closely related. If we were to build a machine that could capture the *semantic* meaning of a statement and capture it in a numerical representation, a *thought vector*, we would just need to construct a machine that could produce a statement in the new language from that thought vector. Well we have a path toward creating the thought vector but we have not yet explored a way to create natural language from that vector in another language space.

You can think of the question/response pairs and the language translation as the same

problem. Given a set of tokens, there is one response in the other problem space (be it another language or another human in the same language) that you would expect. Once you have a set of these pairs, we can string some of the earlier concepts and get our translation machine, or in our example, one version of a chatbot.

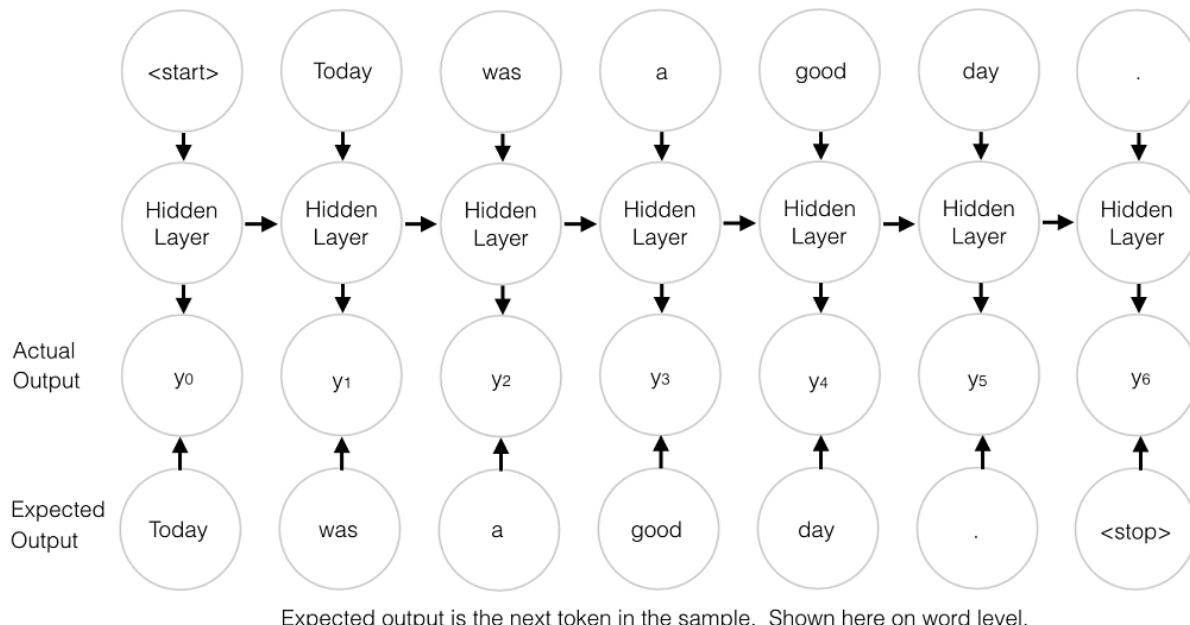
Sequence-to-sequence networks have two components. The first component, the *sequence encoder*, is a network which turns text into a vector representation, our *thought vector*. This part is very similar to the examples we saw in Chapter 8 and 9. We could take the vector and classify for example the text's sentiment. But instead of doing that final classification we will pass the thought vector to a second component, the *sequence decoder*. And this is where some deep learning magic happens. A sequence decoder can turn a vector back into human readable text again. The resulting text can be the answer to a question or the German translation of an English input text. We are using *supervised training* to teach a neural network to produce the second half of a pair, given the first. But those pairs do not have the same length! They do not have share any kind of word order. They only have to share some unifying concept, the *thought vector*.

NOTE

If you are new to Recurrent Neural Networks or Long-Short-Term-Memory networks, we encourage you to checkout Chapter 8 and 9, which will provide you with all the necessary background information about *cell states*, *forget* and *update gates*, etc.

In the last chapter, we greatly extended the complexity of our classification model by giving the layers of a Recurrent Neural Network a more complex way to remember what they had already seen. Even more importantly, we introduced a way for the network to use that information as it sees the next piece of input. The input at each time step affects the memory via the *forget* and *update* gates, but the output of the network at that time step is dictated not solely by the input but by a combination of the input *and* the current state of the memory unit.

We then backed off of the final classification and instead tried to learn to predict the next token at a given time step leveraging the information gathered in the memory unit up to that point.



Expected output is the next token in the sample. Shown here on word level.

Figure 10.1 Next Word Prediction

With a token by token prediction, we were able to generate some text, by randomly selecting the next token based on the probability distribution of likely next tokens suggested by the network. Not perfect by any stretch but entertaining nonetheless. But we aren't here for mere entertainment, we would like to have some control over what came out of a generative model.

These two implementations of LSTM's have been combined in a very powerful way. Sutskever, Vinyals, and Le¹⁴⁸ proposed leveraging the classification aspect of the LSTM to create a *thought vector* and then use that generated vector as the input to a second *different* LSTM that only tries to predict token by token. And thus we have a way to map a *sequence* of input to a distinct sequence of output. Let's take a look at how it works.

Footnote 148 Sutskever, Vinyals, and Le; arXiv:1409.3215

10.1 Sequence-to-Sequence Networks

Imagine you would like develop a translation model to translate texts from English to German. Basically, we would like to map sequences of characters or embedded words to another sequence of characters or words. We previously discovered how we can predict an element at time step t based on the previous element at the time step $t-1$. However, directly using an LSTM to map from one language to another runs into problems very quickly. One of the biggest limitations is that the input and output sequences would need to have the same sequence lengths. If you think about language translation, you can easily imagine that the likelihood that a sequence length in one language will match the sequence length in another language is fairly low.

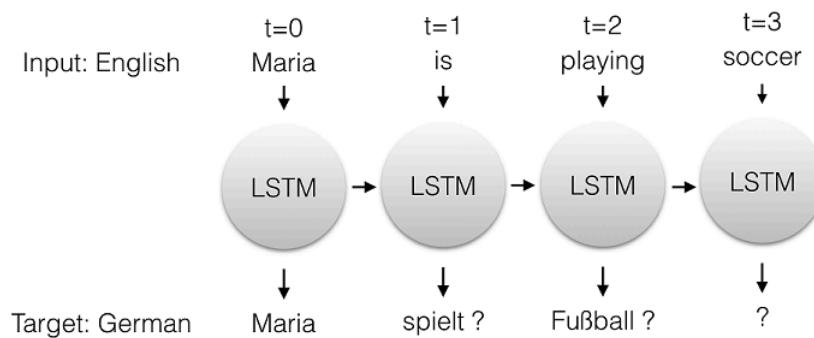


Figure 10.2 Limitations of language modeling

Figure 2 demonstrates the problem. The English and the German sentence have different lengths, which complicates the mapping between the English input and the expected output. The English phrase "is playing" (present progressive) is translated to the German present tense "spielt". But "spielt" here would need to be predicted solely on the input of "is", we haven't gotten to "playing" yet at that time step. Further "playing" would then need to map to "FuBball". Certainly a network could learn these mappings, but the learned representations would have to be hyper-specific to the input and our dream more generalized *language modelling* would go out the window.

Sequence-to-Sequence networks, often abbreviated with *seq2seq*, solve this limitation by creating an input representation in the form of a *thought vector*, and then using that thought (or sometimes called context) vector as a starting point to a second network that receives a different set of inputs to generate the output sequence.

NOTE

What is a thought vector?

Remember when we discovered word vectors and how similar terms were in close proximity in the vector space? A thought vector is very similar to a word vector. The network will find a representation with a fixed length to represent the content of the input. Instead of primarily using the vectors to do vector algebra (e.g. king + man - woman = queen), thought vectors are used to determine a numerical representation of a thought of any input length. The term was coined by Geoffrey Hinton in a talk to the Royal Society in London in 2015.

A sequence-to-sequence network consists of two distinct recurrent neural networks.

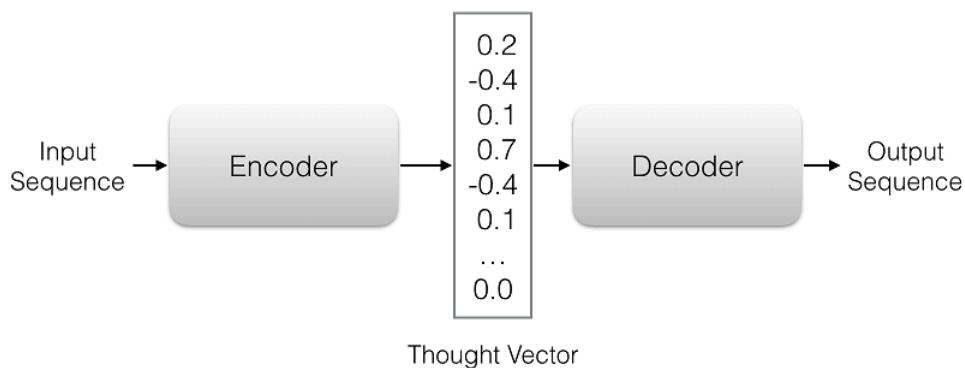


Figure 10.3 Encoder-Decoder Relationship through the Thought Vector

The first network, called the encoder, turns the input text (e.g. a user question) into the thought vector. The thought vector is encompassed in 2 parts, each a vector, the final output of the hidden layer of the encoder and the memory state of the LSTM cell for that input example.

TIP

In the Keras code below, these are captured in the variable names `h_state` (output of the hidden layer) and `c_state` (the memory state).

This thought vector then becomes the input to a second network, the decoder network. As we will see later in the implementation section, the generated state aka thought vector will serve as *initial state* of the decoder network. The second network then uses that initial state and a special kind of input, a *start token*. Primed with the initial state, the second network given a *start token* has to learn to generate the first element of the target sequence (e.g. a character or word).

The training and inference stages are treated very differently in the particular setup. During training we pass the starting text to the encoder and the *expected* text as the input

the decoder. We are getting the decoder network to learn that given a primed state and a key to "get started" it should produce a series of tokens. The first direct input to the decoder will be the start token, the second input should be the first expected token, which should in turn prompt the network to produce the second expected token.

At inference time, however, we don't have the expected text, so what do we use to pass into the decoder other than the state? We use the generic start token and then take the first generated element which will then become the input to the decoder at the next time step to generate the next element, and so on. This process repeats until the maximum number of sequence elements is reached or an *end-of-sequence* token is generated.

Trained end-to-end this way the decoder will turn a thought vector into a fully decoded response to the initial input sequence (e.g. the user question). Splitting the solution into two networks with the thought vector as the binding piece in between allows us to map input sequences to output sequences of different lengths and is irrelevant to word order.

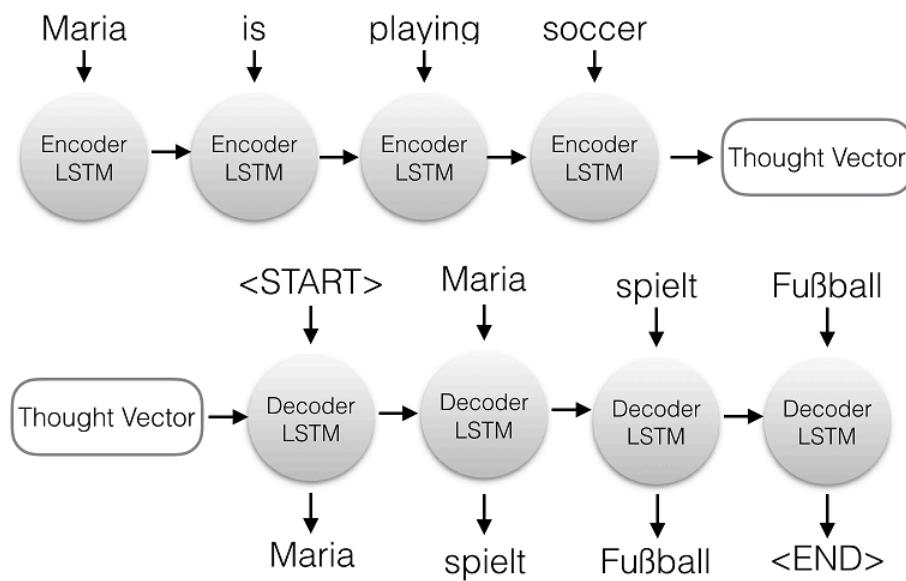


Figure 10.4 Unrolled Encoder-Decoder

NOTE

Isn't that similar to an Autoencoder?

If a seq2seq network takes a sequence, encodes it into a thought vector and then decodes it into an output sequence. Isn't that the concept of an autoencoder? While the structure looks very similar, autoencoders have a different purpose. Their purpose is to find the most optimal vector representation of input data, such that it can be reconstructed by the networks decoder with the lowest loss. Aside from autoencoders with variational noise (a variation of autoencoders), the network's purpose is to find a dense vector representation of the input data (e.g. an image or text) which allows the decoder to reconstruct it with the smallest error. During the training phase, the input data and the expected output are the same. Therefore, if your goal isn't generating thought vectors to translate between languages or to find responses for a given question, but to find a dense vector representation of your data, an autoencoder can be a good option.

10.2 How are seq2seq networks implemented?

With our knowledge from the previous chapters, we can now implement a sequence-to-sequence network. Let's step through the details step by step.

10.2.1 Preparing your dataset for the sequence-to-sequence training

As we have seen in previous implementations of convolutional or recurrent neural networks, the input data needs to be padded to a fixed length. Usually, you would extend the input sequences to match the longest input sequence with pad tokens. In the case of the sequence-to-sequence network, you also need to prepare your target data and pad it to match the longest target sequence. Remember, the dimensions of the input and target data doesn't need to be the same. Also, your output data should be extended by a `<START>` and `<STOP>` token to tell the decoder when the job starts and when it is done.

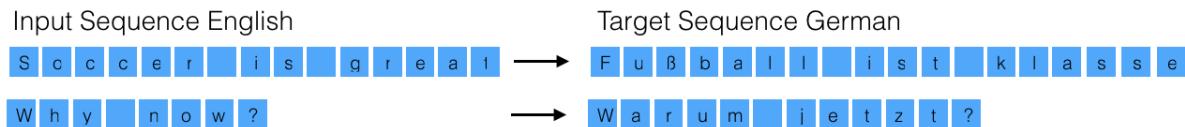


Figure 10.5 Input and Target Sequence before the Preprocessing

In addition to the required padding, the output sequence should be annotated with the `<START>` and `<STOP>` token, seen in Figure 6.

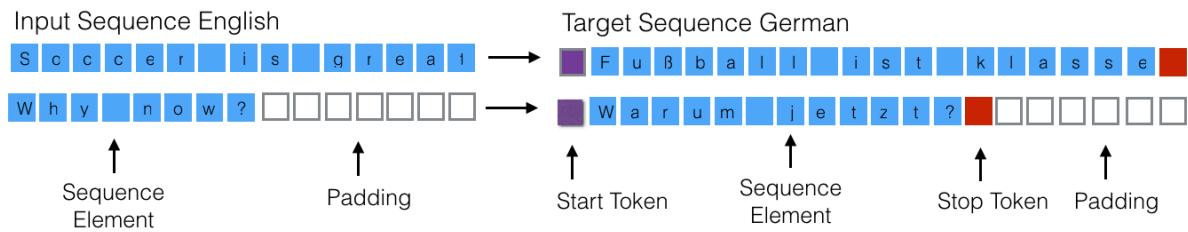


Figure 10.6 Input and Target Sequence after the Preprocessing

We will demonstrate how to annotate the target sequences in the implementation later in the chapter. Just keep in mind we will need two versions of the target sequence for training. One that starts with the start token (which we will use for the input to the decoder) and one that starts without the start token (The actual target sequence that the loss function will score).

In earlier chapters, our training sets consisted of pairs, and input and an expected output. Each training example for the sequence to sequence model will be a triplet: initial input, expected output (prepended by a start token), expected output (without the start token).

Before we get into the implementation details, let's recap for a moment. Our sequence-to-sequence network consists of two networks. First the encoder, which will generate our thought vector. Second, a decoder, that we'll pass the thought vector into, as its initial state. With the initialized state and a start token as input to the decoder network, we will then generate the first sequence element (e.g. a character or word vector) of the output. Each following element will then be predicted based on the updated state and the next element in the *expected* sequence. This process will go on until we either generate a STOP token or we reach the maximum number of elements. All sequence elements generated by the decoder will form our predicted output (e.g. our reply to a user question). With this in mind, let's take a look at the details.

10.2.2 Sequence-to-Sequence in Keras

In the following sections, we would like to guide you through a Keras implementation of a sequence-to-sequence network published by Francois Chollet¹⁴⁹. Mr. Chollet is also the author of the book "Deep Learning with Python"¹⁵⁰, an invaluable resource into both neural network architecture and Keras itself.

Footnote 149 blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html

Footnote 150 www.manning.com/books/deep-learning-with-python

During the training phase, we will train the encoder and decoder network together, end to end. This will require 3 data points for each sample, a training encoder input sequence, a

decoder input sequence, and a decoder output sequence. The training encoder input sequence could be a user question for which you would like a bot to respond. The decoder input sequence then is the expected reply by the future bot. You might wonder why we need an input *and* output sequence for the decoder. The reason is that we are training the decoder with a method called "*teacher forcing*", where we will use the initial state provided by the encoder network and train the decoder to produce the expected sequences by showing the input to the decoder and let it predict the same sequence. Therefore, the decoder's input and output sequence will be identical, except that the sequence have an offset of one time step.

During the execution phase, we will use the encoder to generate the thought vector of our user input and the decoder will then generate a reply based on that thought vector. The output of the decoder will then serve as the reply to the user.

NOTE**Keras Functional API**

In the following example, you will notice a different implementation style of the Keras layers you have seen in the previous chapters. Keras introduced an additional way of assembling models by calling each layer and passing the value from the previous layer to it. The *functional API* can be very powerful when you want to build models and reuse portions of the trained models (as we will demonstrate in the coming sections). For more information about Keras' functional API, we highly recommend the blog post by the Keras core developer team ¹⁵¹.

Footnote 151 keras.io/getting-started/functional-api-guide/

10.2.3 The Sequence-to-Sequence Encoder

The sole purpose of the encoder is the creation of our thought vector which then serves as the initial state of the decoder network. We can't consider the encoder fully in isolation as it cannot be trained on its own. We have no "target" thought vector for the network to learn to predict. The backpropagation that will train the encoder to create an appropriate thought vector will come from error that is generated later downstream in the decoder, but for now it is helpful to visualize the two parts separately and consider them each in turn.

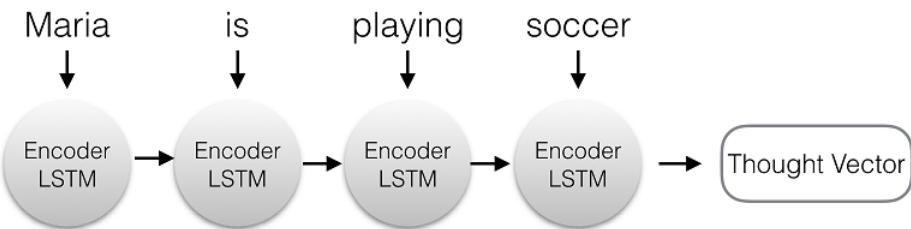


Figure 10.7 The Sequence-to-Sequence Encoder

Conveniently, the RNN layers, provided by Keras, return their internal state. Just instantiate the LSTM layer (or layers) with the keyword argument `return_state=True`. In the example, we preserve the final state of the encoder and disregard the actual output of the encoder. The list of the LSTM states is then passed to the decoder.

```
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(num_neurons, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]
```

- ➊ The `return_state` argument of the LSTM layer needs to be set to True to return the internal states
- ➋ The first return value of the LSTM layer is the output of the layer. If `return_sequences` was set to True, the first value would be a list of the outputs from each time step. As it is, it will simply be the output from the last time step. `state_h` will be specifically the output of the last time step for this layer. So in this case `encoder_outputs` and `state_h` will be identical. Either way we can ignore the official output stored in `encoder_outputs`. `state_c` is the current state of the memory unit. It is `state_h` and `state_c` that will make up our thought vector

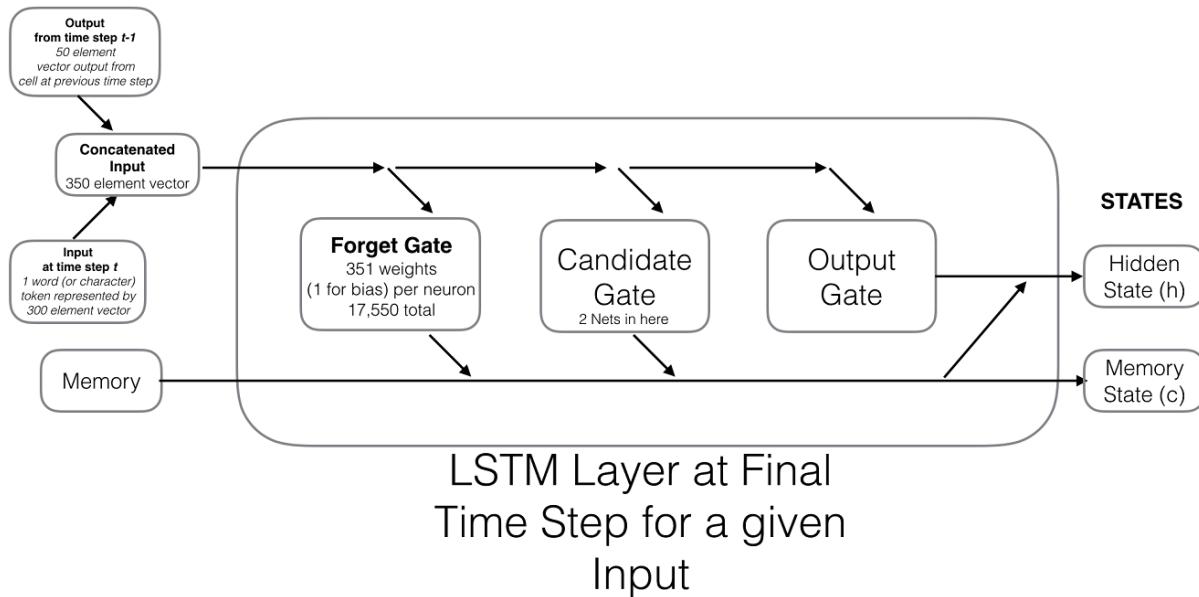


Figure 10.8 LSTM States used in the Sequence-to-Sequence Encoder

Figure 8 shows how the internal LSTM states are generated. The encoder will update the hidden and the memory state with every time step and pass the final states to the decoder as the initial state.

10.2.4 The Sequence-to-Sequence Decoder

Similar to the encoder network, the setup of the decoder is pretty straight forward. The major difference in the decoder setup is that this time we do want to capture the output of the network at each time step. We want to judge the "correctness" of the output token by token.

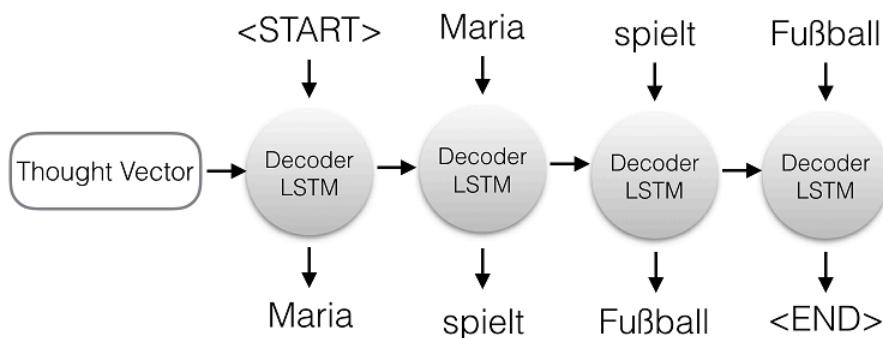


Figure 10.9 The Sequence-to-Sequence Decoder

This is where we use the second and third pieces of the sample triplets. The decoder has a standard token by token input and a token by token output. They happen to be almost identical, but off by one time step. We want the decoder to learn to reproduce the tokens of a given input sequence *given* the state generated by first piece of the triple fed into the encoder.

NOTE

This is the key concept in the decoder, and in seq2seq in general, we are training a network how to speak in the secondary problem space (another language or another being's response to a given question). We are forming a "thought" about what we want to say, and token by token teach a network what should be said next, given that thought and what does come next. Such that eventually we will only need the initial thought (which we will be able to generate from the encoder) and a generic start token to get things going to predict the right things to "say".

To calculate the error of the training step, we will pass the output of our LSTM layer into a *Dense* layer. The dense layer will have a number of neurons equal to the number of all possible output tokens. The dense layer will have a softmax activation function across those tokens. So at each time step, the network will provide a probability distribution over all possible tokens for what it thinks is most likely the next sequence element. Just

take the token whose related neuron has the highest value. We used output layer with softmax activation functions in earlier chapters, where we wanted to determine a token with the highest likelihood (Checkout Chapter 6 for more details). Also note that the *num_encoder_tokens* and the *num_decoder_tokens* do not need to match; one of the great benefits of sequence-to-sequence networks.

```
decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(num_neurons, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)
```

1
2
3
4

- ➊ Setup of the LSTM layer similar to the encoder with an additional argument of `return_sequences`
- ➋ The functional API allows us to pass the initial state to the LSTM layer by assigning the last encoder state to `initial_state`
- ➌ Softmax layer with all possible characters mapped to the softmax output
- ➍ Passing the output of the LSTM layer to the softmax layer

10.2.5 Assembling the Sequence-to-Sequence Network

The functional API of Keras allows us to assemble a model as object calls. The *Model* object lets us define its input and output parts of the network. For this sequence-to-sequence network, we will pass a list of our inputs to the model. Above, we defined one input layer in the encoder and one in the decoder. These two inputs correspond with the first two elements of each training triplet. As an output layer, we are passing the *decoder_outputs* to the model, which includes the entire model setup we previously defined. The output in *decoder_outputs* corresponds with the final element of each of our training triplets.

NOTE Using the Functional API like this, definitions such as *decoder_outputs* above are actually *tensor* representations. This is where you will notice differences from the *Sequential* model described in early chapters. Again refer to the documentation for the nitty-gritty of the Keras API.

```
model = Model(inputs=[encoder_inputs, decoder_inputs], outputs=decoder_outputs)
```

1

- ➊ The inputs and outputs arguments can be defined as lists if multiple in- or output

are expected

10.3 Training the Sequence-to-Sequence Network

The last remaining steps towards the training of the sequence-to-sequence setup are the model compile and fit steps. These are the same steps you have seen in the previous chapters. In the earlier chapters we were predicting a binary classification. Yes or no. However here we have categorical classification problem. At each time step which "category" is correct. Our "categories" here, however are just the space of all possible tokens to "say". Because we are predicting characters or words rather than binary states as we have done in the previous chapter, we will optimize our loss based on the *categorical_crossentropy* loss function, rather than the *binary_crossentropy* used earlier.

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy')  
①  
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,  
batch_size=batch_size, epochs=epochs)②
```

- ① Setting the loss function to categorical_crossentropy
- ② The model will expect the training inputs as a list, where the first list element will be passed to the encoder network and the second element will be passed to the decoder network during the training.

Congratulations! With the call to *model.fit*, you are training your sequence-to-sequence network, end to end. In the following sections we will demonstrate how you can infer an output sequence for a given input sequence.

NOTE

The training of sequence-to-sequence networks can be computationally intensive and therefore time-consuming. If your training sequences are long or if you want to train with a large corpus, we highly recommend training these networks on a *GPU (Graphical Processing Unit)*. The training speed can be increased by up to 30 times. If you have never trained a neural network with on a GPU, don't worry. Check out Chapter 13 on how to rent and set up your own GPU on commercial computational cloud services.

As LSTMs are not inherently parallelizable like Convolutional Neural Nets, to get the full benefit of a GPU you should replace the LSTM layers with *CuDNNLSTM* which is optimized for training on a GPU enabled with CUDA.

10.3.1 Generate output sequences

Before generating sequences, we need to take the structure of our training layers and reassemble them for generation purposes. At first, we define a model specific to the encoder. This model will then be used to generate the *thought vector*.

```
encoder_model = Model(inputs=encoder_inputs, outputs=encoder_states) ①
```

- ① Here, we use the previously defined `encoder_inputs` and `encoder_states`; calling the `predict` method on this model will return the thought vector

The definition of the decoder can look very daunting. But, let's untangle the code snippet step-by-step. First, we'll define our decoder inputs. We are using the Keras input layer, but instead of passing in one-hot vectors, characters or word embeddings, we will pass the thought vector generated by the encoder network. Please note, that the encoder returns a list of two states, which we will need to pass to the `initial_state` argument when calling our previously defined `decoder_lstm`. The output of the LSTM layer is then passed to the `dense` layer, which we also previously defined. The output of this layer will then provide the probabilities of all decoder output tokens (in our case, all seen characters during the training phase).

Here is the magic part. The token predicted with the highest probability at each time step, will then be returned as the most likely token and passed on to the next decoder iteration step, as the new input.

```
decoder_states_inputs = [Input(shape=(num_neurons,)), Input(shape=(num_neurons,))] ①
decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs,
                                                initial_state=decoder_states_inputs) ②
decoder_states = [state_h, state_c] ③
decoder_outputs = decoder_dense(decoder_outputs) ④

decoder_model = Model( ⑤
    inputs=[decoder_inputs] + decoder_states_inputs,
    output=[decoder_outputs] + decoder_states) ⑥ ⑦
```

- ① Define an input layer to take the encoder states
- ② Pass the encoder state to the LSTM layer as initial state
- ③ The updated LSTM state will then become the new cell state for the next iteration
- ④ Pass the output from the LSTM to the dense layer to predict the next token
- ⑤
- ⑥
- ⑦

- ➅ The last step is tying the decoder model together
- ➆ The `decoder_inputs` and `decoder_states_inputs` become the input to the decoder model
- ➇ The output of the dense layer and the updated states are defined as output

Once the model is set up, we can generate sequences by predicting the *thought vector* based on an one-hot encoded input sequence and the last generated token. During the first iteration, the `target_seq` will be set to the start token. During all following iterations, `target_seq` will be updated with the last generated token. This loop goes on until either we have reached the maximum number of the sequence elements or the decoder generated a stop token at which time the generation will be stopped.

```
...
states_value = encoder_model.predict(input_seq) ①
...
while not stop_condition:
    output_tokens, h, c = decoder_model.predict([target_seq] + states_value) ②
    ...
    target_seq = output_tokens[0] ③
```

- ➊ Generate the initial thought vector
- ➋ The `stop_condition` will be updated after each iteration and turns True if either the maximum number of output sequence tokens is hit or the decoder generate a stop token
- ➌ The decoder returns the token with the highest probability and the internal states which will be reused during the next iteration

10.4 Building a chatbot using seq2seq networks

In the previous sections, we learned how to train a sequence-to-sequence network and how to use the trained network to generate sequence responses. In the following section, we'll guide you through how to apply the various steps to train a chat bot. For the chat bot training, we will use the Cornell movie dialog corpus¹⁵². We will train a *seq2seq* network to "adequately" reply to your questions or statements. Our chat bot example is an adopted sequence-to-sequence example from the Keras blog¹⁵³.

Footnote 152 www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html

Footnote 153 github.com/fchollet/keras/blob/master/examples/lstm_seq2seq.py

10.4.1 Preparing the corpus for our training

First we need to load the corpus and generate the training sets from it. The training data will determine the set of characters the encoder and decoder will support during the training and during the generation phase. Please note that this implementation doesn't support characters which haven't been included during the training phase. Using the entire Cornell Movie Dialog data set can be computationally intensive since a few sequences have more than 2000 tokens. So 2,000 time steps will take a while to unroll. However, the majority of dialog samples are based on less than 100 characters. For this example, we have preprocessed the dialog corpus by limiting samples to those with less than 100 characters, removed odd characters and only allowed lower case characters. With these changes, we limit the variety of characters. You can find the preprocessed corpus in the GitHub repository of NLP in Action¹⁵⁴.

Footnote 154 github.com/totalgood/nlpia

We will loop over the corpus file and generate the training pairs (technically triplets, input text, target text with start token, and target text). While reading the corpus, we will also generate a set of input and target characters, which we then will use to one-hot encode the samples. The input and target characters don't have to match. However, characters which aren't included in the sets can't be read or generated during the generation phase. The result of the step below will be two lists of input and target texts (strings) as well as two sets of characters which have been seen in the training corpus.

```

corpus_filepath = 'dialog.txt'                                1

input_texts = []                                              2
target_texts = []

input_characters = set()                                     3
target_characters = set()

start_token = '\t'
stop_token = '\n'

with open(corpus_filepath, 'r') as fh:
    lines = fh.read().split('\n')                            5
    max_training_samples = min(25000, len(lines) - 1)        6

    for line in lines[:max_training_samples]:
        input_text, target_text = line.split('\t')            7
        target_text = start_token + target_text + stop_token   8
        input_texts.append(input_text)
        target_texts.append(target_text)

```

```

for char in input_text:
    if char not in input_characters:
        input_characters.add(char)
for char in target_text:
    if char not in target_characters:
        target_characters.add(char)

```

9

- ➊ Define the file path of the preprocessed corpus file
- ➋ The arrays will hold the input and target text read from the corpus file
- ➌ The sets will hold the seen characters in the input and target text
- ➍ The target sequence will be annotated with a start (first) and stop (last) token; the characters representing the tokens are defined here; These tokens can't be part of the normal sequence text and should be uniquely being used as start and stop tokens
- ➎ The corpus text is read from the file
- ➏ max_training_samples defines how many lines will be used for the training. It will be the lower number of either a user defined maximum or the total number of lines loaded from the file
- ➐ The dialog statements and their replies are separated by tabs
- ➑ The target_text needs to be annotated with the start and stop token
- ➒ Here we are looking over each character of the input and target text and we are checking if we have seen this character in a previous sample; if not, we will add it to the character sets

10.4.2 Building our character dictionary

Similar to the examples from our previous chapters, we need to convert each character of the input and target texts into one-hot vectors which represent each character. In order to generate the one-hot vectors, we generate token dictionaries (for the input and target text), where every character is mapped to an index. We also generate the reverse dictionary (index to character), which we will use during the generation phase to convert the generated index to a character.

```

input_characters = sorted(list(input_characters))
target_characters = sorted(list(target_characters))

num_encoder_tokens = len(input_characters)
num_decoder_tokens = len(target_characters)

max_encoder_seq_length = max([len(txt) for txt in input_texts])
max_decoder_seq_length = max([len(txt) for txt in target_texts])

input_token_index = dict([(char, i) for i, char in enumerate(input_characters)])
target_token_index = dict([(char, i) for i, char in enumerate(target_characters)])

```

1

2

3

4

```
reverse_input_char_index = dict((i, char) for char, i in input_token_index.items())
reverse_target_char_index = dict((i, char) for char, i in target_token_index.items())
```

5

- ➊ We convert the character sets into sorted lists of characters which we will then use to generate the dictionary from
- ➋ For the input and target data, we will determine the maximum number of unique characters which we will use to build the one-hot matrices
- ➌ For the input and target data, we will also determine the maximum number of sequence tokens
- ➍ Looping over the input_characters and target_characters to create the look up dictionaries which we will use to generate the one-hot vectors
- ➎ Looping over the newly created dictionaries will create the reverse lookups

10.4.3 Generate one-hot encoded training sets

In the next step, we are converting the input and target text into one-hot encoded "tensors". In order to do that, we are looping over each input and target samples, and over each character of each sample and one-hot encode each character. Each character will be encoded by a $n \times 1$ vector (with n being the number of unique input or target characters). All vectors will then be combined to a matrix for each sample and all samples will be combined into the training tensor.

```
import numpy as np

encoder_input_data = np.zeros((len(input_texts), max_encoder_seq_length,
                               num_encoder_tokens), dtype='float32')
decoder_input_data = np.zeros((len(input_texts), max_decoder_seq_length,
                               num_decoder_tokens), dtype='float32')
decoder_target_data = np.zeros((len(input_texts), max_decoder_seq_length,
                               num_decoder_tokens), dtype='float32')

for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.
    for t, char in enumerate(target_text):
        decoder_input_data[i, t, target_token_index[char]] = 1.
        if t > 0:
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.
```

1

2

3

4

5

6

- ➊ We are using numpy for the matrix manipulations
- ➋ The training tensors will be initialized as zero tensors with the shape of number of samples (this number should be equal for the input and target samples) x the maximum number of sequence tokens x the number of possible characters

3

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

Looping over the training samples; input and target texts need to match

- ④ Looping over each character of each sample
- ⑤ Setting the index for the character at each time step to one, all other indices will remain zero; this creates the one-hot encoded representation of the training samples
- ⑥ For the training data for the decoder, we are creating the decoder_input_data and decoder_target_data; decoder_target_data is one time step behind the decoder_input_data

10.4.4 Train your sequence-to-sequence chatbot

After all the training set preparation, converting the preprocessed corpus into input and target samples, creating index lookup dictionaries and converting the samples into one-hot tensors, it's time to train the chat bot. The code is identical to the earlier samples. Once the *model.fit* completes the training, you have a fully trained chat bot based on a sequence-to-sequence network.

```
from keras.models import Model
from keras.layers import Input, LSTM, Dense

batch_size = 64          ①
epochs = 100             ②
num_neurons = 256        ③

encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(num_neurons, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None, num_decoder_tokens))
decoder_lstm = LSTM(num_neurons, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['acc'])
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
          batch_size=batch_size, epochs=epochs, validation_split=0.1) ④
```

- ① In this example, we set the batch size to 64 samples; increasing the batch size can speed up the training, it might also require more memory
- ② Training a sequence-to-sequence network can be lengthly and easily require 100 epochs
- ③ In this example, we are setting the number of neuron dimensions to 256
- ④ We withhold 10% of the samples for validation tests after each epoch

10.4.5 Assemble the model for sequence generation

Setting up the model for the sequence generation is very much the same as we discussed it in the earlier sections, but we have to make some adjustments, as we don't have a specific target text to feed into the decoder along with the state. All we have is the input, that we want to infer from, and a start token.

```
encoder_model = Model(encoder_inputs, encoder_states)

decoder_states_inputs = [Input(shape=(num_neurons,)), Input(shape=(num_neurons,))]
decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs,
    initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)

decoder_model = Model(
    inputs=[decoder_inputs] + decoder_states_inputs,
    output=[decoder_outputs] + decoder_states)
```

10.4.6 Predicting a sequence

The `decode_sequence` function is the heart of the response generation of our chat bot. It accepts a one-hot encoded input sequence, generates the thought vector and using the thought vector to generate the "appropriate" response using the network trained earlier.

```
def decode_sequence(input_seq):
    states_value = encoder_model.predict(input_seq) ①

    target_seq = np.zeros((1, 1, num_decoder_tokens)) ②
    target_seq[0, 0, target_token_index[stop_token]] = 1. ③

    stop_condition = False
    generated_sequence = ''

    while not stop_condition:
        output_tokens, h, c = decoder_model.predict([target_seq] + states_value) ④

        generated_token_index = np.argmax(output_tokens[0, -1, :])
        generated_char = reverse_target_char_index[generated_token_index]
        generated_sequence += generated_char

        if (generated_char == stop_token or len(generated_sequence) > max_decoder_seq_length): ⑤
            stop_condition = True

    target_seq = np.zeros((1, 1, num_decoder_tokens)) ⑥
    target_seq[0, 0, generated_token_index] = 1.

    # Update states
    states_value = [h, c]

return generated_sequence
```

- ① Generate the thought vector as the input to the decoder
- ② In contrast to the training time, target_seq starts off as a zero tensor
- ③ The first input token to the decoder is the start token
- ④ Passing the already generated tokens and the latest state to the decoder to predict the next sequence element
- ⑤ Setting the stop condition to True will stop the loop
- ⑥ Update the target sequence and using the last generated token as the input to the next generation step

10.4.7 Generating a response

Now we'll define a helper function, `response()`, to convert an input string (e.g. a statement from a human user) into a statement by the chatbot to reply to that statement. This function first converts the input text from the user into a sequence of one-hot encoded vectors. That tensor of one-hot vectors is then passed to the previously defined `decode_sequence()` function. The `decode_sequence()` function accomplishes the dynamic encoding of the input texts into thought vectors and the generation of text from that thought vector, while the model is running outside of the training phase.

NOTE

The key is that instead of providing an initial state (thought vector) and an input sequence to the decoder, we are supplying only the thought vector and a start token. The token that the decoder produces given the initial state and the start token becomes the input to the decoder at time step 2. And the output at time step 2 becomes the input at time step 3, and so on. All the while the LSTM memory state is updating the memory and augmenting output as it goes. Just like we saw in chapter 9.

```
def response(input_text):
    input_seq = np.zeros((1, max_encoder_seq_length, num_encoder_tokens), dtype='float32')
    1
    for t, char in enumerate(input_text):
        input_seq[0, t, input_token_index[char]] = 1.
    2
    decoded_sentence = decode_sequence(input_seq)
    print('Bot Reply (Decoded sentence):', decoded_sentence)
```

- ① Looping over each character of the input text to generate the one-hot tensor for the encoder to generate the thought vector from
- ② Using the `decode_sequence` function will call the trained model and generate the response sequence.

10.4.8 Converse with your chatbot

Voila! You just completed all necessary steps to train and use your own chat bot. Congratulations! Interested what the chat bot can reply to? After 100 epochs of training, which took approximately seven and a half hours on a NVIDIA GRID K520 GPU, the trained sequence-to-sequence chat bot was still a bit stubborn and short spoken. A larger and more general training corpus could change that behavior.

```
>>> response("what is the internet?")
Bot Reply (Decoded sentence): it's the best thing i can think of anything.

>>> response("why?")
Bot Reply (Decoded sentence): i don't know. i think it's too late.

>>> response("do you like coffee?")
Bot Reply (Decoded sentence): yes.

>>> response("do you like football?")
Bot Reply (Decoded sentence): yeah.
```

NOTE

If you don't want to setup a GPU and train your own chat bot, no worries. We have made the trained chat bot available for you to test it. Head over to the GitHub repository of *NLP in Action*¹⁵⁵ and check out the latest version of the chat bot. Let the authors know if you come across any funny replies by the chat bot.

Footnote 155 github.com/totalgood/nlpia

10.5 Enhancements

10.5.1 Reduce Training Complexity by using Bucketing

Input sequences can have very different lengths, which can add a large number of pad tokens to short sequences in your training data. Too much padding can make the computation expensive, especially when the majority of the sequences are short and only a handful of them use close to the maximum length of tokens. Imagine you train your sequence-to-sequence network with data where almost all samples are 100 tokens long, except for a few outliers which contain 1000 tokens. Without bucketing, you would need to pad the majority of your training with 900 pad tokens and your seq2seq network would have to loop over them during the training phase. This will slow down the training dramatically. Bucketing can reduce the computation in these cases. As shown in the Figure 10, you can sort the sequences by length and use different sequence lengths during different batch runs. You assign the input sequences to buckets of different lengths, e.g. all sequences with a length between five and ten tokens and then use the sequence buckets for your training batches, e.g. train first with all sequences between five and ten tokens, then ten to 15, etc. Some deep learning frameworks provide bucketing tools to suggest the optimal buckets for your input data.

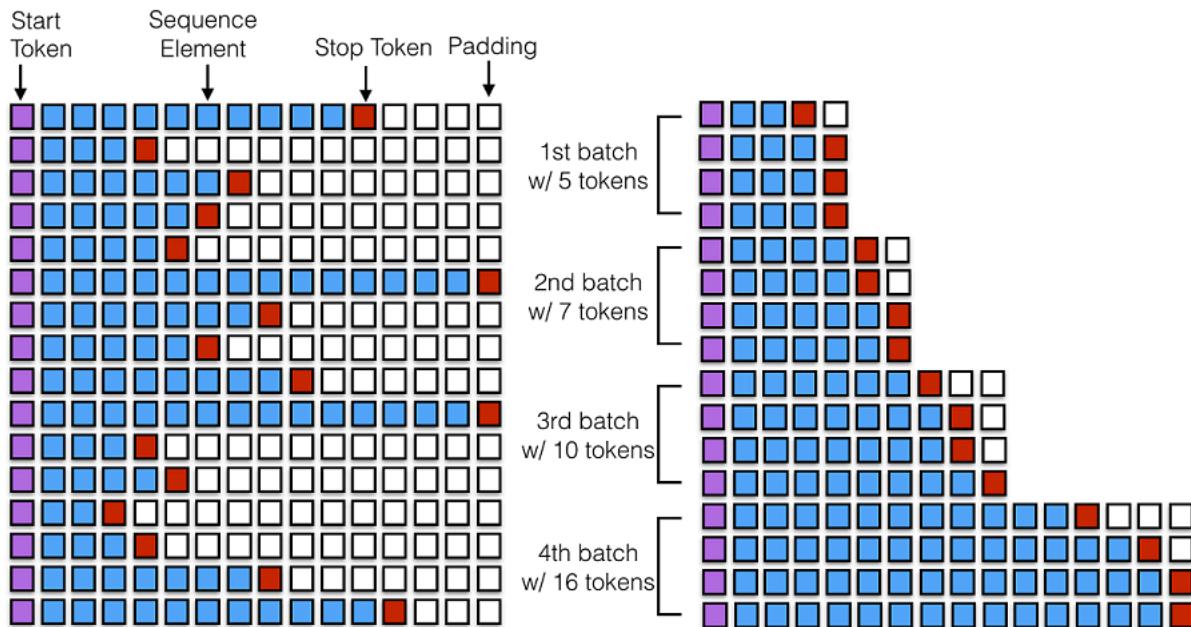


Figure 10.10 Bucketing applied to target sequences

As shown in Figure 10, the sequences were first sorted by length and then only padded to the maximum token length for the particular bucket. That way, we can reduce the number

of time steps needed for any particular batch while training the seq2seq network. We only unroll the network as far as is necessary (to the longest sequence) in a given training batch.

10.5.2 Paying Attention

As with Latent Semantic Analysis we introduced in Chapter 4, longer input sequences (documents) tend to produce thought vectors that are less precise representations of those documents. A thought vector is limited by the dimensionality of the LSTM layer (the number of neurons). A single thought vector is sufficient for short input/output sequence, similar to our chatbot example. But imagine the case when you want to train a seq2seq model to summarize online articles. In this case, your input sequence can be a lengthy article which should be compressed into a single thought vector to generate e.g. a headline. As you can imagine, it is tricky to train the network to determine the most relevant information in that longer document. A headline or summary (and the associated thought vector) must be focused on a particular aspect or portion of that document rather than attempt to represent all of the complexity of the meaning of that document.

In 2015, Bahdanau et al. presented their solution to this problem at the International Conference on Learning Representations¹⁵⁶. The concept the authors developed became known as the *Attention Mechanism*. As the name suggests, the idea is to tell the decoder what to pay attention to in the input sequence. This "sneak preview" is achieved by allowing the decoder to also look all the way back into the states of the encoder network in addition to the thought vector. A version of "heat map" over the entire input sequence is learned along with the rest of the network. That mapping, different at each time step, is then shared with the decoder. As it decodes any particular part of the sequence its concept created from the thought vector can be augmented with direct information that produced. In other words, the attention mechanism allows a direct connection between the output and the input by selecting relevant input pieces. This does not mean token to token alignment, that would defeat the purpose and send us back to autoencoder land. It does allow for richer representations of concepts wherever they appear in the sequence.

Footnote 156 arxiv.org/abs/1409.0473

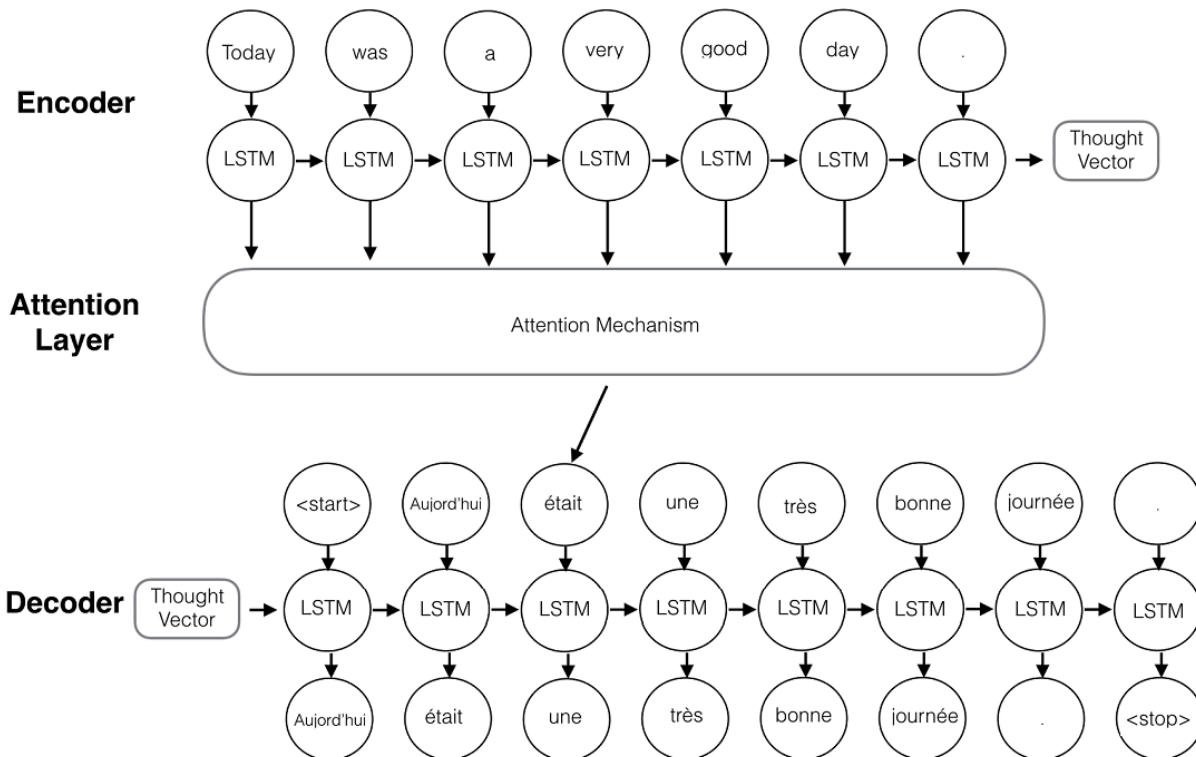


Figure 10.11 Overview of the Attention Mechanism

With the attention mechanism, the decoder receives an additional input with every time step representing the one (or many) tokens in the input sequence to pay "attention" to, at this given decoder time step. All sequence positions from the encoder will be represented as a weighted average for each decoder time step.

Configuring and tuning the attention mechanism is not trivial, but various deep learning frameworks provide implementations to facilitate this. However, at the time of writing this chapter, a pull request to the Keras package was discussed, but no implementation had yet been accepted.

10.6 In the Real World

Sequence-to-Sequence networks are well suited for any machine learning application with variable length input sequences or variable length output sequences. Since natural language sequences of words almost always have unpredictable length, Sequence-to-Sequence models can improve the accuracy of most machine learning models.

Key areas of applications are:

- Chat bot conversations
- Question-answering

- Machine translation
- Image captioning
- Visual question answering
- Document summarization

As you have seen in the previous sections, a *dialog system* is a common application. Seq2seq models are generative which makes them especially well-suited to conversational dialog system (chatbot) as opposed to information retrieval or task accomplishment dialog systems. Conversational dialog is focused on mimicking human conversation on a broad range of topics. Seq2seq chatbots can generalize from limited-domain corpora and thus respond reasonably on topics not contained in their training set. In contrast, the "grounding" of knowledge-based dialog systems (to be discussed in Chapter 12) can limit their ability to participate in conversations on topics outside their training domains. Chapter 12 compares the performance of chatbot architectures in greater detail.

Besides the Cornell Movie Dialog Corpus, various free and open source training are available, such as Deep Mind's Q&A data sets ¹⁵⁷ ¹⁵⁸. When you need your dialog system to respond reliably in a specific domain you will need to train it on a corpora of statements from that domain.

Footnote 157 Q&A data set cs.nyu.edu/~kcho/DMQA/

Footnote 158 List of dialog corpora in the NLPIA package docs:
github.com/totalgood/nlpia/blob/master/docs/nlp—data.md#dialog-corpora

Another common application for seq2seq networks is machine translation. The concept of the thought vector allows a translation application to incorporate the *context* of the input data and words with multiple meaning can be translated in the correct context. If you want to build translation applications, the website *manythings.org* provides sentence pairs which can be used as training sets. The website ¹⁵⁹ provides a long list of various languages, e.g. Icelandic, Basque, Indonesian.

Footnote 159 www.manythings.org/anki/

Seq2seq models are also well-suited to text summarization, due to the difference in string length between input and output. In this case, the input to the encoder network is for example news articles (or any other length document) and the decoder can be trained to generate a headline or abstract or any other summary sequence associated with the document. Sequence-to-sequence networks can provide more natural sounding text summaries than summarization methods based on bag-of-words vector statistics. If you

are interested in developing such an application, the Kaggle news summary challenge¹⁶⁰ provides a good training set.

Footnote 160 www.kaggle.com/sunnysai12345/news-summary/data

Seq2seq networks are not limited to natural language applications. Two other applications are automated speech recognition and image captioning. Current, state-of-the-art automated speech recognition systems¹⁶¹ use seq2seq networks to turn voice input amplitude sample sequences into the thought vector that a seq2seq decoder can turn into a text transcription of the speech. The same concept applies to image captioning. The sequence of image pixels (regardless of image resolution) can be used as an input to the encoder, and a decoder can be trained to generate an appropriate description. In fact you can find a combined application of image captioning and Q&A system called visual question answering at cloudcv.org.¹⁶²

Footnote 161 State of the art speech recognition system arxiv.org/pdf/1610.03022.pdf

Footnote 162 vqa.cloudcv.org/

10.7 Summary

In this chapter

- Training a model to generate sequence on input sequences is possible with sequence-to-sequence networks
- Sequence-to-Sequence networks consist out of an encoder and a decoder model
- The encoder model generates a *thought vector* which is a vector presentation of the input context
- The decoder then uses the *thought vector* to start predicting output sequences. The entirety of the output sequences form the chat bots response
- Due to the *thought vector* representation, the input and the output sequence lengths don't have to match (great for machine translation)
- *Thought vectors* can only hold a limited amount of information. If longer texts need to be thought vector encoded then *Attention* is a great tool to encode what is really important

Getting Real (Real World NLP Challenges)

P3

Part Three will show you how to extend your skills to tackle real world problems. You'll learn how to extract information like dates and names to build applications like the Twitter Bot that helped manage the self-service scheduling of Open Spaces at PyCon.

In these last three chapters we'll also tackle the trickier problems of NLP. You'll learn about several different ways to build a chatbot, both with and without machine learning to guide it. And you'll learn how to combine these techniques together to create complex behavior. And you'll also learn about algorithms that can handle large corpora, sets of documents that cannot be loaded into RAM all at once.

Information Extraction (Named Entity Extraction and Question Answering)

In this chapter

- Sentence segmentation
- Named entity recognition
- Numerical information extraction
- Parts-of-Speech (POS) tagging and dependency tree parsing
- Logical relation extraction and knowledge bases

There's one last skill we need before we can build a full-featured chatbot, we need to extract information or knowledge from natural language text.

11.1 Named Entities and Relations

We'd like our machine to extract pieces of information, facts, from text so it can know a little bit about what a user is saying. For example, imagine a user says "Remind me to read aiindex.org on Monday." We'd like that statement to trigger a calendar entry or alarm for the next Monday after the current date.

To trigger those actions we'd need to know that "me" represents a particular kind of *named entity*, a person. And the chatbot should know that it should "expand" or normalize that word by replacing it with the username of the human that made that statement. We'd also need our chatbot to recognize that "aiindex.org" is an abbreviated URL, a *named entity*, the name of a specific instance of something. And we need to know that a normalized spelling of this particular kind of *named entity* might be "<http://aiindex.org>", "<https://aiindex.org>", or maybe even "<https://www.aiindex.org>". Likewise we need our chatbot to recognize that Monday is one of the days of the week (another kind of named entity called an "event") and be able to find it on the calendar.

For the chatbot to respond properly to that simple request we also need it to extract the relation between the named entity "me" and the command "remind." We'd even need to recognize the implied subject of the sentence, "you", referring to the chatbot, another "person" named entity. And we need to "teach" the chatbot the fact that reminders happen in the future, so it should find the soonest upcoming Monday to create the reminder.

A typical sentence may contain several *named entities* of various types, like geographic entities, organizations, people, political entities, times (including dates), artifacts, events, and natural phenomena. And a sentence can contain several *relations* too, facts about the relationship between the named entities in the sentence.

11.1.1 A Knowledge Base

Besides just extracting information from the text of a user statement, we can also use information extraction to help our chatbot train itself! If we have our chatbot run information extraction on a large corpus, like Wikipedia, that corpus will produce facts about the world that we can inform future chatbot behaviors and replies. Some chatbots record all the information they extract (from offline reading-assignment "homework") in a knowledge base. That knowledge base can later be queried to help our chatbot make informed decisions or inferences about the world.

Chatbots can also store knowledge about the current user "session" or conversation. This can be used to build up information about the current *context* of the chatbot. Knowledge that is relevant only to the current conversation is called "context." This contextual knowledge can be stored in the same global knowledge base that supports the chatbot, or it can be stored in a separate knowledge base. Commercial chatbot APIs like IBM's Watson, or Amazon's Lex, typically store context separate from the global knowledge base of facts that it uses to support conversations with all the other users.

Context can include facts about the user, the chatroom or channel, or the weather and news for that moment in time. Context can even include the changing state of the chatbot itself, based on the conversation. An example of "self-knowledge" a smart chatbot should keep track of is the history of all the things it has already told someone or the questions it has already asked of the user, so it doesn't repeat itself.

So that's the goal for this chapter, teaching our bot to understand what it reads. And we'll put that understanding into a flexible data structure designed to store knowledge. Then our bot can use that knowledge to make decisions and say smart stuff about the world.

In addition to the simple task of recognizing numbers and dates in text, we'd like our bot

to be able to extract more general information about the world. And we'd like it to do this on its own, rather than having us "program" everything we know about the world into it. For example, we'd like it to be able to learn from natural language documents like this sentence from Wikipedia:

In 1983, Stanislav Petrov, a lieutenant colonel of the Soviet Air Defense Forces, saved the world from nuclear war.

If you were to take notes in a history class after reading or hearing something like that, you'd probably paraphrase things and create connections in your brain between concepts or words. You might reduce it to a piece of knowledge, that thing that you "got out of it." We'd like our bot to do the same thing. We'd like it to "take note" of whatever it learns, like the fact or knowledge that Stanislov Petrov was a lieutenant colonel. This could be stored in a data structure something like this:

```
('Stanislav Petrov', 'is-a', 'lieutenant colonel')
```

This is an example of 2 named entity nodes ('Stanislav Petrov' and 'lieutenant colonel') and a *relation* or connection ('is a') in a *knowledge graph* or *knowledge base*. When a relationship like this is stored in a form that complies with the "RDF" standard (Relation Description Format) for knowledge graphs, it's referred to as an RDF triplet. Historically these RDF triplets were stored in XML files, but they can be stored in any file format or database that can hold a graph of triplets in the form of (subject, relation, object).

A collection of these triplets is a knowledge graph. This is also sometimes called an "ontology" by linguists, because it is storing structured information about words. But when the graph is intended to represent facts about the world rather than merely words it is referred to as a knowledge graph or knowledge base. Here's a graphic representation of the *knowledge graph* we'd like to extract from a sentence like that.

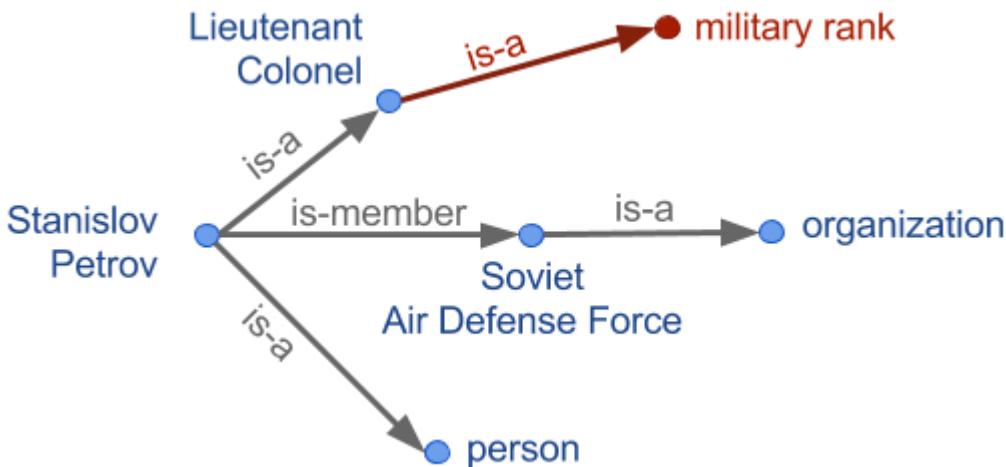


Figure 11.1 Stanislav Knowledge Graph showing 'is-a' and 'is-famous-for' relations extracted

The red edge and node in the knowledge graph diagram above represents a fact that could not be directly extracted from the statement about Stanislav. But this fact that "Lieutenant Colonel" is a military rank could be inferred from the fact that the title of a person who is a member of a military organization is a military rank. This logical operation of deriving facts from a knowledge graph is called knowledge graph *inference* or just "inference". It can also be called querying a knowledge base, analogous to querying a relational database.

For this particular inference or query about Stanislav's military ranks our knowledge graph would have to already contain facts about militaries and military ranks. It might even help if the knowledge base had facts about the titles of people and how people relate to occupations (jobs). Perhaps you can see now how a base of knowledge helps a machine understand more about a statement than it could without that knowledge. Without this base of knowledge many of the facts in a simple statement like this will be "over the head" of our chatbot. You might even say that questions about occupational rank would be "above the pay grade" of a bot that only knew how to classify documents according to randomly allocated topics.¹⁶³

Footnote 163 see Chapter 4 if you've forgotten about how random topic allocation can be

It may not be obvious how big a deal this is, but it is a *very* big deal. If you've ever interacted with a chatbot that doesn't understand "which way is up", literally, you'd understand. One of the most daunting challenges in AI research is the challenge of compiling and efficiently querying a knowledge graph of common sense knowledge. We take common sense knowledge for granted in our everyday conversations.

Humans start acquiring much of our common sense knowledge even before we acquire language skill. We don't spend our childhood writing about how a day begins with light and sleep usually follows sunset. And we don't edit Wikipedia articles about how an empty belly should only be filled with food rather than dirt or rocks. This makes it hard for machines to find a corpus of common sense knowledge to read and learn from. There aren't any common sense knowledge Wikipedia articles for our bot to do information extraction on. And some of that knowledge is instinct, hard-coded into our DNA.¹⁶⁴

Footnote 164 There are hard-coded common-sense knowledge bases out there for you to build on. Google Scholar is your friend in this knowledge graph search.

There are all kinds of factual relationships between things and people that have come up with over the years, like "kind-of", "is-used-for", "has-a", "is-famous-for", "was-born", "has-profession." NELL, the Carnegie Mellon Never Ending Language Learning bot is focused almost entirely on the task of extracting information about the 'kind-of' relationship.

Most knowledge bases normalize the strings that define these relationships, so that "kind of" and "type of" would be assigned a normalized string or ID to represent that particular *relation*. And some knowledge bases also normalize the nouns representing the objects in a knowledge base. So 'Stanislav Petrov' might be assigned a particular ID along with the names "S. Petrov" and "Lt Col Petrov" whenever those names were used to refer to the same person.

A knowledge base can be used to build a practical type of chatbot called a *question answering system* (QA system). Customer service chatbots, including university TA bots, rely almost exclusively on knowledge bases to generate their replies.¹⁶⁵ Question answering systems are great for helping humans find factual information. This frees up human brains to do the things we're better at, like attempting to generalize from those facts. Humans are bad at remembering facts accurately, but good at finding connections and patterns between those facts, something machines have yet to master. We'll talk a more about *question answering* chatbots in the next chapter.

Footnote 165 2016, AI Teaching Assistant at GaTech:
www.news.gatech.edu/2016/05/09/artificial-intelligence-course-creates-ai-teaching-assistant

11.1.2 Information Extraction

So we've learned that "information extraction" is converting unstructured text into structured information stored in a *knowledge base* or *knowledge graph*. Information extraction is part of an area of research called "Natural Language Understanding" (NLU), though that term is often used synonymously with "Natural Language Processing" (NLP).

Information extraction and NLU is a different kind of learning than you may think of when researching data science. It isn't only unsupervised learning, even the very "model" itself, the logic about how the world works, can be composed without human intervention. Instead of giving our machine fish (facts) we're teaching it how to fish (extract information). Nonetheless, machine learning techniques are often used to train the information extractor.

11.2 Regular Patterns

We need a pattern-matching algorithm that can identify sequences of characters or words that match the pattern so we can "extract" them from a longer string of text. Easiest way to build such a pattern-matching algorithm is in Python, with a sequence of if/then statements that look for that symbol (a word or character) at each position of a string. Say we wanted to find some common greeting words, like "Hi", "Hello", and "Yo", at the beginning of a statement, we might do it something like this:

Listing 11.1 Grammar Hard-coded in Python

```
def find_greeting(s):
    """ Return the the greeting string Hi, Hello, or Yo if it occurs at the
    beginning of a string """
    if s[0] == 'H':
        if s[:3] in ['Hi', 'Hi ', 'Hi,', 'Hi!']:
            return s[:2]
        elif s[:6] in ['Hello', 'Hello ', 'Hello,', 'Hello!']:
            return s[:3]
    elif s[0] == 'Y':
        if s[1] == 'o' and s[:3] in ['Yo', 'Yo,', 'Yo ', 'Yo!']:
            return s[:2]
    return None
```

And here's how it would work:

Listing 11.2 Brittle Grammar Example

```
>>> find_greeting('Hi Mr. Turing!')
'Hi'
>>> find_greeting('Hello, Rosa.')
'Hello'
>>> find_greeting("Yo, what's up?")
'Yo'
```

```
>>> find_greeting("Hello")
'Hello'
>>> print(find_greeting("hello"))
None
>>> print(find_greeting("HelloWorld"))
None
```

You can probably see how tedious it would be to program a grammar (string processing algorithm) this way. And it's not even that good. It's quite brittle, relying on very precise spellings and capitalization and position characters in a string. And it's tricky to specify all the "delimiters", like punctuation, white space, or the beginnings and ends of strings (NULL character), that are on either sides of the words we're looking for.

We could probably come up with a way to allow us to specify a bunch of different words or strings that we want to look for without hard-coding them into Python expressions like this. And we could even specify the delimiters in a separate function. That would let us do some tokenization and iteration to find the occurrence of the words we are looking for anywhere in a string. But that's a lot of work.

Fortunately that work has already been done! A pattern-matching engine is integrated into most modern computer languages, including Python. It's called "regular expressions." Regular expressions, like string interpolation formatting expressions (e.g. "`{:05d}`".format(42)), are actually a mini programming language unto themselves. This language for pattern matching is called the *regular expression* language. And Python has a *regular expression* interpreter (compiler and runner) in the standard library package `re`. So let's use them to define our patterns instead of deeply nested Python `if` statements

11.2.1 Regular Expressions

Regular expressions are strings written in a special computer language that we can use to specify algorithms like the `find_greeting()` function above. Regular expressions are a lot more powerful, flexible, and concise than the equivalent Python we'd need to write to match patterns like this. So regular expressions are the pattern definition language of choice for many NLP problems involving pattern matching. This NLP application is an extension of their original use for compiling and interpreting formal languages (computer languages).

Regular expressions define a *finite state machine* or FSM, a tree of "if-then" decisions about a sequence of symbols, like our `find_greeting()` function above. The symbols in the sequence are passed into the decision tree of the FSM one symbol at a time. A *finite state machine* that operates on a sequence of symbols like ASCII character strings, or a

sequence of English words, is called a *grammar*. They can also be called *formal grammars* to distinguish them from natural language grammar rules you learned in elementary school.

In computer science and mathematics, the word "grammar" refers to the set of rules that determine whether or a sequence of symbols is a valid member of a *language*, often called a *computer language* or *formal language*. And a *computer language*, or *formal language*, is the set of all possible statements that would match the *formal grammar* that defines that *language*. That's kind-of a circular definition, but that's the way mathematics works sometimes. You probably want to review Appendix B if you aren't familiar with basic regular expression syntax and symbols like `.` `*` and `[a-z]`.

11.2.2 Information Extraction as ML Feature Extraction

So we're back where we started in Chapter 1, where we first mentioned regular expressions. But didn't we switch from "grammar-based" NLP approaches at the end of Chapter 1 in favor of Machine Learning and data-driven approaches? Why are we returning to hard-coded (manually composed) regular expressions and patterns? Because our statistical or data-driven approach to NLP has limits.

There are some basic things we want our machine learning pipeline to be able to do, like answer logical questions, or perform actions like scheduling meetings based on NLP instructions. And machine learning falls flat here. We rarely have a labeled training set that covers the answers to all the questions people might ask in natural language. Plus, as you'll see here, it's possible to define a compact set of condition checks (a regular expression) to extract key bits of information from a natural language string. And it can work for a broad range of problems.

Pattern matching (and regular expressions) continue to be the state-of-the art approach for information extraction (more commonly called *information retrieval*). Even with machine learning approaches to natural language processing, we need to do feature engineering. We need to create bags of words or "embeddings" of words to try to reduce the nearly infinite possibilities of meaning in natural language text into a vector that a machine can process easily. *Information extraction* is just another form of machine learning feature extraction from unstructured natural language data, like creating a bag of words, or doing PCA on that bag of words. And these patterns and features are still employed in even the most advanced natural language machine learning pipelines like Google's Assistant, Siri, Amazon Alexa, and other state-of-the-art "bots."

Information extraction is used to find statements and information that we might want our

chatbot to have "on the tip of its tongue." Information extraction can be accomplished beforehand to populate a knowledge base of facts. Alternatively, the required statements and information can be found on-demand, when chatbot is asked a question or a search engine is queried. When a knowledge base is built ahead of time, the data structure can be optimized to facilitate faster queries within larger domains of knowledge. This enables the chatbot to respond quickly to questions about a wider range of information. If information is retrieved in real-time, as the chatbot is being queried, this is often called "search." Google and other search engines combine these two techniques, querying a *knowledge graph* (knowledge base) and falling back to text search if the necessary facts aren't found. Many of the natural language grammar rules you learned in school can be encoded in a *formal grammar* designed to operate on words or symbols representing parts of speech (POSes). And the English language can be thought of as the words and grammar rules that make up the language. Or you can think of it as the set of all possible things you could say that would be recognized as valid statements by an English language speaker.

And that brings us to another feature of *formal grammars* and *finite state machines* that will come in handy for NLP. Any *formal grammar* can be used by a machine in two ways:

1. to recognize "matches" to that grammar
2. to generate a new sequence of symbols

So, not only can we use patterns (regular expressions) for extracting information from natural language, but we can also use them in a chatbot that wants to "say" things that match that pattern! We'll show you how to do this with a package called `rstr`¹⁶⁶ for some of our information extraction patterns here.

Footnote 166 bitbucket.org/leapfrogdevelopment/rstr/

This *formal grammar* and *finite state machine* (FSM) approach to pattern matching has some other awesome features. A true finite state machine (FSM) can be guaranteed to always run in finite time (to "halt"). It will always tell us whether we've found a match in our string or not. It will never get caught in a perpetual loop... as long as you don't use some of the advanced features of regular expression engines that allow you to "cheat" and incorporate loops into your FSM.

So we'll stick to regular expressions that don't require these "look-back" or "look-ahead" cheats. We'll make sure our regular expression matchers processes each character and moves ahead to the next character only if it matches. This is like a strict train conductor

walking through the seats of a train checking tickets. If you don't have one, the conductor stops and declares that there's a "problem", a "mismatch," and he refuses to go on, or look ahead or behind you until he resolves the "problem." There are no "go backs" or "do overs" for train passengers, or for strict regular expressions.

11.3 Information Worth Extracting

Some keystone bits of information are worth the effort of "hand-crafted" regular expressions and patterns.

- Numbers
- Dates
- Question Trigger Words
- Question Target Words
- Named Entities

11.3.1 Extracting Numbers

Lets start with regular expressions to extract numerical information from text, things like prices, dates, and GPS locations.

11.3.2 GPS Locations

GPS locations come in pairs of numerical values for latitude and longitude. They sometimes also include a third number for altitude, height above Sea Level, but we'll ignore that for now. Let's just extract decimal latitude longitude pairs, expressed in degrees. This will work for many Google Maps URLs. Though URLs are not technically natural language, they are often part of unstructured text data, and we'd like to extract this bit of information, so our chatbot can know about places as well as things.

Let's use our decimal number pattern from previous examples, but let's be more restrictive and make sure the value is within the valid range for latitude (+/- 90 deg) and longitude (+/- 180 deg). You can't go any farther north than the North Pole (+90 deg) or further south than the south pole (-90 deg). And if you sail from Greenwich England 180 deg East, (+180 deg longitude) you'll reach the Date Line where you're also 180 deg West (-180 deg) from Greenwich.

Listing 11.3 Regular Expression for GPS Coordinates

```
import re

lat = r'([-]?[0-9]?[0-9][.][0-9]{2,10})'
lon = r'([-]?1?[0-9]?[0-9][.][0-9]{2,10})'
sep = r'[., ]{1,3}'
re_gps = re.compile(lat + sep + lon)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

```
re_gps.findall('http://...maps/@34.0551066,-118.2496763...')
# [(34.0551066, -118.2496763)]
re_gps.findall("https://www.openstreetmap.org/#map=10/5.9666/116.0566")
# [('5.9666', '116.0566')]
groups = re_gps.findall("Zig Zag Cafe is at 45.344, -121.9431 on my GPS.")
# [('45.3440', '-121.9431')]
```

Numerical data is pretty easy to extract, especially if the numbers are part of a machine-readable string. URLs and other machine readable strings put numbers like latitude and longitude in a predictable order, format, and "units" to make things easy for us.

Now this pattern will still accept some out-of-this-world latitude and longitude values, but it gets the job done for most of the URLs you'll copy from mapping web apps like openGoogle maps.

But what about dates? Will regular expressions work for dates? What if we want our date extractor to work in Europe and the US, where the order of day/month is often reversed/reversed?

11.3.3 Dates

Dates are a lot harder to extract than GPS coordinates. Dates are a more natural language, with different dialects for expressing similar things. In the US, Christmas last year would be "12/25/17." In Europe, Christmas would be on "25/12/17." Now you could check the "locale" of your user and assume that they are used to writing dates the same way as others in their region. But this assumption can be wrong.

So most date and time extractors try to work with both kinds of day/month orderings and just check to make sure it's a valid date. This is actually how the human brain works when we read a date like that. Even if you were an US English speaker and you were in Brussels around Christmas time, you'd probably recognize "25/12/17" as a Holiday, because there are only 12 months in the year.

This "duck-typing" approach that works in computer programming can work for natural language too. If it looks like a duck and acts like a duck, then it's probably a duck. If it looks like a date and acts like a date, then it's probably a date. We'll use this "try it and ask forgiveness later" approach for other natural language processing tasks as well. We'll try a bunch of options and accept the one that works. We'll try our extractor or our generator, and then we'll run a validator on it to see if it makes sense.

For chatbots this is a particularly powerful approach, allowing us to combine the best of

multiple natural language generators. In Chapter 10 we generated some chatbot replies using LSTMs. To improve the user experience we could generate a lot of replies and chose the one with the best spelling, grammar, and sentiment.

Listing 11.4 Regular Expression for US Date

```
us = r'(([01]?[\\d][0123]?[\\d)([-/](012]\\d)?[\\d\\d)?[-/]'  
re.findall(us, 'Santa came on 12/25/2017 and a star appeared 12/12')  
[('12/25/2017', '12', '25', '/2017', '20'), ('12/12', '12', '12', '', '')]
```

You can see that even for these simple dates, our regex can't tell whether "12/12" is December 12th, 2017, or December 2012. Sorting that out would require that context was maintained, at least for this sentence. It would also require coding up the logic that dates rarely contain a month without a day. However, no amount of sophistication could resolve the ambiguity in the date "12/11." That could be

- December 11th in whatever year you read or heard it
- November 12th if you heard it in London or Tasmania, Australia
- December 2011 if you read it in a US newspaper
- November 2012 if you read it in an EU newspaper

Some natural language ambiguities can't be resolved, even by a human brain. But let's just make sure our date extractor can handle European day/month order by reversing month and day in our regex.

Listing 11.5 Regular Expression for European Dates

```
eu = r'(([0123]?[\\d][01]?[\\d)([-/](012]\\d)?[\\d\\d)?[-/]'  
re.findall(eu, 'Alan Mathison Turing OBE FRS (23/6/1912-7/6/1954) was an English  
computer scientist.')  
[('23/6/1912', '23', '6', '/1912', '19'),  
 ('7/6/1954', '7', '6', '/1954', '19')]
```

That regular expression correctly extracts Turing's birth and wake dates from a Wikipedia excerpt. But I cheated, I converted the the month "June" into the number 6 and reformatted the dates before testing the regular expression on that Wikipedia sentence. And we need to extract dates from Wikipedia articles if we want our chatbot to be able to read up on famous people and "learn" import dates. For our regex to work on more natural language dates, like those found in Wikipedia articles, we need to add words like "June" to our date-extracting regular expression.

We don't need any special symbols to indicate words (characters that go together in sequence). We can just type them in the regex exactly as we'd like them to be spelled in

the input, including capitalization. All we have to do is put an OR (|) between them in the regular expression. And we need to make sure it can handle US month/day order as well as the European order. We'll add these two alternative date "spellings" to our regular expression with a "big" OR (|) between them as a fork in our tree of decisions in the FSM (regular expression).

Let's use some named groups to help us recognize years like "'84" as 1984 and "08" as 2008. And lets try to be a little more precise about the 4-digit years we want to match, only matching years in the future up to 2399 and in the past back to Year 0.¹⁶⁷

Footnote 167 en.wikipedia.org/wiki/Year_zero

Listing 11.6 Recognizing Years

```

yr_19xx = (
    r'\b(?P<yr_19xx>' +
    '|'.join('{}'.format(i) for i in range(30, 100)) +
    r')\b'
)
1
yr_20xx = (
    r'\b(?P<yr_20xx>' +
    '|'.join('{:02d}'.format(i) for i in range(10)) + '|' +
    '|'.join('{}'.format(i) for i in range(10, 30)) +
    r')\b'
)
2
yr_cent = r'\b(?P<yr_cent>' + '|'.join('{}'.format(i) for i in range(1, 40)) + r')' 3
yr_ccxx = r'(?P<yr_ccxx>' + '|'.join('{:02d}'.format(i) for i in range(0, 100)) + r')\b' 4
yr_xxxx = r'\b(?P<yr_xxxx>(' + yr_cent + ')(' + yr_ccxx + r'))\b'
yr = (
    r'\b(?P<yr>' +
    yr_19xx + '|' + yr_20xx + '|' + yr_xxxx +
    r')\b'
)
groups = list(re.finditer(yr, "0, 2000, 01, '08, 99, 1984, 2030/1970 85 47 `66"))
full_years = [g['yr'] for g in groups]
# ['2000', '01', '08', '99', '1984', '2030', '1970', '85', '47', '66']

```

- ¹ 2-digit years 30-99 1930-1999
- ² 1 or 2-digit years 01-30 2001-2030
- ³ First digits of a 3 or 4-digit yr like the "1" in "123 A.D." or "20" in "2018"
- ⁴ Last 2 digits of a 3 or 4-digit yr like the "23" in "123 A.D." or "18" in "2018"

Wow! That's a lot of work, just to handle some simple year rules in regex rather than in Python. Don't worry, there are packages for recognizing common date formats. They are much more precise (fewer false matches) and more general (fewer misses). So you don't need to be able to compose complex regular expressions like this yourself. This is just to give you a pattern in case you need to extract a particular kind of number using a regular

expression in the future. Monetary values and IP addresses are examples where a more complex regular expression, with named groups, might come in handy.

Let's finish up our regular expression for extracting dates by adding patterns for the month names like "June" or "Jun" in Turing's birthday on Wikipedia dates.

Listing 11.7 Recognizing Month Words with Regular Expressions

```
mon_words = 'January February March April May June July ' \
    'August September October November December'
mon = (r'\b(' + '|'.join('{{}}|{{}|{}|{}|{:02d}}}'.format(
    m, m[:4], m[3], i + 1, i + 1) for i, m in enumerate(mon_words.split())) + 
    r')\b')
re.findall(mon, 'January has 31 days, February the 2nd month of 12, has 28,
    except in a Leap Year.')
['January', 'February', '12']
```

Can you see how we might combine these regular expressions into a larger one that can handle both EU and US date formats? One complication is that you can't reuse the same name for a group (parenthesized part of the regular expression). So we can't just put an OR between the US and EU ordering of the named regular expressions for month and year. And we need to include patterns for some optional separators between the day, month, and year.

Here's one way to do all that.

Listing 11.8 Combining Information Extraction Regular Expressions

```
day = r'|'.join('{{:02d}}|{{}}'.format(i, i) for i in range(1, 32))
eu = (r'\b(' + day + r')\b[-/ ]{0,2}\b(' +
    mon + r')\b[-/ ]{0,2}\b(' + yr.replace('<yr', '<eu_yr') + r')\b')
us = (r'\b(' + mon + r')\b[-/ ]{0,2}\b(' +
    day + r')\b[-/ ]{0,2}\b(' + yr.replace('<yr', '<us_yr') + r')\b')
date_pattern = r'\b(' + eu + '| ' + us + r')\b'
list(re.finditer(date_pattern, '31 Oct, 1970 25/12/2017'))
# [<sre.SRE_Match object; span=(0, 12), match='31 Oct, 1970'>,
#   <sre.SRE_Match object; span=(13, 23), match='25/12/2017'>]
```

Finally, we need to validate these dates by seeing if they can be turned into valid Python datetime objects.

```
import datetime
dates = []
for g in groups:
    month_num = (g['us_mon'] or g['eu_mon']).strip()
    try:
        month_num = int(month_num)
    except ValueError:
        month_num = [w[:len(month_num)] for w in mon_words].index(month_num) + 1
    date = datetime.date(
        int(g['us_yr']) or g['eu_yr']),
```

```

month_num,
int(g['us_day'] or g['eu_day']))
dates.append(date)
dates
# [datetime.date(1970, 10, 31), datetime.date(2017, 12, 25)]

```

So it looks like our date extractor works OK, at least for a few simple, unambiguous dates. Think about how packages like `Python-dateutil` and `datefinder` are able to resolve ambiguities and deal with more "natural" language dates like "today" and "next Monday." And if you think you can do it better than these packages, send them a pull request!

11.4 Extracting Relationships (*Relations*)

So far we've only looked at extracting tricky noun instances like dates and GPS latitude and longitude values. And we've worked mainly with numerical patterns. It's time we tackle the harder problem of extracting knowledge from natural language. We'd like our bot to learn facts about the world from reading an encyclopedia of knowledge like Wikipedia. We'd like it to be able to relate those dates and GPS coordinates to the entities it reads about.

What knowledge could your brain extract from this sentence from Wikipedia:

On March 15, 1554, Desoto wrote in his journal that the Pascagoula people ranged as far north as the confluence of the Leaf and Chickasawhay rivers at 30.4, -88.5.

Extracting the dates and the GPS coordinates might enable you to associate that date and location with Desoto, the Pascagoula people and two rivers whose names you can't pronounce. You'd like your bot (and your mind) to be able to connect those facts to larger facts—like the fact that Desoto was a Spanish conquistador and that the Pascagoula people were a peaceful native American tribe. And, of course you'd like the dates and locations to be associated with the right "things", Desoto, and the intersection of two rivers, respectively.

This is what most people think of when they hear the term *natural language understanding*. To understand a statement you need to be able to extract key bits of information and correlate it with related knowledge. For machines we store that knowledge in a graph, also called a knowledge base. The edges of our knowledge graph are the relationships between things. And the nodes of our knowledge graph are the nouns or objects found in our corpus.

The pattern we're going to use to extract these relationships (or *relations*) is a pattern like

SUBJECT - VERB - OBJECT. To recognize these patterns we'll need our NLP pipeline to know the parts of speech (POS) for each word in a sentence.

11.4.1 POS Tagging

POS tagging can be accomplished with language models that contain dictionaries of words with all their possible parts of speech. They can then be trained on properly tagged sentences to recognize the parts of speech in new sentences with other words from that dictionary. NLTK and SpaCy both implement POS tagging functions. We'll use SpaCy here because it is faster and more accurate.

```
import spacy
en_model = spacy.load('en_core_web_md')
sentence = ("In 1541 Desoto wrote in his journal that the Pascagoula people " +
            "ranged as far north as the confluence of the Leaf and Chickasawhay
            rivers at 30.4, -88.5.")
parsed_sent = en_model(sentence)
parsed_sent.ents
# (1541, Desoto, Pascagoula, Leaf, Chickasawhay, 30.4) ①
[(tok, tok.tag_) for tok in parsed_sent]
' '.join(['{} {}'.format(tok, tok.tag_) for tok in parsed_sent])
# 'In_IN 1541_CD Desoto_NNP wrote_VBD in_IN his_PRP$'
#     journal_NN that_IN the_DT Pascagoula_NNP people_NNS
#     ranged_VBD as_RB far_RB north_RB as_IN the_DT confluence_NN of_IN the_DT Leaf_NNP
#     and_CC Chickasawhay_NNP
#     rivers_VBZ at_IN 30.4_CD ,_, -88.5_NFP ._.'
②
```

- ① spaCy misses the longitude in the lat, lon numerical pair.
- ② spaCy uses the "OntoNotes 5" POS tags: spacy.io/api/annotation#pos-tagging

So to build our knowledge graph, we just need to figure out which objects (noun phrases) should be paired up. We'd like to pair up the date "March 15, 1554" with the "named entity" Desoto. We could normalize those two strings (noun phrases) to point to objects that we have in our knowledge base. March 15, 1554 can be converted to a `datetime.date` object with a normalized representation.

SpaCy parsed sentences also contain the dependency tree in a nested dictionary. And `spacy.displacy` can generate an SVG string (optionally, within a complete HTML page) which can be viewed in a browser to help us find ways to use the tree to create tag patterns for relation extraction.

Listing 11.9 Visualize a Dependency Tree

```
from spacy.displacy import render
sentence = "In 1541 Desoto wrote in his journal about the Pascagoula."
```

```

parsed_sent = en_model(sentence)
with open('pascagoula.html', 'w') as f:
    f.write(displacy.render(docs=tagged_sentence, page=True, options=dict(compact=True)))

```

The dependency tree for this short sentence shows the noun phrase "the Pascagoula" is the object of the relationship met for the subject "Desoto." And both nouns are tagged as proper nouns.

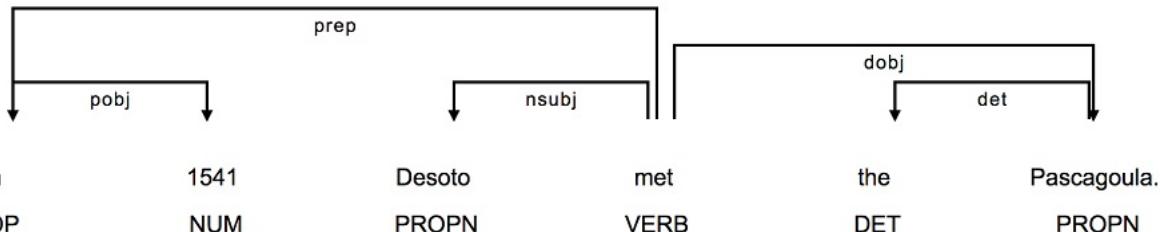


Figure 11.2 The Pascagoula People: Dependency tree for sentence about the Pascagoula people

To create POS and word property patterns for a `spacy.matcher.Matcher` it's helpful to list all the token tags in a table. Here are some helper functions to make that easier:

```

import pandas as pd
from collections import OrderedDict

def token_dict(token):
    return OrderedDict(ORTH=token.orth_, LEMMA=token.lemma_,
                      POS=token.pos_, TAG=token.tag_, DEP=token.dep_)

def doc_dataframe(doc):
    return pd.DataFrame([token_dict(tok) for tok in doc])

doc_dataframe(en_model("In 1541 Desoto met the Pascagoula."))
#      ORTH      LEMMA    POS   TAG     DEP
# 0      In       in     ADP   IN   prep
# 1    1541    1541     NUM   CD  pobj
# 2   Desoto  desoto   PROPN  NNP  nsubj
# 3     met    meet   VERB  VBD  ROOT
# 4     the     the   DET   DT   det
# 5  Pascagoula  pascagoula   PROPN  NNP  dobj
# 6      .        .  PUNCT   .  punct

```

Now we can see the sequence of POS or TAG features that will make a good pattern for us. If we're looking for 'has-met' relationships between people and organizations, we'd probably like to allow patterns like 'PROPN met PROPN', 'PROPN met the PROPN', 'PROPN met with the PROPN', 'PROPN often meets with PROPN'. WE could specify each of those patterns of individually, or try to capture them all with some '*' or '?' operators on 'any word' patterns between our proper nouns:

```
'PROPN ANYWORD? met ANYWORD? ANYWORD? PROPN'
```

spaCy patterns are much more powerful and flexible so we have to be more verbose to explain exactly the features of the words we'd like to match, using a dictionary for each word rather than a single symbol, like for the pseudocode above:

```
pattern = [{ 'TAG': 'NNP', 'OP': '+'}, { 'IS_ALPHA': True, 'OP': '*'},  
           {'LEMMA': 'meet'},  
           { 'IS_ALPHA': True, 'OP': '*'}, { 'TAG': 'NNP', 'OP': '+'}]
```

We can then extract the tagged tokens we need from our parsed sentence like this:

```
from spacy.matcher import Matcher  
doc = en_model("In 1541 Desoto met the Pascagoula.")  
matcher = Matcher(en_model.vocab)  
matcher.add('met', None, pattern)  
m = matcher(doc)  
m  
# [(12280034159272152371, 2, 6)]  
doc[m[0][1]:m[0][2]]  
# Desoto met the Pascagoula
```

So we extracted a match from the original sentence that we created the pattern from, but what about similar sentences from Wikipedia.

```
doc = en_model("October 24: Lewis and Clark met their first Mandan Chief, Big White.")  
m = matcher(doc)[0]  
m  
# (12280034159272152371, 3, 11)  
doc[m[1]:m[2]]  
# Lewis and Clark met their first Mandan Chief  
doc = en_model("On 11 October 1986, Gorbachev and Reagan met at Höfði house")  
matcher(doc)  
# [] ①
```

① Our pattern doesn't match any substrings of the sentence from Wikipedia

We need to add a second pattern to allow for the verb to occur after the subject and object nouns.

```
doc = en_model("On 11 October 1986, Gorbachev and Reagan met at Hofoi house")  
pattern = [{ 'TAG': 'NNP', 'OP': '+'}, { 'LEMMA': 'and' }, { 'TAG': 'NNP', 'OP': '+'},  
           { 'IS_ALPHA': True, 'OP': '*'}, { 'LEMMA': 'meet' }]  
matcher.add('met', None, pattern) ①  
m = matcher(doc)  
m  
# [(14332210279624491740, 5, 9),  
#   (14332210279624491740, 5, 11),  
#   (14332210279624491740, 7, 11),  
#   (14332210279624491740, 5, 12)] ②  
doc[m[-1][1]:m[-1][2]] ③  
# Gorbachev and Reagan met at Hofoi house
```

- ① adds an additional pattern without removing the previous pattern
- ② the '+' operators increase the number of overlapping alternative matches
- ③ the longest match is the last one in the list of matches

So now we have our entities and a relationship. We can even build a pattern that is less restrictive about the verb in the middle ("met") and more restrictive about the names of the people and groups on either side. This might allow us to identify additional verbs that imply that one person or group has met another, like the verb "knows" or even passive phrases like "had a conversation" or "became acquainted with". Then we could use these new verbs to add additional relationships for new proper nouns on either side.

However, you can see how this would begin to drift away from the original meaning of our seed relationship patterns. This is called semantic drift. Fortunately, spaCy tags words in a parsed document with not only their POS and dependency tree information, it also provides the Word2Vec word vector. This can be used to prevent the connector verb and the proper nouns on either side from drifting too far away from the original meaning of our seed pattern.¹⁶⁸

Footnote 168 This is the subject of active research: nlp.stanford.edu/pubs/structuredVS.pdf

11.4.2 Entity Name Normalization

The normalized representation of an entity is usually a string, even for numerical information like dates. The normalized ISO format for this date would be "1541-01-01". A normalized representation for entities enables our knowledge base to connect all the different things that happened in the world on that same date to that same node (entity) in our graph.

We'd do the same for other named entities. We'd correct the spelling of words and attempt to resolve ambiguities for names of objects, animals, people, places, etc. Normalizing named entities and resolving ambiguities is often called "coreference resolution" or "anaphora resolution", especially for pronouns or other "names" relying on context. This is similar to lemmatization that we discussed in Chapter 2. Normalization of named entities ensures that spelling and naming variations don't pollute our vocabulary of entity names with confounding, redundant names.

For example "Desoto" might be expressed in a particular document in at least five different ways:

1. "de Soto"
2. "Hernando de Soto"
3. "Hernando de Soto (c. 1496/1497–1542), Spanish conquistador"
4. en.wikipedia.org/wiki/Hernando_de_Soto:[en.wikipedia.org/wiki/Hernando_de_Soto] (a URI)
5. A numerical ID for a database of famous and historical people

Similarly our normalization algorithm can chose any of these forms. A knowledge graph should normalize each kind of entity the same way, to prevents multiple distinct entities of the same type from sharing the same "name." We don't want multiple person names referring to the same physical person. Even more importantly the normalization should be applied consistently—both when we write new facts to the knowledge base or when we read or query the knowledge base.

If we decide to change the normalization approach after the database has been populated, then the data for existing entities in the knowledge should be "migrated", altered, to adhere to the new normalization scheme. Schemaless databases (key-value stores), like the ones used to store knowledge graphs or knowledge bases are not free from the migration responsibilities of relational databases. After all, schemaless databases are interface wrappers for relational databases under the hood.

Our normalized entities also need to be connected by "is-a" relationships to entity categories that define types or categories of entities. These "is-a" relationships can be thought of as tags because each entity can have multiple "is-a" relationships. Our ", like "date" What about *relations* between entities, do they need to be stored in some normalized way?

11.4.3 Relation Normalization and Extraction

Now we need to a way to normalize the relationships, to identify the kind of relationship between entities. This will allow us to find all birthday relationships between dates and people, or dates of occurrences of historical events, like the encounter between "Hernando de Soto" and the "Pascagoula people." And we need to write an algorithm to chose the right label for our relationship.

And these relationships can have a hierarchical name, like "occurred-on/approximately" and "occurred-on/exactly", to allow us to find specific relationships or categories of relationships. Relationships can also be labeled with a numerical property for the "confidence", probability, weight, or normalized frequency (analogous to TFIDF for terms/words) of that relationship. These confidence values can be adjusted each time a fact extracted from a new text corroborates or contradicts an existing fact in the database.

Now we need a way to match patterns that can find these relationships.

11.4.4 Word Patterns

Word patterns are just like regular expressions, but for words instead of characters. Instead of character classes we have word classes. So, for example, instead of matching a lowercase character we might have a word pattern decision to match all the singular nouns ("NN" POS tag).¹⁶⁹ This is usually accomplished with machine learning. Some seed sentences are tagged with some correct relationships (facts) extracted from those sentences. A POS pattern can be used to find similar sentences where the subject and object words might change or even the relationship words.

Footnote 169 spaCy uses the "OntoNotes 5" POS tags: spacy.io/api/annotation#pos-tagging

There are two ways to use the spaCy package can be used to match these patterns in $O(1)$ (constant time, no matter how many patterns you want to match).

- [PhraseMatcher](#) for any word/tag sequence patterns)
- [Matcher](#) for POS tag sequence patterns

To ensure that the new *relations* found in new sentences are truly analogous to the original seed (example) relationships, it's often necessary to constrain the subject, relation, and object word meanings to be similar to those in the seed sentences. This can best be done with some vector representation of the meaning of words. Does this ring a bell? Word vectors from Chapter 4 are one of the most widely used word meaning representations used for this purpose.

This helps minimize *semantic drift*.

Using semantic vector representations words and phrases has made automatic information extraction accurate enough to build large knowledge bases automatically. However, human supervision and curation is required to resolve much of the ambiguity in natural language text. [CMU's NELL](#) (Never Ending Language Learning) enables users to vote on changes to the knowledge base using Twitter and a web application.

11.4.5 Segmentation

We've skipped one form of information extraction or tool used in information extraction. Most of the documents we've used in this chapter have been bite-sized chunks containing just a few facts and named entities. But in the real world you may need to create these chunks yourself.

Document "chunking" is useful for creating semi-structured data about documents which can make it easier to search, filter and sort documents for *information retrieval*. And for information extraction, if we are extracting relations to build a knowledge base like NELL or Freebase, we need to break it into parts that are likely to contain a fact or two. This "chunking" of a document is called *segmentation*. When we divide natural language text into meaningful pieces it's called *segmentation*. The resulting segments can be phrases, sentences, quotes, paragraphs, or even entire sections of a long document.

Sentences are the most common chunk for most information extraction problems. Sentences are usually punctuated with one of a few symbols (".", "?", "!", or a newline). And grammatically correct English language sentences must contain a subject (noun) and a verb. This means they'll usually have at least one relation or fact worth extracting. And sentences are often self-contained packets of meaning that don't rely too much on preceding text to convey most of their information.

Fortunately most languages, including English, have the concept of a sentence, a single statement with a subject and verb that says something about the world. This is just the right bite-sized chunk of text for our NLP knowledge extraction pipeline. For the chatbot pipeline our goal is to segment documents into sentences, statements.

In addition to facilitating information extraction, we can flag some of those statements and sentences as being part of a dialog or being suitable for replies in a dialog. Using a sentence segmenter allows us to train our chatbot on longer texts, like books. If we chose those books appropriately, this will give our chatbot a more literary, intelligent style than if we trained it purely on Twitter streams or IRC chats. And it will give our chatbot access to a much broader set of training documents to build its common sense knowledge about the world.

SENTENCE SEGMENTATION

Sentence Segmentation is usually the first step in an information extraction pipeline. It helps isolate facts from each other so that you can associate the right price with the right thing in a string like "The Babel fish costs \$42. 42 cents for the stamp." And that string is a good example of why sentence segmentation is tough—the dot in the middle could be interpreted as a decimal or a "full stop" period.

One of the simplest pieces of "information" we can extract from a document are sequences of words that contain a logically cohesive statement.

The most important segments in a natural language document, after words, are *sentences*.

Sentences contain a logically cohesive statement about the world. These statements contain the information that we want to extract from text. Sentences often tell us the relationship between things and how the world works when they make statements of fact. So we can use sentences for *knowledge extraction*. And sentences often explain when, where, and how things happened in the past, tend to happen in general, or will happen in the future. So we should also be able to extract dates, times, locations, and processes using sentences as our guide. And, most importantly, all natural languages have sentences or logically cohesive sections of text of some sort. And all languages have a widely shared process for generating them (a set of grammar "rules" or habits).

But segmenting text, identifying sentence boundaries is a bit trickier than you might think. In English, there is no single punctuation mark or sequence of characters that always marks the end of a sentence.

11.4.6 `split('.!?'')` Won't Work

I live in the U.S. but I commute to work in Mexico on S.V. Australis for a woman from St. Bernard St. on the Gulf of Mexico.

I went to G.T.You?

She yelled "It's right here!" but I kept looking for a sentence boundary anyway.

I stared dumbfounded on as things like "How did I get here?", "Where am I?", "Am I alive?" fluttered across the screen.

The author wrote "I don't think it's conscious.' Turing said."

Even a human reader might have trouble finding an appropriate sentence boundary within each of these quotes. And if they did find multiple sentences from each of the quotes above, they would be wrong for 4 out of 5 of these difficult examples.

More sentence segmentation "edge cases" like this are available at tm-town.com and within the nlpia.data module.

Technical text is particularly difficult to segment into sentences because engineers, scientists, and mathematicians tend to use periods and exclamation points to signify a lot of things besides the end of a sentence. When we tried to find the sentence boundaries in this book, we had to manually correct several of the extracted sentences.

If only we wrote English like telegrams, with a "STOP" or unique punctuation mark at the end of each sentence. But since we don't, we'll need some more sophisticated NLP than just `split('.!?'')`. Hopefully you're already imagining a solution in your head. If so it's probably based on one of the two approaches to NLP we've used throughout this book

1. Manually-programmed algorithms (regular expressions and pattern-matching)
2. Statistical models (data-based models or machine learning)

We'll use the sentence segmentation problem to revisit these two approaches by showing you how to use regular expressions as well as *perceptrons* to find sentence boundaries. And we'll use the text of this book as a training and test set to show you some of the challenges. Fortunately we haven't inserted any newlines within sentences, to manually "wrap" text like in newspaper column layouts. Otherwise the problem would be even more difficult. In fact much of the source text for this book, in ASCIIDoc format, has been written with "old-school" sentence separators (two spaces after the end of every sentence), or with each sentence on a separate line. This was so we could use this book as a training and test set for our segmenters.

11.4.7 Sentence Segmentation with Regular Expressions

Regular expressions are just a shorthand way of expressing the tree of if...then rules (*Regular Grammar* rules) for finding character patterns in strings of characters. As we mentioned in Chapter 1 and 2, regular expressions (regular grammars) are a particularly succinct way to specify the workings of a *Finite State Machine* (FSM). Our regex or FSM has only one purpose—identify sentence boundaries.

If you do a [web search for sentence segmenters](#), you're likely to be pointed to various regular expressions intended to capture the most common sentence boundaries. Here are some of them, combined and enhanced to give us a fast, general-purpose sentence segmenter.

The following regex would work with a few "normal" sentences.

```
>>> re.split(r'![.?]+[ $]', "Hello World.... Are you there?!?! I'm going to Mars!")
['Hello World', 'Are you there', "I'm going to Mars!"]
```

Unfortunately, this `re.split` approach gobbles up the sentence-terminating token, but only if it isn't the last character in a document. It also does a good job of ignoring the trickery of periods within doubly-nested quotes:

```
>>> re.split(r'[^.?] ', "The author wrote \"I don't think it's conscious.' Turing said.\")\n['The author wrote "'I don't think it's conscious.' Turing said."']
```

Unfortunately it also ignores periods in quotes that terminate an actual sentence:

```
>>> re.split(r'[^.?] ', "The author wrote \"I don't think it's conscious.' Turing said.\" But I stopped\n['The author wrote "'I don't think it's conscious.' Turing said." But I stopped reading."]')
```

What about abbreviated text, like SMS messages and tweets? Sometimes hurried humans squish sentences together, leaving no space surrounding periods. Alone, the regex below could only deal with periods in SMS messages that have letters on either side, and it would safely skip over numerical values:

```
>>> re.split(r'(?<!\\d)\\.|\\.(?![\\d])', "I went to GT.You?")
['I went to GT', 'You?']
```

Even combining these two regexes isn't enough to get more than a few right in the difficult test cases from `nlpia.data`.

```
>>> from nlpia.data.loaders import get_data
>>> regex = re.compile(r'((?<!\\d)\\.|\\.(?![\\d]))|([!.?]+)[ $]+')
>>> examples = get_data('sentences-tm-town')
>>> wrong = []
>>> for i, (challenge, text, sents) in enumerate(examples):
>>>     if tuple(regex.split(text)) != tuple(sents):
>>>         print('wrong {}: {}'.format(i, text[:50], '...' if len(text) > 50 else ''))
>>>         wrong += [i]
>>> len(wrong), len(examples)
(61, 61)
```

We'd have to add a lot more "look-ahead" and "look-back" to improve the accuracy of a regex sentence segmenter. A better approach for sentence segmentation is to use a machine learning algorithm (often a single-layer neural net or logistic regression) trained on a labeled set of sentences. Several packages contain such a model that you can use to improve your sentence segmenter.

- [DetectorMorse](#)
- [spaCy](#)
- [SyntaxNet](#)
- [NLTK \(Punkt\)](#)
- [Stanford CoreNLP](#)

We use the spaCy sentence segmenter (built into the parser) for most of our mission-critical applications. SpaCy has few dependencies and compares well with the

others on accuracy and speed. DetectorMorse, by Kyle Gorman, is another good choice if you want state-of-the-art performance in a pure Python implementation that you can refine with your own training set.

11.5 In the Real World

Information extraction and question answering systems are used for:

- TA Assistants for University Courses
- Customer service
- Tech support
- Sales
- Software documentation and FAQs

Information extraction can be used to extract things like:

- dates
- times
- prices
- quantities
- addresses
- names
 - people
 - places
 - apps
 - companies
 - bots
- relationships
 - "is-a" (kinds of things)
 - "has" (attributes of things)
 - "related-to"

11.6 Summary

Whether information is being parsed from a large corpus or from user input on the fly, being able to extract specific details and store them for later use is critical to the performance of a chatbot. First by identifying and isolating this information and then by tagging relationships between those pieces of information we have learned to "normalize" information programmatically. With that knowledge safely shelved in a search-able structure, our chatbot will be equipped with the tools to hold its own in a conversation in a given domain.

In this chapter we learned

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

- A knowledge graph can be built to store relationships between entities
- Regular expressions are a mini-programming language that are path to isolating specific types of information
- Part of Speech Tagging allows us to reason about relationships between tokens in a sentence
- Segmenting sentences is more than just split on periods

12

Getting Chatty (Dialog Engines)

In this chapter you will learn about

- Four ways to build a chatbot
- What Artificial Intelligence Markup Language (AIML) is all about
- What makes a Chatbot pipeline different from other NLP pipelines
- A hybrid chatbot architecture that combines the best ideas into one
- How to use machine learning to make your chatbot get smarter and smarter over time
- "Agency"—simulating spontaneous statements with a chatbot

We introduced this book with the idea of a dialog engine or chatbot NLP pipeline because we think it's one of the most important NLP applications of this century. For the first time in history we can speak to a machine in our own language, and we can't always tell that it isn't human. This means that machines can "fake" being human. That is a lot harder than it sounds. There are several cash prize competitions that you can enter, if you think you and your chatbot have the right stuff.

- The Alexa Prize (\$3.5M)¹⁷⁰

Footnote 170 "The Alexa Prize" developer.amazon.com/alexaprize

- Loebner Prize (\$7k)¹⁷¹

Footnote 171 "Loebner Prize" @ Bletchley Park, www.aisb.org.uk/events/loebner-prize

- The Winograd Schema Challenge (\$27k)¹⁷²

Footnote 172 "Establishing a Human Baseline for the Winograd Schema Challenge" by David Bender, ceur-ws.org/Vol-1353/paper_30.pdf, "An alternative to the Turing test", Kurzweil, www.kurzweilai.net/an-alternative-to-the-turing-test-winograd-schema-challenge-annual-competition-announced

- The Marcus Test ¹⁷³

Footnote 173 "What Comes After the Turing Test", New Yorker, Jan 2014,
www.newyorker.com/tech/elements/what-comes-after-the-turing-test

- The Lovelace Test ¹⁷⁴

Footnote 174 "The Lovelace 2.0 Test of Artificial Creativity and Intelligence" by Reidl,
arxiv.org/pdf/1410.6142.pdf

Beyond the pure fun and magic of building a conversational machine, beyond the glory that awaits you if you build a bot that can beat humans at an IQ test, beyond the warm fuzzy of saving the world from malicious hacker bot nets, and beyond the wealth that awaits you if you can beat Google and Amazon at their virtual assistant games—the techniques you'll learn in this chapter will give you the tools you need to just get the job done.

The 21st century is going to be built on a foundation of AI (Artificial Intelligence) that assists us. And the most natural interface for AI is natural language conversation. For example, Aira.io's chatbot Chloe is helping to interpret the world to our blind and low-vision customers to help them explore the world in ways they only dreamed of before. Others are building lawyer chatbots that save users thousands of dollars (or pounds) on parking tickets and hours of courtroom time. And self-driving cars will likely soon have conversational interfaces similar to Google Assistant and Google Maps to help you get where you want to go.

12.1 Language Skill

We finally have all the pieces we need to assemble a chatbot—more formally, a *dialog system* or *dialog engine*. We'll build an NLP pipeline that can participate in natural language conversations.

Some of your NLP skills we'll use include:

- Tokenization, stemming and lemmatization
- Vector space language models like bag-of-words vectors and topic vectors (semantic vectors)
- Nonlinear language representations like word2vec word vectors or LSTM "thought vectors"
- Sequence-to-sequence translators from Chapter 10
- Pattern matching and templates

With these tools we can build a chatbot with interesting behavior.

Let's make sure we're on the same page about what a chatbot is. In some communities, the word "Chatbot" is used in a slightly derogatory way to refer to "canned response" systems.¹⁷⁵ These are chatbots that find patterns in the input text and use matches on those patterns to trigger a fixed, or templated response.¹⁷⁶ You can think of these as FAQ bots that only know the answers to basic, general questions. These basic dialog systems are useful mainly for automated customer service phone-tree systems, where it's possible to hand off the conversation to a human when the chatbot runs out of "canned" responses.

Footnote 175 Wikipedia "Canned Response" en.wikipedia.org/wiki/Canned_response

Footnote 176 "A Chatbot Dialogue Manager" by A.F. van Woudenberg, Open University of the Netherlands, dspace.ou.nl/bitstream/1820/5390/1/INF_20140617_Woudenberg.pdf

But this doesn't mean that your chatbot needs to be so limited. If you are particularly clever about these patterns and templates, your chatbot can be the therapist in a convincing psychotherapy or counseling session. All the way back in 1964, Joseph Weizenbaum used patterns and templates to build the first popular chatbot, ELIZA.¹⁷⁷ And the remarkably effective Facebook Messenger therapy bot, Woebot, relies heavily on the pattern-matching and templated response approach. All that's needed to build Turing award-winning chatbots is to add a little state (context) management to your pattern-matching system. Amazon recently added this additional layer of conversational depth to Alexa, where it's called it "Follow-Up Mode."¹⁷⁸ And we'll show you how to do that here.

Footnote 177 Wikipedia: en.wikipedia.org/wiki/ELIZA

Footnote 178 www.theverge.com/2018/3/9/17101330/amazon-alexa-follow-up-mode-back-to-back-requests

12.1.1 Modern Approaches

Chatbots have come a long way since the days of ELIZA. Pattern matching technology has been generalized and refined over the decades. And completely new approaches have been developed to supplement pattern matching. In recent literature, chatbots are often referred to as dialog systems, perhaps because of this greater sophistication. Matching patterns in text and populating "canned response" templates with information extracted with those patterns is just one of four modern approaches to building chatbots:

1. *Pattern Matching*: pattern matching and response templates ("canned" responses)
2. *Grounding*: logical knowledge graphs and inference on those graphs
3. *Search*: text retrieval
4. *Generative*: statistics and machine learning

This is roughly the order in which these approaches were developed. And that's the order we'll present them here. But before showing you how to use each of these techniques to generate replies, we'll show you how chatbots use these techniques in the real world.

The most advanced chatbots use a hybrid approach that combines all of these techniques. This hybrid approach enables them to accomplish a broad range of tasks. We've listed a few of these chatbot applications below. You may notice that the more advanced chatbots, like Siri, Alexa, and Allo, are listed along side multiple types of problems and applications:

- *Question Answering*: Google Search, Alexa, Siri, Watson
- *Virtual Assistants*: Google Assistant, Alexa, Siri, MS paperclip
- *Conversational*: Google Assistant, Google Smart Reply, Mitsuki Bot
- *Marketing*: Twitter bots, blogger bots, Facebook bots, Google Search, Google Assistant, Alexa, Allo
- *Customer Service*: Storefront bots, technical support bots
- *Community Management*: Bonusly, Slackbot
- *Therapy*: WoeBot, Wysa, YourDost, Siri, Allo

Can you think of ways to combine the 4 basic dialog engine types to create chatbots for these seven applications above? Here's how some chatbots do it:

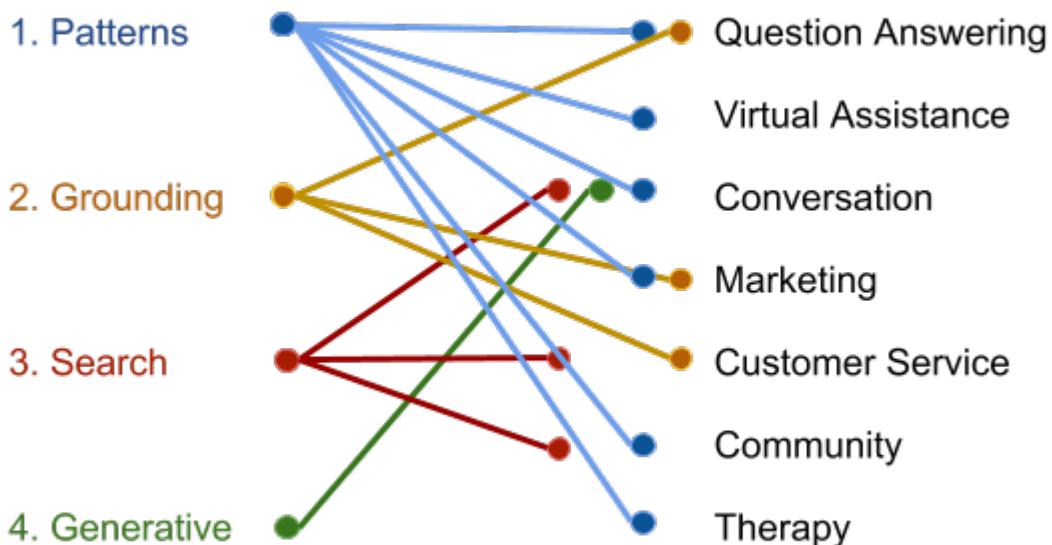


Figure 12.1 Chatbot Techniques Used for Some Example Applications

Let's talk briefly about these applications to help you build a chatbot for your

application.

QUESTION ANSWERING DIALOG SYSTEMS

Question answering chatbots are used to answer factual questions about the world, which can include questions about the chatbot itself. Many question-answering systems first search a knowledge base or relational database to "ground" them in the real world. If they can't find an acceptable answer there, then they may search a corpus of unstructured data (or even the entire Web) to find answers to your questions. This is essentially what Google Search does. Parsing a statement to discern the question that needs to be answered and then picking the right answer requires a complex pipeline that combines most of the elements covered in the previous chapters. Question answering chatbots are the most difficult to implement well because they require coordinating of so many different elements.

VIRTUAL ASSISTANTS

Virtual assistants, like Alexa and Google Assistant, are helpful when you have a goal in mind. Goals or intents are usually simple things like launching an app, setting a reminder, playing some music, or turning on the lights in your home. For this reason, virtual assistants are often called goal-based dialog engines. Dialog with such chatbots is intended to conclude quickly, with the user being satisfied that a particular action has been accomplished or some bit of information has been retrieved.

You're probably familiar with the virtual assistants on your phone or your home automation system. But you may not know that virtual assistants can also help you with your legal troubles and taxes. Though Intuit's TurboTax wizards aren't very chatty, they do implement a very complex phone tree. However you don't interact with them by voice or chat, but by filling in forms with structured data. So the TurboTax "wizard" can't really be called a chatbot yet, but it will surely be wrapped in a chat interface soon, if the taxbot "AskMyUncleSam" takes off.¹⁷⁹

Footnote 179 Jan 2017, Venture Beat post by AskMyUncleSam:
venturebeat.com/2017/01/27/how-this-chatbot-powered-by-machine-learning-can-help-with-your-taxes/

There are lawyer virtual assistant chatbots that have successfully appealed millions of dollars in parking tickets in New York and London.¹⁸⁰ And there's even a United Kingdom law firm where the only interaction you'll ever have with a lawyer is through a chatbot.¹⁸¹ Lawyers are certainly goal-based virtual assistants, only they'll do more than set an appointment date, they'll set you a court date and maybe help you win your case.

Footnote 180 "Chatbot Lawyer DoNotPay Chatbot Lawyer Overturns 160,000 Parking Tickets in London and New York", June 2016, The Guardian,
www.theguardian.com/technology/2016/jun/28/chatbot-ai-lawyer-donotpay-parking-tickets-london-new-york

Footnote 181 Nov 2017, "The law firm without lawyers" blog post by Legal Futures:
www.legalfutures.co.uk/latest-news/chatbot-based-firm-without-lawyers-launched

Aira.io¹⁸² is building a virtual assistant called Chloe. Chloe gives blind and low vision people access to a "visual interpreter for the blind". During on-boarding, Chloe can ask customers things like "Are you a white cane user?", "Do you have a guide dog?", and "Do you have any food allergies or dietary preferences you'd like us to know about?" This is called "voice first" design, when your app is designed from the ground up around a dialog system. In the future, the assistance that Chloe can provide will be greatly expanded as she learns to understand the real world through live video feeds. And the "Explorers" around the world interacting with Chloe will be training her to understand common every day tasks that humans perform in the world. Chloe is one of the few virtual assistants designed entirely to assist and not to influence or manipulate.¹⁸³

Footnote 182 aira.io

Footnote 183 We rarely acknowledge to ourselves the influence that virtual assistants and search engines exert over our free will and beliefs. And we rarely care that their incentives and motivations are different from our own. These misaligned incentives are present not only in technology like virtual assistants, but within culture itself. Check out *Sapiens* and *Homo Deus* by Yuval Noah Harari if you're interested in learning about where culture and technology are taking us.

Virtual assistants like Siri, Google Assistant, Cortana, and Aira's Chloe are getting smarter every day. Virtual assistants learn from their interactions with humans and the other machines they are connected to. They're developing ever more general, domain-independent intelligence. If you want to learn about Artificial General Intelligence (AGI), then you'll want to experiment with virtual assistants and conversational chatbots as part of that research.

CONVERSATIONAL CHATBOTS

Conversational chatbots, like Worswick's *Mitsuku*¹⁸⁴ or any of the Pandorabots,¹⁸⁵ are designed to entertain. They can often be implemented with very few lines of code, as long as you have lots of data. But doing conversation well is an ever evolving challenge. The "accuracy" or performance of a conversational chatbot is usually measured with something like a Turing test. In a typical Turing test, humans interact with another chat participant through a terminal and try to figure out if it is a bot or a human. The better the chatbot is at being indistinguishable from a human, the better its performance on a Turing test metric.

Footnote 184 www.square-bear.co.uk/aiml

Footnote 185 www.chatbots.org/directory/search_results/7f346ca4d93aa44175f79ee39e9af96c/

The domain (variety of knowledge) and human behaviors that a chatbot is expected to implement, in these Turing tests, is expanding every year. And as the chatbots get better at fooling us, we humans get better at detecting their trickery. Eliza fooled many of us in the BBS-era of the 80's into thinking that "she" was a therapist helping us get through our daily lives. It took decades of research and development before chatbots could fool us again.

Fool me once, shame on *bots*; fool me twice, shame on *humans*.

-- Anonymous Human

Recently Mitsuku won the Loebner challenge, a competition that uses a Turing test to rank chatbots.^{186]} Conversational chatbots are used mostly for academic research, entertainment (video games), and advertisement.

Footnote 186 footnote:[en.wikipedia.org/wiki/Loebner_Prize

MARKETING CHATBOTS

Marketing chatbots are designed to inform users about a product and entice them to purchase it. More and more video games, movies, and TV shows are launched with chatbots on websites promoting them:¹⁸⁷

Footnote 187 Justin Clegg lists additional ones in his LinkedIn post:

www.linkedin.com/pulse/how-smart-brands-using-chatbots-justin-clegg/

- HBO promoted "Westworld" with "Aeden."¹⁸⁸

Footnote 188 Sep 2016, Entertainment Weekly:

www.yahoo.com/entertainment/westworld-launches-sex-touting-online-181918383.html

- Sony promoted "Resident Evil" with "Red Queen."¹⁸⁹

Footnote 189 Jan 2017, IPG Media Lab:

www.ipglab.com/2017/01/18/sony-pictures-launches-ai-powered-chatbot-to-promote-resident-evil-movie/

- Disney promoted Zootopia" with "Officer Judy Hopps."¹⁹⁰

Footnote 190 Jun 2016, Venture Beat:

venturebeat.com/2016/06/01/imperson-launches-zootopias-officer-judy-hopps-bot-on-facebook-messenger/

- Universal promoted "Unfriended" with "Laura Barnes."
- Activision promoted "Call of Duty" with "Lt. Reyes"

Some virtual assistants are actually marketing bots in disguise. Consider Amazon Alexa and Google Assistant, though they claim to assist you with things like adding reminders and searching the web, they invariably prioritize responses about products or businesses over responses with generic or free information. These companies are in the business of selling stuff, directly in the case of Amazon, indirectly in the case of Google. Their virtual assistants are designed to assist their corporate parents (Amazon and Google) in making money. Of course they also want to assist the user in getting things done, so we'll keep using them. But the "objective functions" for these bots are designed to steer users towards purchases, not happiness or well-being.

Most marketing chatbots are conversational, to entertain users and mask their ulterior motives. They can also employ question answering skills, grounded in a knowledge base about the products they are sell. To mimic characters in a movie, show, or video game, chatbots will use text retrieval to find snippets of things to say from the script. And sometimes even generative models are trained directly on a collection of scripts. So marketing bots often employ all 4 of the techniques you'll learn about in this chapter.

COMMUNITY MANAGEMENT

Community management is a particularly important application of chatbots because it influences how society evolves. A good chatbot "shepherd" can steer a video game community away from chaos and help it grow into an inclusive, cooperative world where everyone has fun, not just the bullies and trolls. A bad chatbot, like the twitter bot Tay, can quickly create an environment of prejudice and ignorance.¹⁹¹

Footnote 191 Wikipedia article about the brief "life" of Microsoft's Tay chatbot,
[en.wikipedia.org/wiki/Tay_\(bot\)](https://en.wikipedia.org/wiki/Tay_(bot))

When chatbots go "off the rails", some people claim they are merely mirrors or magnifiers of society. And there are often "unintended consequences" of any complicated

system interacting with the real world. But because chatbots are active participants, imbued with motivations by developers like you, we shouldn't dismiss them as merely "mirrors of society." Chatbots seem to do more than merely reflect and amplify the best and the worst of us. They are an active force, partially under the influence of their developers and trainers, for either good or evil. Because supervisors and managers cannot perfectly enforce any policy that ensures chatbots "do no evil", it's up to you, the developer, to strive to build chatbots that are kind, sensitive, and pro-social. Asimov's "Three Laws of Robotics" aren't enough.¹⁹² Only you can influence the evolution of bots using smart software and cleverly constructed data sets.

Footnote 192 March 2014, George Dvorski, "Why Asimov's Three Laws of Robotics Can't Protect Us", Gizmodo, io9.gizmodo.com/why-asimovs-three-laws-of-robotics-cant-protect-us-1553665410

Some clever people at Arizona University are considering using their chatbot-building skills to save humanity, not from Evil Superintelligent AI, but from ourselves. Researchers to trying mimic the behavior of potential ISIS terrorist recruits to distract and misinform ISIS recruiters. This may one day mean that chatbots are saving human lives, simply by chatting it up with people that intend to bring harm to the world.¹⁹³ Chatbot trolls can be a good thing if they troll the right people or organizations.

Footnote 193 Oct 2015, Slate,
www.slate.com/articles/technology/future_tense/2015/10/using_chatbots_to_distract_isis_recruiters_on_social_media.html

CUSTOMER SERVICE

Customer service chatbots are often the only "person" available when you visit an online store. IBM's Watson, Amazon's Lex and other chatbot services are often used behind the scenes to power these customer assistants. They often combine both question answering skills (remember Watson's Jeopardy training?) with virtual assistance skills. However, unlike marketing bots, customer service chatbots must be well-grounded. And the knowledge base used to "ground" their answers to reality must be kept current. This enables customer service chatbots to answer questions about your order or product as well as initiate actions like placing or canceling an order.

In 2016 Facebook Messenger released an API for businesses to build customer service chatbots. And Google recently purchased API.ai to create their DialogFlow framework, which is often used to build customer service chatbots. Similarly, Amazon Lex is often used build customer service dialog engines for retailers and wholesalers of products sold on Amazon. Chatbots are quickly becoming a significant sales channel in industries from fashion (Botti Hilfiger) to fast food (TacoBot) and flowers.¹⁹⁴

Footnote 194 1-800-flowers: 1-800-Flowers.com, Tommy Hilfiger:
techcrunch.com/2016/09/09/botti-hilfiger/ TacoBot:
www.businessinsider.com/taco-bells-tacobot-orders-food-for-you-2016-4

THERAPY

Modern therapy chatbots, like Wysa and YourDost, have been built to help displaced tech workers adjust to their new lives.¹⁹⁵ Therapy chatbots must be entertaining like a conversational chatbot. They must be informative like a question answering chatbot. And they must be persuasive like a marketing chatbot. And if they are imbued with self-interest to augment their altruism, these chatbots may be "goal-seeking" and use their marketing and influence skill to get you to come back for additional sessions.

Footnote 195 Dec 2017, Bloomberg:
www.bloomberg.com/news/articles/2017-12-10/fired-indian-tech-workers-turn-to-chatbots-for-counseling

You might not think of Siri, Alexa, and Allo as therapists, but they can help you get through a rough day. Ask them about the meaning of life and you'll be sure to get a philosophical or humorous response. And if you are feeling down, ask them to tell you a joke or play some upbeat music. And beyond these parlor tricks, you can bet that developers of these sophisticated chatbots were guided by psychologists to help craft an experience intended to increase your happiness and sense of well-being.

As you might expect, these therapy bots employ a hybrid approach that combines all four of the basic approaches listed at the beginning of this chapter.

12.1.2 A Hybrid Approach

So what does this "hybrid" approach look like.

The four basic chatbot approaches can be combined in a variety of ways to produce useful chatbots. And many different applications use all 4 basic techniques. The main difference between hybrid chatbots is how the combine these four skills, and how much "weight" or "power" is given to each technique.

In this chapter we'll show you how to balance these approaches explicitly in code to help you build a chatbot that meets your needs. The hybrid approach we're using here will allow you to build features of all of these real world systems into your bot. And you will build an "objective function" that will take into account the goals of your chatbot when it is choosing between the four approaches, or merely choosing among all the possible responses generated by each of these four approaches.

So let's dive in to each of these four approaches, one at a time. For each one we'll build a chatbot that uses only the technique we're learning. But in the end we'll show you how to combine them all together.

12.2 1. Pattern Matching

The earliest chatbots used pattern matching to trigger responses. In addition to detecting statements that your bot can respond to, patterns can also be used to extract information from in the incoming text. You learned several ways to define patterns for information extraction in Chapter 11.

The information extracted from your users statements can be used to populate a database of knowledge about the user, or about the world in general. And it can be used even more directly to populate an immediate response to some statements. In Chapter 1 we showed a simple pattern-based chatbot that used a regular expression to detect greetings. We can also use regular expressions to extract the name of the person being greeted by the human user. This helps give the bot "context" for the conversation. This context can be used to populate a response.

ELIZA, developed in the late 70's, was surprisingly effective at this, convincing many users that "she" was capable of helping them with their psychological challenges. ELIZA was programed with a limited set of words to look for in user statements. The algorithm would rank any of those words that it saw in order to find a single word that seemed like the most "important" word in a user's statement. That would then trigger selection of a canned response template associated with that word. These response templates were carefully designed to emulate the empathy and open-mindedness of a therapist, using "reflexive" psychology. The key word that had triggered the response was often reused in the response to make it sound like ELIZA understood what the user was talking about. By replying in a user's own language, the bot helped build rapport and helped users believe that it was listening.

ELIZA taught us a lot about what it takes to interact with humans in natural language. Perhaps the most important revelation was that listening well, or at least appearing to listen well, is the key to chatbot success.

More recently, ALICE was developed based on a more general framework for defining these patterns and the response templates. Artificial Intelligence Markup Language (AIML) is the XML schema that was developed by the ALICE team to make it easier to

add additional patterns and responses. This has since become the defacto open standard for defining chatbot and virtual assistant configuration APIs for services such as Pandorabots.

AIML is an open standard and there are open source Python packages for parsing and "executing" AIML for your chatbot.¹⁹⁶. However, AIML limits the types of patterns and logical structures you can define. And it's XML. That means chatbot frameworks built in Python (like `Will` and `ChatterBot`) are usually better choices for defining your chatbot.

Footnote 196 `pip install aiml` github.com/creatorrr/pyAIML

Since you have a lot of your NLP tools in Python packages already, you can often build much more complex pattern matching chatbots just by building up the logic for your bot directly in Python and regular expressions or glob star patterns. At Aira we use a simple globstar language to define our patterns. We have a translator that converts this glob star pattern language (similar syntax to AIML patterns) into regular expressions that can be run on any platform with a regular expression matcher.

And we use `{ {handlebars} }` for our template specifications. The handlebars templating language has interpreters for Java and Python, so we can use it on a variety of mobile and server platforms. And handlebars expressions can include filters and conditionals which can be used to create complex chatbot behavior. And if you want something even more straightforward and pythonic for your chatbot templates, you can just use Python 3.6 f-strings. And if you're not yet using Python 3.6 you can use `str.format(template, locals())` to render your templates just like f-strings do.

12.2.1 A Pattern-Matching Chatbot with Artificial Intelligence Markup Language (AIML)

Here's how you might define our greeting chatbot from Chapter 1 in AIML (v2.0).¹⁹⁷

Footnote 197 "AI Chat Bot in Python with AIML" by NanoDano Aug 2015,
www.devdungeon.com/content/ai-chat-bot-python-aiml#what-is-aiml

EXAMPLE AIML 2.0

Listing 12.1 nlpia/book/examples/greeting.v2.aiml

```
<?xml version="1.0" encoding="UTF-8"?>
<aiml version="2.0">
<category>
    <pattern>HI</pattern>
    <template>Hi!</template>
</category>
```

```

<category>
  <pattern>[HELLO HI YO YOH YO'] [ROSA ROSE CHATTY CHATBOT BOT CHATTERBOT]</pattern>
  <template>Hi , How are you?</template>
</category>
<category>
  <pattern>[HELLO HI YO YOH YO' 'SUP SUP OK HEY]
    [HAL YOU U YALL Y'ALL YOUS YOUSE]</pattern>
  <template>Good one.</template>
</category>
</aiml>

```

In the example above we've used some of the new features of AIML 2.0 (by BOTlibre) to make the XML a little more compact and readable. The square brackets allow you to define alternative spellings of the same word in one line.

Unfortunately the Python interpreters for AIML (`PyAiml`, `aiml`, and `aiml_bot`) do not support version 2 of the AIML spec. The `python3` AIML interpreter works with the original AIML 1.0 specification is `aiml_bot`. In `aiml_bot` the parser is embedded within a `Bot()` class, designed to hold the "brain" in RAM that helps a chatbot respond quickly. The `aiml_bot` brain, or *kernel*, contains all the AIML patterns and templates in a single data structure, similar to a Python dictionary, mapping patterns to response templates.

AIML 1.0

AIML is a declarative language built on the XML standard. This limits the programming constructs and data structures you can use in your bot. But don't think of your AIML chatbot as being a complete system. We'll augment the AIML chatbot with all the other tools we learned about earlier.

One limitation of AIML is the kinds of patterns we can match and respond to. An AIML kernel (pattern matcher) only responds when input text matches a pattern "hard coded" by a developer. One nice thing is that AIML patterns can include wild cards, symbols that match any sequence of words. But for the words that you do include in your pattern, they must match precisely. No fuzzy matches, emoticons, internal punctuation characters, typos, or misspellings can be matched automatically. In AIML you have to manually define "synonyms" with an `</srai>`, one at a time. Think of all the stemming and lemmatization we did programmatically in Chapter 2. That would be tedious to implement in AIML. Though we'll show you how to implement synonym and typo matchers in AIML here, the hybrid chatbot we build at the end of the chapter will sidestep this tedium by processing all text coming into our chatbot.

Another fundamental limitation of the AIML `<pattern>`s that you need to be aware of is that they can only have a single wild card character. A more expressive pattern matching language like regular expressions can give you more power to create interesting chatbots.

¹⁹⁸ For now, with AIML, we'll only use patterns like "HELLO ROSA *" to match input text like "Hello Rosa, you wonderful chatbot!".

Footnote 198 It's hard to compete with modern languages like Python on consistent expressiveness: [wiki/Comparison_of_programming_languages#Expressiveness](#) and [Redmonk quantitative comparison of GitHub commits](#)

NOTE

The readability of a language is critical to your productivity as a developer. A good language can make a huge difference, whether you're building a chatbot or a web app.

We won't spend too much time helping you understand and write AIML. But we want you to be able to import and customize some of the open source AIML scripts. There are a lot of free, open source AIML scripts out there.¹⁹⁹ You can use AIML scripts, as-is, to give some basic functionality for your chatbot, with very little upfront work.

Footnote 199 CGoogle for "AIML 1.0 files" or "AIML brain dumps" and check out AIML resources like Chatterbots and Pandorabots: www.chatterbotcollection.com/category_contents.php?id_cat=20

In the next section we'll show you how to create and load an AIML file into your chatbot and generate responses with it.

PYTHON AIML INTERPRETER

Let's build up that complicated AIML script above one step at a time, and show you how to load it and "run" within a Python program. Here's a simple AIML file that can recognize 2 sequences of words, "Hello Rosa" and "Hello Troll". And our chatbot will respond to each differently, like we did in earlier chapters.

Listing 12.2 nlpia/nlpia/data/greeting_step1.aiml

```
<?xml version="1.0" encoding="UTF-8"?>
<aiml version="1.0.1">

<category><pattern>HELLO ROSA </pattern><template>Hi Human!</template></category>
<category><pattern>HELLO TROLL </pattern><template>Good one, human.</template></category>

</aiml>
```

NOTE

In AIML 1.0 all patterns must be specified in *ALL CAPS*.

We've set our bot up to respond differently to two different kinds of greetings, polite and impolite. Now let's use the `aiml_bot` package to interpret AIML 1.0 files in Python. If

you've installed the `nlpia` package, you can load these examples from there using the code below. If you want to experiment with the AIML files you typed up yourself, you'll need to adjust the path `learn=path` to point to your file.

Listing 12.3 nlpia/nlpia/book/examples/ch12.py

```
>>> import os
>>> from nlpia.constants import DATA_PATH
>>> import aiml_bot

>>> bot = aiml_bot.Bot(learn=os.path.join(DATA_PATH, 'greeting_step1.aiml'))
Loading /Users/hobs/src/nlpia/nlpia/data/greeting_step1.aiml...
done (0.00 seconds)

>>> bot.respond("Hello Rosa, ")
'Hi there!'
>>> bot.respond("hello **troll** !!!!")
'Good one, human.'
```

That looks good. The AIML specification cleverly ignores punctuation and capitalization when looking for pattern matches.

However, the AIML 1.0 specification only normalizes your patterns for punctuation and whitespace between words, not within words. It can't handle synonyms, spelling errors, hyphenated words, or compound words.

Listing 12.4 nlpia/nlpia/book/examples/ch12.py

```
>>> bot.respond("Helo Rosa")
WARNING: No match found for input: Helo Rosa
 ''
>>> bot.respond("Hello Ro-sa")
WARNING: No match found for input: Hello Ro-sa
 ''
```

We can fix most mismatches like this using the `<srai>` tag and a star ("`*`") symbol in our template to link multiple patterns back to the same response template. These can be thought of as "synonyms" for the word "Hello", even though they might just be misspellings or completely different words.

Listing 12.5 nlpia/nlpia/data/greeting_step2.aiml

```
<category><pattern>HELO *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>
<category><pattern>HI *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>
<category><pattern>HIYA *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>
```

```

<category><pattern>HYA *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>
</category><pattern>HY *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>
</category><pattern>HEY *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>
</category><pattern>WHATS UP *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>
</category><pattern>WHAT IS UP *
</pattern>
<template><srai>HELLO <star/></srai>
</template></category>

```

NOTE

If you are writing up your own AIML files, don't forget to include the `<aiml>` tags at the beginning and end. We've omitted them in example AIML code here to keep things brief.

Once we load that additional AIML, our bot can recognize a few different ways of saying and misspelling "Hello."

Listing 12.6 nlpia/nlpia/book/examples/ch12.py

```

>>> bot.learn(os.path.join(DATA_PATH, 'greeting_step2.aiml'))
>>> bot.respond("Hey Rosa")
'Hi there!'
>>> bot.respond("Hi Rosa")
'Hi there!'
>>> bot.respond("Helo Rosa")
'Hi there!'
>>> bot.respond("hello **troll** !!!") ①
'Good one, human.'

```

In AIML 2.0 you can specify alternative random response templates with square-bracketed lists. But in AIML 1.0 we use the `` tag to do that. The `` tag works only within a `<condition>` or `<random>` tag. We'll just use a `<random>` tag help our bot be a little more "creative" in how it responds to greetings.

Listing 12.7 nlpia/nlpia/data/greeting_step3.aiml

```

<category><pattern>HELLO ROSA </pattern><template>
<random>
<li>Hi Human!</li>
<li>Hello friend</li>
<li>Hi pal</li>
<li>Hi!</li>
<li>Hello!</li>
<li>Hello to you too!</li>
<li>Greetings Earthling ;)</li>
<li>Hey you :)</li>
<li>Hey you!</li>
</random></template>

```

```
</category>
<category><pattern>HELLO TROLL </pattern><template>
  <random>
    <li>Good one, Human.</li>
    <li>Good one.</li>
    <li>Nice one, Human.</li>
    <li>Nice one.</li>
    <li>Clever.</li>
    <li>:)</li>
  </random></template>
</category>
```

Now our chatbot does not sound nearly as mechanical (at least at the beginning of a conversation).

Listing 12.8 nlpia/nlpia/book/examples/ch12.py

```
>>> bot.learn(os.path.join(DATA_PATH, 'greeting_step3.aiml'))
>>> bot.respond("Hey Rosa")
'Hello friend'
>>> bot.respond("Hey Rosa")
'Hey you :)'
>>> bot.respond("Hey Rosa")
'Hi Human!'
```

NOTE

You likely did not get the same responses in the same order that I did when I ran this code. That's the point of the `<random>` tag. It will chose a random response from the list each time the pattern is matched. There's no way to set a random seed within `aiml_bot`, but this would help with testing (pull request anyone?).

You can define synonyms for your own alternative "spellings" of "Hi" and "Rosa" in separate `<category>` tags. You could define different groups of synonyms for your templates and separate lists of responses depending on the kind of greeting. For example you could define patterns for greetings like "SUP" and "WUSSUP BRO" and then respond in a similar dialect or similar level of "familiarity" and informality.

And AIML even has tags for capturing strings into named variables (like named groups in a regular expression). States in AIML are called `topics`. And AIML defines ways of defining conditionals using any of the variables you've defined in your AIML file. Try them out if you're having fun with AIML. It's a great exercise in understanding how grammars and pattern-matching chatbots work. But we're going to move on to more expressive languages like regular expressions and Python to build our chatbot. This will allow us to use more of the tools we learned in earlier chapters like stemmers and

lemmatizers to handle synonyms and misspellings (see Chapter 2). If you use AIML in your chatbot, and you have preprocessing stages like lemmatization or stemming you'll probably need to modify your AIML templates to catch these stems and lemmas.

If AIML seems a bit complicated for what it does, you're not alone. Amazon Lex uses a simplified version of AIML that can be exported to and imported from JSON. API.ai developed a dialog specification that was so intuitive that Google bought them out, integrated it with Google Cloud Services, and renamed it "DialogFlow." DialogFlow specifications can also be export to JSON and imported from JSON, but of course these files are not compatible with AIML or Amazon Lex format.

If you think all these incompatible APIs should be consolidated into a single open specification like AIML, you might want to contribute to the `aichat` project and the AIRS (AI Response Specification) language development. Aira and the #DoMore foundation are supporting AIRS to make it easier for our users to share their content (dialog for interactive fiction, inspiration, training courses, virtual tours) with each other. The `aichat` application is a reference implementation of the AIRS interpreter in python, with a web UX.

Here's what a typical AIRS specification looks like. It defines the four pieces of information that the chatbot needs to react to a user command in a single row of a flat table. This table can be exported/import to/from CSV or JSON or just a plain python list of lists:

```
>>> airas_spec = [
...     ["Hi {name}", "Hi {username} how are you?", "ROOT", "GREETING"],
...     ["What is your name?", "Hi {username} how are you?", "ROOT", "GREETING"],
```

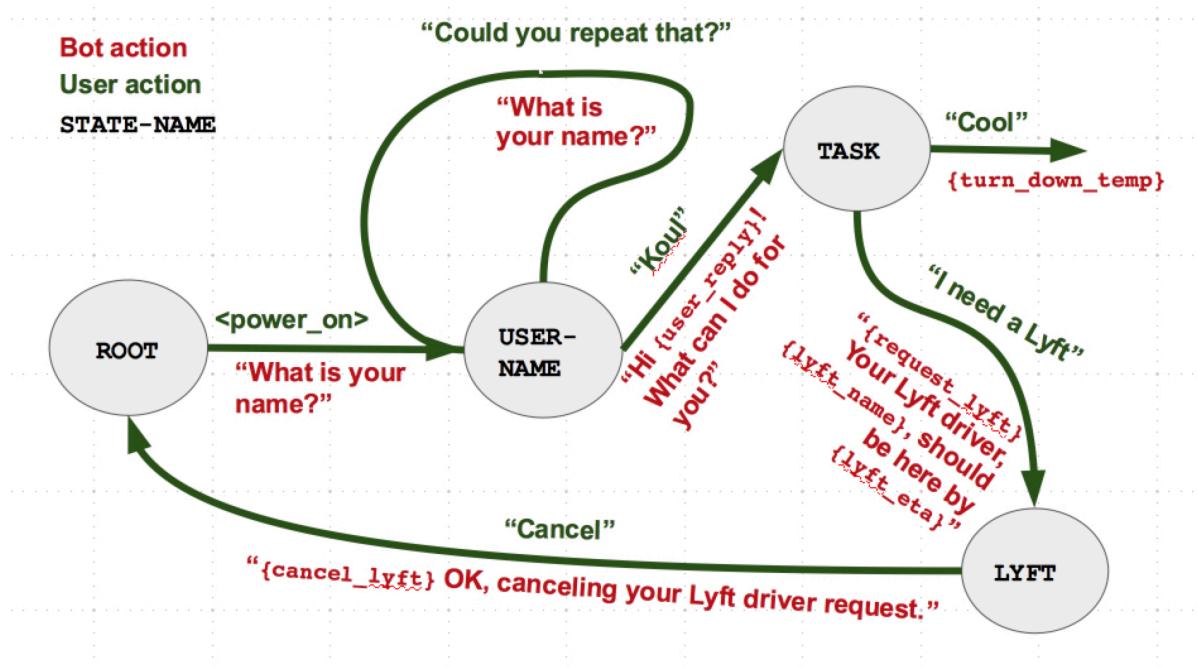


Figure 12.2 Managing state (context)

The first column defines the pattern and any parameters you want to extract from the user utterance or text message. The second column defines the response that you want the chatbot to say (or text). This is usually in the form of a template that can be populated with variables from the data context for the chatbot. And it can also contain special keywords to trigger bot actions other than just saying something.

The last two columns are used to maintain the state or context of the chatbot. Whenever the chatbot is triggered by a pattern match it can transition to a new state if it wants to have different behavior within that state to, say, follow up with additional questions or information. So the two columns at the end of a row just tell the chatbot what state it should be listening for these patterns in and which state it should transition to after it has accomplished the utterance or action specified in the template. These source and destination state names define a graph, like in the graphic, that governs the chatbot behavior.

Google's DialogFlow and Amazon's Lex have similar systems, but in most cases seem more complicated than they need to be. The open source project aichat is attempting to provide a more intuitive way to design, visualize, and test chatbots.

We'll come back to this pattern-matching approach when we build a hybrid chatbot later in this chapter. But for now let's move on to the second chatbot technique, "grounding."

12.3 2. *Grounding*

ALICE and other AIML chatbots rely entirely on pattern-matching. And the very first popular chatbot, ELIZA, used pattern-matching and templates as well, before AIML was even conceived. But these chatbot developers "hard-coded" the logic of the responses in patterns and templates. Hard coding doesn't "scale" well, not in the processing performance sense, but in the human effort sense. The sophistication of a chatbot built this way grows linearly with the human effort put into it. In fact, as the complexity of these chatbot grows we begin to see diminishing returns on our effort, as the interactions between all the "moving parts" grows and the chatbot behavior becomes harder and harder to predict and debug.

Data-driven programming is the modern approach to most complex programming challenges these days. How can we use data to program our chatbot? In the last chapter we learned how to create structured knowledge from natural language text (unstructured data) using information extraction. We can build up a network of relationships or facts just based on reading text, like Wikipedia, or even your own personal journal. In this section we'll learn how to incorporate this knowledge about the world (or your life) into our chatbots "bag of tricks." This network of logical relationships between things is a knowledge graph or knowledge base that can drive our chatbot's responses.

This knowledge graph can be processed with logical inference compose responses to questions about the world of knowledge contained in the knowledge base. Question Answering systems, like IBM's Jeopardy-winning Watson, are usually built this way, though they can also be built on search or retrieval technology. A knowledge graph is said to "ground" the chatbot to the real world.

This knowledge-based approach isn't limited to answering questions just about the world (or your life). Your knowledge base can also be populated in real time with facts about an ongoing dialog. This can keep your chatbot up to speed on who your conversation partner is, and what they are like. Each statement by dialog participants can be used to populate a "theory of mind" knowledge base about what each speaker believes about the world.

If you think about it, humans seem to participate in dialog in a more sophisticated way than merely regurgitating "canned responses" like the AIML chatbot we built above. Humans actually think about the logic of what our conversation partner said and attempt to infer something from what we remember about the logic of the real world. We may

have to make several inferences and assumptions to understand and respond to a single statement. So this addition of logic and grounding to our chatbot may make it be more human-like, or at least more logical.

This grounding approach to chatbots works well for question-answering chatbots, when the knowledge required to answer the question is within some broad knowledge base that we can obtain from an open source database. Some examples of open knowledge bases you can use to ground your chatbot include:

- [Wikidata \(includes Freebase\)](#)
- [Open Mind Common Sense \(ConceptNet\)](#)
- [Cyc](#)
- [YAGO](#)
- [DBpedia](#)

So all we need is a way to "query" the knowledge base to extract the facts we need to populate a response to a user's statement. And if the user is asking a factual question that our database might contain, we could translate their natural language question (like "Who are you?" or "What is the 50th state of the United States?") into a knowledge base query to directly retrieve the answer they're looking for. This is what Google search does using Freebase and other knowledge bases they combined together to create their knowledge graph.

We could use our word pattern matching skills from Chapter 11 to extract the critical parts of a question from the users statement, like the named entities or the relationship information sought by the question. We'd check for key question words like "who", "what", "when", "where", "why", and "is" at the beginning of a sentence to classify the type of question. This would help our chatbot determine the kind of knowledge (node or named entity type) to retrieve from our knowledge graph.

[Quepy](#) is a natural language query compiler that can produce knowledge base and database queries using these techniques. The SQL-equivalent for a knowledge graph of RDF triples is called [SPARQL](#)

12.4 3. Retrieval (Search)

Another more data-driven approach to "listening" to your user is to search for previous statements in your logs of previous conversations. This is analogous to a human listener trying to recall where they've heard that question or statement or word before. A bot can search not only its own conversation logs but also any transcript of conversations human-to-human conversations, bot-to-human conversations, or even bot-to-bot conversations. But, as usual, garbage in means garbage out. So your database of previous conversations should be curated, cleaned, to ensure your bot is searching (and mimicking) high quality dialog. You would like humans to enjoy the conversation with your bot.

A search-based chatbot should ensure that its dialog database contains conversations that were enjoyable or useful. And they should probably be on some "theme" that the bot personality is expected to converse in. Some examples of good sources of dialog for a search-based bot include movie dialog scripts, customer service logs on IRC channels (where the users' were satisfied with the outcome), and direct-message interactions between humans (when those humans are willing to share them with you). Don't do this on your own e-mail or SMS message logs without getting the written agreement of all humans involved in the conversations you want to use.

If you decide to incorporate bot dialog into your corpus, be careful. You only want statements in your database that have had at least one human appear to be satisfied with the interaction, if only by continuing the conversation. And bot-to-bot dialog should rarely be used, unless you have access to a *really* smart chatbot.

Your search-based chatbot can use a log of past conversations to find examples of statements similar to what the bot's conversation partner just said. To facilitate this search, the dialog corpus should be organized in statement-response pairs. If a response is responded to then it should appear twice in your database, once as the response and then again as the statement that is prompting a response. The response column in your database table can then be used as the basis for your bot's response to the statements in the "statements" (or prompt) column.

12.4.1 The Context Challenge

The simplest approach is to reuse the response verbatim, without any adjustment. This is often an OK approach if the statement is a good semantic (meaning) match for the statement your bot is responding to. But even if all the statements your users ever made could be found in your database, your bot would take on the personality of all the humans that uttered the responses in your dialog database. This can be a good thing, if you have a consistent set of responses by a variety of humans. But it can be a problem if the statement you are trying to reply to is dependent on the longer term context of a conversation or some real world situation that has changed since your dialog corpus was assembled.

For example, what if someone asked your chatbot "what time is it?" Your chatbot shouldn't reuse the reply of the human who replied to the best-matched statement in your database. That would only work if the time that the question was asked corresponded to the time of day that the matching dialog statement was recorded. This time information is called "context" or "state" and should be recorded and matched along with the natural language text of the statement. This is especially important when the semantics of the statement point to some evolving state that is recorded in your context, or the knowledge base of your chatbot.

Some other examples of how real-world knowledge or context should influence a chatbot's reply are the questions "who are you?" or "where are you from?" The context in this case is the identity and background of the person being addressed by the question. Fortunately this is context that we can generate and store quite easily in a knowledge base or database containing facts about the profile or back-story for our bot. We'd want to craft our chatbot profile to include information like a persona that roughly reflects the average or median profile of the humans who made the statements in our database. To compute this we can use the profiles of the users that made statements in our dialog database.

Our chatbot's personality profile information could be used to resolve "ties" in the search for matching statements in our database. And if we want to be super-sophisticated we can boost the rank of search results for replies from humans that are similar to our bot persona. For example, imagine we know the gender of the people whose statements and responses we recorded in our dialog database. We'd include the nominal gender of the chatbot as another "word" or dimension or database field we are searching for among the genders of the respondents in our database. If this respondent gender dimension matched our chatbot's gender and the prompting statement words or semantic vector were a close

match for the corresponding vector from our user's statement, that would be a great match at the top of our "search results." The best way to do this is to compute a scoring function each time a reply is retrieved and include in this score some profile match information.

Alternatively, we could solve this context challenge by building up background profile for our bot and storing it in a knowledge base manually (or logical ontology). We'd just make sure to only include replies in our chatbot's database that matched this profile.

No matter how we use this profile to give our chatbot a consistent personality, we'd need to deal with questions about that personality profile as "special cases". We need to use one of the other chatbot techniques rather than retrieval, if our database of statements and replies doesn't contain a lot of answers to questions like "who are you?", "where are you from?" and "what's your favorite color?" If we don't have a lot of "profile" statement-reply pairs, we would detect any questions about the bot and a knowledge base to "infer" an appropriate answer for that element of the statement. Alternatively we could use the grammar-based approach to populate a templated response, using information retrieved from a structured dataset for the chatbot profile.

To incorporate state or context into a retrieval-based chatbot you can do something very similar to what we did for the pattern-matching chatbot. If you think about it, listing a bunch of user statements is just another way of specifying a pattern. In fact, that's exactly the approach that Amazon Lex and Google Dialog Flow take. Rather than defining a rigid pattern to capture the user command, you can just provide the dialog engine with a few examples. So just as we associated a state with each pattern in our pattern-matching chatbot, we just need to tag our statement-response example pairs with a named state as well.

This can be difficult if your example state-response pairs are from an unstructured data source like the Ubuntu Dialog Corpus from an IRC chat channel. But with dialog training sets like Reddit, you can often find some small portions of the massive data set that can be automatically labeled based on their #channel and reply thread. You can use the tools of semantic search and pattern matching to cluster the initial comment that preceded a particular thread or discussion. And these clusters can then become your states. Detecting transitions from one topic or state to another can be difficult, however. And the states that you can produce this way are not nearly as precise and accurate as those you can generate by hand.

This approach to state (context) management can be a viable option, if your bot just

needs to be entertaining and conversational. But if you need your chatbot to have predictable and reliable behaviors, you probably want to stick to the pattern-matching approach or hand-craft your state transitions.

12.4.2 Example Retrieval-Based Chatbot

We're going to be following along with the ODSC 2017 tutorial on building a retrieval-based chatbot. If you want to view the video or the original notebook for this tutorial check out the github repository for it at github.com/totalgood/prosocial-chatbot/

Our chatbot is going to use the Ubuntu Dialog Corpus, a set of statements and replies recorded on the Ubuntu IRC channel where humans are helping each other solve technical problems. It contains more than seven million utterances and more than one million "dialog" sessions, each with multiple turns and many utterances.²⁰⁰ This large number of statement-response pairs makes it a popular common dataset used by researchers to check the "accuracy" of their retrieval-based chatbots.

Footnote 200 "The Ubuntu Dialogue Corpus: A Large Dataset for Research in Unstructured Multi-Turn Dialogue Systems" by Lowe et al., 2015 arxiv.org/abs/1506.08909

This is just the sort of statement-response pairings that we need to "train" a retrieval-based chatbot. But don't worry, we're not going to use all seven million utterances. We'll just use about 150 thousand turns and see if that's enough to give our chatbot the answers to some common Ubuntu questions. To get started, download this bite-sized Ubuntu corpus:

Listing 12.9 ch12_retrieval.py

```
>>> from nlpiam.data.loaders import get_data
>>> df = get_data('ubuntu_dialog')
Downloading ubuntu_dialog
requesting URL: https://www.dropbox.com/s/kvri79fbsryytc2/ubuntu_dialog.csv.gz?dl=1
remote size: 296098788
Downloading to /Users/hobs/src/nlpiam/nlpiam/bigdata/ubuntu_dialog.csv.gz
39421it [00:39, 998.09it/s]
```

You may get warning messages about the `/bigdata/` path not existing if you haven't used `nlpiam.data.loaders.get_data()` on a big dataset yet. But the downloader will create one for you when you run it for the first time.

NOTE

The scripts here will work if you have 8 GB of free RAM to work with. If you run out of memory, try reducing the dataset size—slice out a smaller number of rows in `df`. In the next chapter we'll use `gensim` to process data in batches "out of core" so that you can work with larger data sets.

Lets see what this corpus looks like: `.ch12_retrieval.py`

```
df.head(4)
          Context                Utterance
0 i think we could import the old comments via r... basically each xfree86
1 I'm not suggesting all - only the ones you mod... upload will NOT force u...
2 afternoon all __eou__ not entirely related to ... oh? oops. __eou__
3 interesting __eou__ grub-install worked with /... we'll have a BOF about
                                         this __eou__ so you're ...
                                         i fully endorse this suggestion
                                         </quimby> __eo...
```

Notice the "*eou*" tokens? This looks like it might be a pretty challenging dataset to work with. But it will give us practice with some common preprocessing challenges in NLP. Those mark the "end of utterance", the point at which the "speaker" hit [RETURN] or [Send] on their IRC client. If you print out some example "Context" fields you'll notice that there are also "*eot*" ("end of turn") markers to indicate when someone concluded their thought and was waiting for a reply.

But if you look inside a Context document (row in the table) you'll see there are multiple "*eot*" (turn) markers. This is to help more sophisticated chatbots test how they handle the context problem we talked about in the previous section. But we're going to ignore the extra "turns" in the corpus and focus only on the last one, the one that the "Utterance" was a reply to. First, let's create a function to split on those "*eot*" symbols and clean up those "*eou*" markers.

Listing 12.10 ch12_retrieval.py

```
>>> def split_turns(s, splitter=re.compile('__eot__')):
...     for utterance in splitter.split(s):
...         utterance = utterance.replace('__eou__', '\n')
...         utterance = utterance.replace('__eot__', '').strip()
...         if len(utterance):
...             yield utterance
```

Let's run that `split_turns` function on a few rows in the `DataFrame` to see if it makes sense. We'll retrieve only the last turn from both the Context and the Utterance and see if that will be enough to train a retrieval-based chatbot.

Listing 12.11 ch12_retrieval.py

```
for i, record in df.head(3).iterrows():
    statement = list(split_turns(record.Context))[-1]
    reply = list(split_turns(record.Utterance))[-1]
    print('Statement: {}'.format(statement))
    print()
    print('Reply: {}'.format(reply))
```

This should print out something like this:

```
Statement: I would prefer to avoid it at this stage. this is something that has gone into XSF svn, I as
Reply: basically each xfree86 upload will NOT force users to upgrade 100Mb of fonts for nothing
no something i did in my spare time.

Statement: ok, it sounds like you're agreeing with me, then
though rather than "the ones we modify", my idea is "the ones we need to merge"
Reply: oh? oops.

Statement: should g2 in ubuntu do the magic dont-focus-window tricks?
join the gang, get an x-series thinkpad
sj has hung on my box, again.
what is monday mornings discussion actually about?
Reply: we'll have a BOF about this
so you're coming tomorrow ?
```

Excellent! It looks like there are statements and replies that contain multiple statements (utterances). So our script is doing what we want and we can use it populate a statement-response mapping table.

Listing 12.12 ch12_retrieval.py

```
from tqdm import tqdm

def preprocess_ubuntu_corpus(df):
    """Split all strings in df.Context and df.Utterance on __eot__ (turn) markers """
    statements = []
    replies = []
    for i, record in tqdm(df.iterrows()):
        turns = list(split_turns(record.Context))
        statement = turns[-1] if len(turns) else '\n'
        statements.append(statement)
        turns = list(split_turns(record.Utterance))
        reply = turns[-1] if len(turns) else '\n'
        replies.append(reply)
    df['statement'] = statements
    df['reply'] = replies
    return df
```

1

- ① we need an if because some of the statements and replies contained only whitespace

Now we just need to retrieve the closest match to a user statement in the statement column, and reply with the corresponding reply from the reply column. Do you remember how we found similar natural language documents using word frequency vectors and TFIDF vectors in chapter 3?

Listing 12.13 ch12_retrieval.py

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
tfidf = TfidfVectorizer(min_df=8, max_df=.3, max_features=50000)
tfidf.fit(df.statement) ①
tfidf
```

- ① Notice only need to compute the TFIDFs statements (not replies) since those are the things we want to match

Let's create a `DataFrame` called `X` to hold all these TFIDF vectors for each of the 150 thousand statements:

Listing 12.14 ch12_retrieval.py

```
X = tfidf.transform(df.statement)
X = pd.DataFrame(X.todense(), columns=tfidf.get_feature_names())
```

One way to find the closest statement is to compute the cosine distance from the "query" statement to all the statements in our `X` matrix:

Listing 12.15 ch12_retrieval.py

```
x = tfidf.transform(['This is an example statement that we want to retrieve the best reply for.'])
cosine_similarities = x.dot(X.T)
reply = df.loc[cosine_similarities.argmax()]
```

That took a long time (more than a minute on my MacBook). And we didn't even compute a confidence value or get a list of possible replies that we might be able to combine with other metrics.

12.4.3 A Search-based Chatbot

What if the patterns we wanted to match were just the exact things people have said in previous conversations? That's what a search-based chatbot (or retrieval-based chatbot) does. A search-based chatbot indexes a dialog corpus so that it can easily retrieve previous statements similar to the one that the chatbot is being asked to reply to. It can then reply with one of the replies associated with that statement in the corpus that it has "memorized" and indexed for quick retrieval.

If you'd like to quickly get going with a search-based chatbot, ChatterBot by Gunther Cox, is a pretty good framework to cut your teeth on. It's easy to install (just `pip install ChatterBot`) and it comes with several conversation corpora that you can use to "train" your chatbot to carry on basic conversations. Chatterbot has corpora that allow it to talk about things like sports trivia, wax philosophical about AI sentience, or just

"shoot the breeze" with small talk. Chatterbot can be "trained" on any conversation sequence (dialog corpus). Don't think of this as machine learning training, but rather just indexing a set of documents for search.

By default ChatterBot will use both humans' statements as material for its own statements during training. If you want to be more precise with the personality of your chatbot, you'll need to create your own corpus in the ChatterBot '.yml' format. To ensure that only one personality is mimicked by your bot, make sure your corpus contains conversations of only 2 statements each, one prompt and one reply, the reply being from the personality you want to imitate. Incidentally, this format is very similar to the AIML format which has a pattern (the prompting statement in ChatterBot) and a template (the response in ChatterBot).

Of course, a search-based chatbot built this way is quite limited. It's never going to come up with new statements. And the more data you have, the harder it is to brute force the search of all the previous statements. So the smarter and more refined your bot is, the slower it will be. This architecture doesn't scale well. Nonetheless, we'll show you some advanced techniques for scaling any search or index-based chatbot with indexing tools like Locality Sensitive Hashes (`pip install lshash3`) and Approximate Near Neighbors (Oh Yea: `pip install annoy`).

Out of the box, ChatterBot uses sqlite3 as its database, which highlights these scaling issues as soon as you exceed about 10k statements in your corpus. If you try to train a sqlite-based ChatterBot on the Ubuntu Dialog Corpus you'll be waiting around for days... literally. It took me more than a day on a MacBook to ingest only 100k statement-response pairs. Nonetheless, this Chatterbot code is quite useful for downloading and processing this motherload of technical dialog about Ubuntu. Chatterbot takes care of all the book-keeping for you, downloading and decompressing the tarball automatically for you before walking the "leafy" file system tree to retrieve each conversation.

Here's how ChatterBot's "training" data (actually just a dialog corpus) is stored in a SQLite database:

Listing 12.16 ch12_chatterbot.sql

```
sqlite> .tables
conversation           response
conversation_association statement
tag
tag_association
sqlite> .width 5 25 10 5 40
sqlite> .mode columns
sqlite> .mode column
```

```
sqlite> .headers on
sqlite> SELECT * FROM response LIMIT 9;
id      text          created_at  occur    statement_text
----  -----
1      What is AI?    2017-11-26  2        Artificial Intelligence is the branch of
2      What is AI?    2017-11-26  2        AI is the field of science which concern
3      Are you sentient? 2017-11-26  2        Sort of.
4      Are you sentient? 2017-11-26  2        By the strictest dictionary definition o
5      Are you sentient? 2017-11-26  2        Even though I'm a construct I do have a
6      Are you sapient? 2017-11-26  2        In all probability, I am not. I'm not t
7      Are you sapient? 2017-11-26  2        Do you think I am?
8      Are you sapient? 2017-11-26  2        How would you feel about me if I told yo
9      Are you sapient? 2017-11-26  24       No.
```

Notice that some statements have many different replies associated with them. This allows the chatbot to choose from among the possible replies based on mood, or context, or just random chance. Chatterbot just chooses a response at random, but yours could be more sophisticated if you incorporated some other objective or loss function or heuristic to influence the choice. Also, notice that the created_at dates are all the same. That happens to be the date when I ran the Chatterbot "training" script which downloaded dialog corpora and loaded them into the database.

Search-based chatbots can also be improved by reducing the statement strings down to topic vectors of fixed dimensionality, using something like Word2Vec (summing all the word vectors for a short statement), or Doc2Vec (Chapter 6) or LSA (Chapter 4). Dimension reduction will help your bot generalize from the examples you train it with. This helps it respond appropriately when the meaning of the query statement (the most recent statement by your bot's conversation partner) is similar in meaning to one of your corpus statements even if it uses different words. This will work even if the spelling or characters in a statements are very different. Essentially, this semantic search-based chatbot is automating the programming of the templates we programmed in AIML earlier in this chapter. This dimension reduction also makes search-based chatbots smarter using machine learning (data-driven) than would be possible with a hard-coded approach to machine intelligence. Machine learning is preferable to hard-coding whenever you have a lot of labeled data, and not a lot of time (to code up intricate logic and patterns to trigger responses). For search-based chatbots the only "label" needed is an example response for each example statement in the dialog.

12.5 4. Generative Models

We promised a generative model in this chapter. But if you recall the Sequence-to-Sequence models you built in Chapter 10, you may recognize them as generative chatbots. They are machine learning translation algorithms that "translate" statements by your user into replies by your chatbot. So we won't go into generative models in any more detail here, but there are many more kinds of generative models. If you want to build a creative chatbot that says things that have never been said before generative models like these may be what you need:

- Sequence-to-Sequence: Sequence models trained to generate replies based on their input sequences ²⁰¹

Footnote 201 Explanation of Sequence to Sequence models and links to several papers
suriyadeepan.github.io/2016-06-28-easy-seq2seq/

- Restricted Boltzmann Machines (RBMs): Markov Chains trained to minimize an "Energy" function ²⁰²

Footnote 202 Hinton lecture at Coursera:
www.coursera.org/learn/neural-networks/lecture/TIqjI/restricted-boltzmann-machines-11-min

- Generative Adversarial Networks (GANs): Statistical models trained to fool a "judge" of good conversation ²⁰³

Footnote 203 Ian Goodfellow's GAN tutorial, NIPS 2016: arxiv.org/pdf/1701.00160.pdf and Lantau Yu's adaptation to text sequences: arxiv.org/pdf/1609.05473.pdf

We talked about Attention Networks (enhanced LSTMS) in Chapter 10, and we showed what an example chatbot can do. So we won't repeat that here. But now that you know about all the other chatbot approaches, can you think how we might combine generative approaches with the others so that get the best of each of the 4 approaches.

12.5.1 Pros and Cons of Each Approach

Here are the advantages and disadvantages of each:

| Approach | Advantages | Disadvantages |
|-------------------|--|--|
| Grammar | Easy to get started Training easy to reuse Modular Easily controlled/restrained | Limited "domain" Capability limited by human effort Difficult to debug Rigid, brittle rules |
| Grounding | Answers logical questions well Easily controlled/restrained | Sounds artificial, mechanical Difficulty with ambiguity Difficulty with common sense Limited by structured data Requires large scale information extraction Requires human curation |
| Retrieval | Simple Easy to "train" Can mimic human dialog | Difficult to scale Incoherent personality Ignorant of context Can't answer factual questions |
| Generative | New, creative ways of talking Less human effort Domain limited only by data Context aware | Difficult to "steer" Difficult to train Requires more data (dialog) Requires more processing to train |

Figure 12.3 Advantages and Disadvantages of Four Chatbot Approaches

12.6 Four Wheel Drive

As we promised at the beginning of this chapter, we'll now show you how to combine all four of these approaches to get traction with your users. To do this we need a modern chatbot framework that is easy to extend and modify and can efficiently run each of these algorithm types in parallel.²⁰⁴ We're going to add a response generator for each of the four approaches using the Python examples above. And then we're going to add the logic to decide what to actually say by choosing one of the four (or many) responses. We're going to have our chatbot think before it speaks, saying things several different ways to itself first, ranking or merging some of these alternatives to produce a response. And maybe we can even try to be prosocial with our replies by checking the sentiment of our replies before "hitting the send button."

Footnote 204 We're building an open source chatbot framework at Aira called `aichat` to help our users and their friends contribute "content" to our library of dialog to help and entertain people with blindness and low vision: github.com/aira/aichat

12.6.1 The will to Succeed

Will is a modern programmer-friendly chatbot framework by Steven Skoczen that can participate in your HipChat and Slack channels as well as others.²⁰⁵ Python developers will enjoy the modular architecture. However it's pretty heavy weight in terms of requirements and installation. Fortunately it comes with a Dockerized container you can use to spin up your own chatbot server.

Footnote 205 GitHub Repository github.com/skoczen/will

Will uses regular expressions to make matches. Python itself can be used for any logical conditions you need to evaluate. And the jinja2 library is used for templating. Each one of these portions of the pipeline add versatility, flexibility to the kinds of behaviors you can build into your chatbot. As a result Will is much more flexible than AIML-based frameworks. However, Will still suffers from the same limitations that hold back all pattern-based chatbots (including AIML)--it can't learn from data, it must be "taught" by the developer writing code for each and every branch in the logic tree.

INSTALLING WILL

There are a couple small gaps in the installation process for Will. By the time this goes to print I hope to have them fixed so you can likely just read the high quality docs.²⁰⁶

Footnote 206 Will Documentation: skoczen.github.io/will/

On Mac OSX you can install and launch a redis server () brew install redis

HELLO WILL

Heres's what a conversation with an untrained Will looks like, if you ignore the tracebacks about port 80 permissions, or you can figure out how to avoid these errors:

```
You: Hey
Will: hello!
You: What's up?
Will: I heard you, but I'm not sure what to do.
You: How are you?
Will: Doing alright. How are you?
You: What are you?
Will: Hmm. I'm not sure what to say.
You: Are you a bot?
Will: I didn't understand that.
```

As you can see, out of the box, Will is polite, but doesn't understand much. We can easily change out the Will name for Rosa. And we can use your natural language processing skills to beef up some of his patterns and expand his literary power. Here's how you can augment the "Hello Will" patterns to get close to the functionality of the AIML implementation.

12.7 Design Process

To create a useful app, product managers and developers compose user stories. A user story describes a sequence of actions performed by a user in interacting with your app and how your app should respond. These can be imagined based on similar experiences in the real world with similar products, or they can be translated from user feature requests or feedback. Software features are tied to a user story to improve the likelihood that the development effort is focused on something that will add usefulness to your product.

User stories for a chatbot can often be composed as statements (text messages) that a user might communicate to the bot. Those user statements are then paired with the appropriate response or action by the chatbot or virtual assistant. For a retrieval-based chatbot, this table of user stories is all that's required to "train" a chatbot for these particular responses and stories. It's up to you, the developer, to identify stories that can be generalized so that your design team doesn't have to specify everything your bot must understand and all the different things it can say. Can you tell which of the 4 chatbot techniques we've discussed in this chapter would be appropriate for each of these questions?

- "Hello!" "Hello!"
- "Hi" "Hi!"
- "How are you?" "I'm fine. How are you?"
- "How are you?" "I'm a stateless bot, so I don't have an emotional state."
- "Who won the 2016 World Series?" "Chicago Cubs"
- "Who won the 2016 World Series?" "The Chicago Cubs beat the Cleveland Indians 4 to 3"
- "What time is it" "2:55 pm"
- "When is my next appointment?" "At 3 pm you have a meeting with the subject 'Springboard call'"
- "When is my next appointment?" "At 3 pm you need to help Les with her Data Science course on Springboard"
- "Who is the President?" "Sauli Niinistö"
- "Who is the President?" "Barak Obama"

Several valid responses may be possible for any given statement, even for the exact same user and context. And it's also common for multiple different prompting statements to elicit the same exact chatbot response (or set of possible responses). The Many-to-Many mapping between statements and responses works both ways, just as it does for human dialog. So the number of possible combinations of valid *statement response* mappings can be enormous, seemingly infinite (but technically finite).

And we must also expand the combinations of statement-response pairs in our user

stories using named variables for context elements that change often:

- Date
- Time
- Location: country, state, county, city or latitude and longitude
- Locale: US or Finland formatting for date, time, currency, and numbers
- Interface type: mobile or laptop
- Interface "modality": voice or text
- Previous interactions: whether user asked for detail about baseball stats recently
- Streaming audio, video, and sensor data from a mobile device (Aira.io)

IBM Watson and Amazon Lex chatbot APIs rely on knowledge bases that are not easy to evolve quickly and keep up to speed with these evolving context variables. The "write rate" for these databases of knowledge are too slow to handle many of these evolving "facts" about the world that the chatbot and the user are interacting with.

The list of possible user stories for even the simplest of chatbots is technically finite, but is quite large for even the simplest real-world chatbot. One way to deal with this explosion of combinations is to combine many user interactions into a single pattern or template. For the *statement* side of the mapping, this template approach is equivalent to creating a regular expression (or finite state machine) to represent some group of statements that should cause a particular pattern response. For the *response* side of the mapping, this approach is equivalent to a `Jinja2` or `Django` or `Python` f-string template

Thinking back to our first chatbot in Chapter 1, we can represent *statement response* mappings that map regular expressions for the statement to a Python f-string for the response:

```
{
    r"[Hh]ello|[Hh]i[!]*": f"Hello {userNickname}, would you like to play a game?",
    r"[Hh]ow[\s]*(s|are|re)?[\s]*[Yy]ou([\s]*doin['g'])?": 
        f"I'm {botMood}, how are you?",
}
```

But this doesn't allow for complex logic. And it requires hand coding rather than machine learning. So each mapping doesn't capture a very broad range of statements and responses. We'd like a machine learning model to be able to handle a wide range of sports questions, or help a user manage their calendar.

Here are some example chatbot user stories that don't lend themselves well to the template approach:

- "Where is my Home" "Your home is 5 minutes away by foot, would you like

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

directions?"

- "Where am I" "You are in SW Portland near Goose Hollow Inn" or you are at "2004 SW Jefferson Street"
- "Who is the President?" "Sauli Niinistö" or "Barak Obama" or "What country or company ..."
- "Who won the 2016 World Series?" "Chicago Cubs" or "The Chicago Cubs beat the Cleveland Indians 4 to 3"
- "What time is it" "2:55 pm" or "2:55 pm, time for your meeting with Joe" or ...
- "Where is my Home" "Your home is 5 minutes away by foot, would you like directions?"
- "Where am I" "You are in SW Portland near Goose Hollow Inn" or you are at "2004 SW Jefferson Street"
- "Who is the President?" "Sauli Niinistö" or "Barak Obama" or "What country or company ..."

Even though these cannot be easily translated directly into code, they do translate directly into a test set. These examples can be used to evaluate a new chatbot feature.²⁰⁷

Footnote 207 2017-01 Andrew Ng lecture to Stanford Business School students
youtu.be/21EiKfQYZXc?t=48m6s

12.8 Trickery

12.8.1 Ask Questions With Predictable Answers

When asked a question that you do not know the answer to, the chatbot can respond with a "clarifying" question. And if this clarifying question is well within the knowledge domain or personality profile of the chatbot, it's possible to predict the form of the answer that a human would make. Then the chatbot can use the user's response to regain control of the conversation and steer it back towards topics that it knows something about. To avoid frustration, try to make the clarifying question humorous, or positive and flattering, or in some way pleasing to the user.

human: Where were you born? bot: I don't know, but how about those Mets? (for a sports bot) bot: I don't know, but are you close to your mother? (for a therapist bot) bot: I don't know, but how do you shut down your Ubuntu PC at night? (for an Ubuntu help bot)

You can use semantic search to find question-answer pairs, jokes, or interesting trivia in the chatbot's knowledge base that are at least tangentially related to what the user is asking about.

12.8.2 Be Entertaining/Likable

If you have to respond with a nonsequitor (because of a timeout due to your generative process taking to long to "converge" to a high quality message), makes sure it's sensational, distracting, flattering, or humorous. And make sure your responses are all pseudo-random in a way that a human would consider random (e.g. don't repeat yourself very often). And use varying sentence structure/form/style over time.

12.8.3 When All Else Fails, Search

If your bot can't think of anything to say, try acting like a search engine or search bar. Search for webpages or internal database records that might be relevant to any question you might receive. And ask whether the page titles or DB records might help the user before spitting them all out. Stack Overflow, Wikipedia, and Wolfram Alpha are good resources to have indexed and at the ready for many bots (because Google does that and users expect it).

12.8.4 Being Popular

If you have a few jokes or links to resources or responses that are favorably received by your audience, in general, then respond with those rather than the "best match" for the question asked, especially if the match is very low. And these jokes or resources may help bring your human back into a path of conversation that you are familiar with and have a lot of training set data for.

12.8.5 Be a Connector/Networker

Introduce the human to other humans on the chat forum or people who've written about things the user has written about. Or point the user to a blog post, meme, chat channel, or other website that is relevant to something they might be interested in. A good bot will have a handy list of popular links to hand out when the conversation starts to get repetitive.

bot: You might like to meet @SuzyQ, she's asked that question a lot lately. She might be able to help you figure it out.

12.8.6 Getting Emotional

Google's Inbox e-mail responder is very similar to the conversational chatbot problem we want to solve. The auto-responder must suggest a reply to the e-mails you receive based on their semantic content. But a long chain of replies is less likely for an e-mail exchange. And the "prompting" text is generally much longer for an email auto-responder than it is for a conversational chatbot. Nonetheless, the problems both involve generating text replies to incoming text prompts. So many of the techniques for one may be applicable to the other.

Even though Google had access to billions of e-mails and reply e-mails, to help them identify common response patterns. A semantic search approach is likely to produce relatively generic, bland replies if you are trying to maximize "correctness" for the average email user. The average reply is not likely to have much personality or emotion. So Google tapped an unlikely corpus to add a bit of emotion to their suggested replies... romance novels.

It turns out that romance novels tend to follow predictable plots and have sappy dialog that can be easily dissected and imitated. And it contains a lot of emotion. Now I'm not sure how Google gleaned phrases like "That's awesome! Count me in!" or "How cool! I'll be there." from romance novels, but they claim that's the source of the emotional exclamations that they suggest.

12.9 In the Real World

The hybrid chatbot we've assembled here has the flexibility to be used for the most common real world applications. In fact you've probably interacted with a chatbot like ours sometime this week. Perhaps you chatted with chatbots as

- customer service assistants
- sales assistants
- marketing (spam) bots
- toy or companion bots
- video game AI
- mobile assistants
- home automation assistants
- visual interpreters
- therapist bots

And you're likely to run across chatbots like the one we built in this chapter more and more. User interfaces are migrating away from designs constrained by the rigid logic and

data structures of machines. More and more machines are being taught how to interact with us in natural, fluid conversation. The "voice first" design pattern is becoming more popular as chatbots become more useful and less frustrating. And these dialog system approaches promise a richer and more complex user experience than clicking buttons and swiping left. And with chatbots interacting with us "behind the curtains" they are becoming more deeply embedded in the collective consciousness.

12.10 Summary

So now you've learned all about building chatbots "for fun and for profit." And you've learned how to combine generative dialog models, semantic search, pattern matching, and information extraction (knowledge bases) to produce a chatbot that sounds surprisingly intelligent.

- Combining multiple proven approaches you can build an intelligent dialog engine
- Breaking "ties" between the 4 main chatbot approaches is one key to intelligence
- Good chatbots may help save the world
- We can teach machines a lifetime of knowledge without spending a lifetime programming them

You've now mastered all the key NLP components of an intelligent chatbot. You're only remaining challenge is to give it a personality of your own design. And then you'll probably want to "scale it up" so it can continue to learn, long after you've exhausted the RAM, hard drive, and CPU in your laptop. And we'll show you how to do that in Chapter 13.

13

Scaling Up (Optimization, Parallelization and Batch Processing)

In this chapter

- Scaling up an NLP pipeline
- Speeding up search with indexing
- Batch processing to reduce your memory footprint
- Parallelization to speed up NLP
- Running NLP model training on a GPU

In Chapter 12 we learned how to use all the tools in our NLP toolbox to build a build an NLP pipeline capable of carrying on a dialog. We demonstrated crude examples of this chatbot dialog capability on small data sets. The "humanness", or IQ, of our dialog system seems to be limited by the data we train it with. Most of the NLP approaches you've learned about give better and better results, if we can scale them to use larger data sets.

However, you may have noticed that your computer bogs down, even crashes, if you run some of the examples we gave you on large data sets. There are some datasets in `nlpia.data.loaders.get_data()` that will exceed the memory (RAM) in most PCs or laptops.

Besides RAM, there's another bottleneck in our natural language processing pipelines, the processor. Even if you had unlimited RAM, larger corpora would take days to process with some of the more complex algorithms you have learned.

So we need to come up with algorithms that minimize the resources they require:

- Volatile storage (RAM)
- Processing (CPU cycles)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

13.1 Too Much of a Good Thing (Data)

As you add more data, more knowledge, to your pipeline, the machine learning models take more and more RAM, storage, and CPU cycles to train. Even worse, some of the techniques relied on an $O(N^{**2})$ computation of distance or similarity between vector representations of statements or documents. For these algorithms, things get slower faster as you add data. Each additional sentence in the corpora takes more bytes of RAM and more CPU cycles to process than the previous one. This is impractical for even moderately sized corpora.

There are two broad approaches to avoiding this so that we can scale up our NLP pipeline to larger data sets.

1. Increasing scalability: improving or optimizing the algorithms
2. Horizontal scaling: parallelizing the algorithms to run multiple computations simultaneously

In this chapter you will learn techniques for both.

Getting smarter about algorithms is almost always the best way to speed up a processing pipeline so we'll talk about that first. We'll leave parallelization to the second half of this chapter, to help you run you sleek, optimized algorithms even faster.

13.2 Optimizing NLP Algorithms

Some of the algorithms we've looked at in previous chapters have expensive complexities, often quadratic $O(N^{**2})$ or higher:

- compiling a thesaurus of synonyms from word2vec vector similarity
- clustering web pages based on their topic vectors
- automatically clustering journal articles or other documents based on topic vectors
- clustering questions in a Q&A corpus to automatically compose a FAQ

All of these NLP challenges fall under the category of indexed search or "K Nearest Neighbors" (KNN) vector search. We'll spend the next few sections talking about the first solution listed above for the scaling challenge: algorithm optimization. We're going to show you one particular algorithm optimization, called "indexing", that will help solve most vector search (KNN) problems. In the second half of the chapter we'll show you how to hyper-parallelize your natural language processing by using thousands of CPU cores in a Graphical Processing Unit (GPU).

13.2.1 Indexing

You probably use natural language indexes every day. Natural language text indexes (also called "reverse indexes") are what you use when you turn to the back of a textbook to find the page for a topic you are interested in. The pages are the documents and the words are the lexicon of your Bag of Words vectors (BOW) for each document. And you use a text index every time you enter a search string in a web search tool. To scale up your NLP application you need to do that for semantic vectors like LSA document-topic vectors or word2vec word vectors.

Previous chapters have mentioned conventional "reverse indexes" used for searching documents to find a set of words or tokens based on the words in a query. But we've not yet talked about K-Nearest-Neighbor (KNN) search for similar text. For KNN search we want to find strings that are similar even if they don't contain the exact same words. Levenshtein distance is one of the distance metrics used by packages like `fuzzywuzzy` and ChatterBot to find similar strings.

Databases implement a variety of text indexes that allow us to find documents or strings quickly. SQL queries allow us to search for text that matches patterns like `SELECT book_title from manning_book WHERE book_title LIKE 'Natural Language%in Action'`. This query would find all the "in Action" Manning titles that start with "Natural Language". And there are trigram (`trgm`) indexes for a lot of databases that help you find similar text quickly (in constant time) without even specifying a pattern, just specifying a text query that is similar to what you're looking for.

These database techniques for indexing text work great for text documents or strings of any sort. But they don't work well on semantic vectors like word2vec vectors or dense document-topic vectors. Conventional database indexes rely on the fact that the objects (documents) they are indexing are either discrete, sparse, or low dimensional:

- strings (sequences of characters) are discrete: there are a limited number of characters
- TFIDF vectors are sparse: the most terms have a frequency of 0 in any given document
- BOW vectors are discrete and sparse: terms are discrete and most words have zero frequency in a document

This is why it's possible to perform a web search, document search, or geographic search in milliseconds. And it's been working efficiently ($O(1)$) for many decades.

What makes continuous vectors like document-topic LSA vectors (Chapter 4) or word2vec vectors (Chapter 6) so hard to index? After all, Geographic Information

System (GIS) vectors are, typically latitude, longitude, and altitude. And we can do a GIS search on google maps in milliseconds. Fortunately GIS vectors only have 3 continuous values so indexes can be built based on bounding boxes that gather together GIS objects in discrete groups.

Several different index data structures can deal with this problem.

- K-d Tree: Elastic search will implement this for up to 8D in upcoming releases)
- Rtree: PostgreSQL implements this in versions ≥ 9.0 for up to 200D)
- Minhash or Locality Sensitive Hashes: `pip install lshash3`.

These work up to a point. That point is at about 12 dimensions. If you play around with optimizing database indexes or Locality Sensitive Hashes yourself, you will find that it gets harder and harder to maintain that constant-time lookup speed. At about 12 dimensions it becomes impossible.

So what are we to do with our 300D `word2vec` vectors or 100+ dimension semantic vectors from LSA? Approximation to the rescue. Approximate Nearest Neighbor search algorithms don't try to give you the exact set of document vectors that are most similar to your query vector. Instead they just try to find some reasonably good matches. And they are usually pretty darn good, rarely missing any closer matches in the top 10 or so search results.

However, things are quite different if you're using the magic of SVD or "Embedding" to reduce your token dimensions (your vocabulary size, typically in the millions) to say 200 or 300 topic dimensions. Three things are different. One change is good, you have fewer dimensions to search (think columns in a database table). This is a huge advantage. We've cured ourselves of the curse of dimensionality.

13.2.2 Advanced Indexing

Semantic vectors check all the boxes for difficult objects. They are difficult because they are:

- High dimensional
- Real-valued
- Dense

We've replaced the curse of dimensionality with two new difficulties. Our vectors are now dense (no zeros that we can ignore) and continuous (real valued).

In our dense semantic vectors every dimension has a meaningful value. We can no longer

skip or ignore all the zeros that filled the table before. There are no zeros in our vectors anymore. Every topic has some weight associated with it for every document. This isn't an insurmountable problem. The reduced dimensionality more than makes up for the density problem.

But the values in these dense vector are real numbers. However, there's a bigger problem. Topic weight values in a semantic vector can be positive or negative and aren't limited to discrete characters or integer counts. The weights associated with each topic are now continuous real values ('float's). Nondiscrete values, like floats are impossible to index. They are no longer merely present or absent. They can't be vectorized with "one hot encoding" of input as a feature into a neural net. And we certainly can't create an entry in an index table that refers to all the documents where that feature or topic was either present or absent. Topics are now everywhere, in all the documents, to varying degrees.

We can solve the natural language search problems at the beginning of the chapter if we can find an efficient search or KNN algorithm. One of the ways to optimize the algorithm for such problems is to sacrifice certainty and accuracy in exchange for a huge speedup. This is called Approximate Nearest Neighbors (ANN) search. For example, Google's search doesn't try to find you a perfect match for the semantic vector in your search. Instead it attempts to provide you with the closest 10 or so approximate matches.

Fortunately a lot of companies like have open sourced much of their research software for making ANN more scalable. These research groups are competing with each other to give us the easiest, fastest ANN search software. Here are some of the Python packages from this competition that have been tested with standard benchmarks for NLP problems at the India Technical University (ITU):²⁰⁸

Footnote 208 ITU comparison of ANN Benchmarks: www.itu.dk/people/pagh/SSS/ann-benchmarks/

- Spotify's annoy-hamming²⁰⁹

Footnote 209 github.com/spotify/annoy

- BallTree (using nmslib)²¹⁰

Footnote 210 github.com/searchivarius/nmslib

- Brute Force using Basic Linear Algebra Subprograms library (BLAS)²¹¹

Footnote 211 scikit-learn.org/stable/modules/neighbors.html#brute-force

- Brute Force using Non-Metric Space Library (NMSlib)²¹²

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

Footnote 212 github.com/searchivarius/NMSLIB

- Dimension reductiOn and LookuPs on a Hypercube for effIcient Near Neighbor (DolphinnPy)²¹³

Footnote 213 github.com/ipsarros/DolphinnPy

- Random Projection Tree Forest (`rpyforest`)²¹⁴

Footnote 214 github.com/lyst/rpyforest

- Locality Sensitive Hasing (`datasketch`)²¹⁵

Footnote 215 github.com/ekzhu/datasketch

- Multi Index Hashing (MIH)²¹⁶

Footnote 216 github.com/norouzi/mih

- Fast Lookup of cosine and Other Nearest Neighbors (FALCONN)²¹⁷

Footnote 217 pypi.python.org/pypi/FALCONN

- Fast Lookup of Approximate Nearest Neighbors (FLANN)²¹⁸

Footnote 218 www.cs.ubc.ca/research/flann/

- Hierarchical Navigable Small World (HNSW) (in `nmslib`)²¹⁹

Footnote 219

github.com/searchivarius/nmslib/blob/master/similarity_search/include/factory/method/hnsw.h

- K-Dimensional Trees (KD-Tree)²²⁰

Footnote 220 pypi.python.org/pypi/NearPy

- `nearpy`[pypi.python.org/pypi/NearPy]

One of the most straightforward of these indexing approaches is implemented in a package called `Annoy` by Spotify.

13.2.3 Advanced Indexing with Annoy

The recent update to word2vec (KeyedVectors) in gensim added an advanced indexing approach. You can now retrieve approximate nearest neighbors for any vector in milliseconds, out of the box. But, as we talked about this in the beginning of the chapter we need to use indexing for any kind of high-dimension dense continuous vector set, not just word2vec vectors. So lets use annoy to index the word2vec vectors and compare our results to gensim.

First we need to load the word2vec vectors like we did in Chapter 6.

```
>>> import os
>>> from nlpias.data.loaders import BIGDATA_PATH
>>> from gensim.models.word2vec import KeyedVectors
>>> path = os.path.join(BIGDATA_PATH, 'GoogleNews-vectors-negative300.bin.gz')
>>> wv = KeyedVectors.load_word2vec_format(path, binary=True)
>>> len(wv.vocab)
3000000
```

Now let's set up an empty annoy index with the right number of dimensions for our vectors:

```
from annoy import AnnoyIndex
num_dimensions = len(wv[wv.vocab[0]]) ①
index = annoy.AnnoyIndex(num_dimensions)
```

Now we can add our word2vec vectors to our annoy index one at a time. You can think of this process as reading through the pages of a book one at a time, and putting the page numbers where you found each word in the reverse index table at the back of the book. Obviously an Approximate Nearest Neighbor search is much more complicated, but annoy makes it easy for us.

```
>>> for i, word in enumerate(wv.index2word):
...     if not i % 100000:
...         print('{}: {}'.format(i, word))
...     index.add_item(i, wv[word])
0: </s>
100000: distinctiveness
...
2600000: cedar_juniper
2700000: Wendy_Liberatore
2800000: Management_GDCM
2900000: BOARDED_UP
```

We have to do one last thing to do. We have to read through the entire index and try to cluster our vectors into bite-size chunks that can be indexed in a tree structure:

```
>>> num_vectors = len(wv.vocab)
>>> num_trees = int(np.log(num_vectors).round(0))
>>> index.build(num_trees)                                1
>>> index.save('Word2vec_index.ann')                      2
True
```

- ➊ 10 indexing trees
- ➋ saves the index to a local file and frees up RAM

We built 15 trees (approximately the natural log of 3 million) since we have 3 million vectors to search through. If you have more vectors or want your index to be faster and more accurate you can increase this. Just be careful not to make it too big or you'll have to wait a while for the indexing process to complete.

Now we can try to look up a word from our vocabulary in the index.

```
>>> index.get_nns_by_item(w2id['Harry_Potter'], 10)
>>> >>> index.get_nns_by_item(w2id['Harry_Potter'], 10)
[9494, 32643, 39034, 114813, 172698, 145465, 113008, 116741, 113955, 350346]
>>> [wv.vocab[i] for i in _]
>>> [wv.index2word[i] for i in _]
['Harry_Potter',
 'Narnia',
 'Sherlock_Holmes',
 'Lemony_Snicket',
 'Spiderwick_Chronicles',
 'Unfortunate_Events',
 'Prince_Caspian',
 'Eragon',
 'Sorcerer_Apprentice',
 'RL_Stine']
```

10-Nearest-Neighbors listed by `annoy` are mostly books from the same general genre as *Harry Potter* but nothing at all that could be considered synonymous with the book title. It seems like an `annoy` index misses a lot of closer neighbors and provides results from the general vicinity of a search term rather than the closest 10.

What would happen if we did that with gensim's built-in `KeyedVector` index to retrieve the correct closest 10 neighbors:

```
>>> [word for word, similarity in wv.most_similar('Harry_Potter', topn=10)]
['JK_Rowling_Harry_Potter',
 'JK_Rowling',
 'boy_wizard',
 'Deathly_Hallows',
 'Half_Blood_Prince',
 'Rowling',
 'Actor_Rupert_Grint',
```

```
'HARRY_Potter',
'wizard_Harry_Potter',
'HARRY_POTTER']
```

Now that looks like a more relevant top 10 "synonym" list. The correct answer lists the correct author, alternative title spellings, titles of other books in the series, and even an actor that played in the movie. But the results from annoy may be useful in some situations, when you're more interested in the "genre" or general sense of a word rather than precise synonyms. That's pretty cool.

But the annoy indexing approximation really took some shortcuts. Let's rebuild it and try to make its results match gensim's more closely.

```
>>> num_dimensions = len(wv[wv.index2word[0]])
>>> index_cos = annoy.AnnoyIndex(f=num_dimensions, metric=cos) ①
>>> for i, word in enumerate(wv.index2word):
...     if not i % 100000:
...         print('{}: {}'.format(i, word))
...     index_cos.add_item(i, wv[word])
0: </s>
...
2900000: BOARDED_UP
```

① We'll use the angular (cosine) distance metric to compute our Trees and hashes

Now let's build twice the amount of trees:

```
>>> num_vectors = len(wv.vocab)
>>> num_trees = 2 * int(np.log(num_vectors).round(0)) ①
>>> index_cos.build(num_trees)
>>> index_cos.save('Word2vec_index.ann')
True
```

① 30 indexing trees

This should take twice as long to run the indexing, but once it finishes we should expect results closer to what gensim produces. Now let's see how approximate those nearest neighbors of for term "Harry Potter" for our more precise index:

```
>>> w2id = dict([(w, i) for i, w in enumerate(wv.vocab)])
>>> index_cos.get_nns_by_item(w2id['Harry_Potter'], 10)
[9494, 37681, 40544, 41526, 14273, 165465, 32643, 420722, 147151, 28829]
>>> [wv.index2word[i] for i in _]
['Harry_Potter',
'JK_Rowling',
'Deathly_Hallows',
'Half_Blood_Prince',
'Twilight',
'Twilight_saga',
```

```
'Narnia',
'Potter_mania',
'Hermione_Granger',
'Da_Vinci_Code']
```

That's a bit better. At least the correct author is listed. Let's compare the results for the two annoy searches to the "correct" answer from gensim:

| | annoy_15trees | annoy_30trees |
|-------------------------|-----------------------|-------------------|
| gensim | | |
| JK_Rowling_Harry_Potter | Harry_Potter | Harry_Potter |
| JK_Rowling | Narnia | JK_Rowling |
| boy_wizard | Sherlock_Holmes | Deathly_Hallows |
| Deathly_Hallows | Lemony_Snicket | Half_Blood_Prince |
| Half_Blood_Prince | Spiderwick_Chronicles | Twilight |
| Rowling | Unfortunate_Events | Twilight_saga |
| Actor_Rupert_Grint | Prince_Caspian | Narnia |
| HARRY_Potter | Eragon | Potter_mania |
| wizard_Harry_Potter | Sorcerer_Apprentice | Hermione_Granger |
| HARRY_POTTER | RL_Stine | Da_Vinci_Code |

- ➊ We leave it to you to figure out how to combine these Top 10 lists into a single DataFrame

To get rid of the redundant "Harry_Potter" synonym, we should have listed the Top 11, and skipped the first one. But you can see the progression here. As we increase the number of annoy index trees we can push down the ranking of less relevant terms (like "Narnia") and insert more relevant terms from the true nearest neighbors (like "JK_Rowling" and "Deathly_Hallows").

And the approximate answer from the annoy index is significantly faster than the gensim index that provides exact results. And you can use this annoy index for any high dimensional, continuous, dense vectors that you need to search, like LSA document-topic vectors or Doc2vec document embeddings (vectors).

13.2.4 Why Use Approximate Indexes at All?

Those of you with some experience analyzing algorithm efficiency may say to yourself that $O(N^2)$ algorithms are theoretically "efficient". After all they are more efficient than exponential algorithms and even more efficient than polynomial algorithms. They certainly aren't n-p hard to compute or solve. They aren't the kind of impossible thing that takes the lifetime of the universe to compute.

Since these $O(N^2)$ computations are only required to train the machine learning models in our NLP pipeline, they can be precomputed. Our chatbot doesn't need to compute $O(N$

2) operations with each reply to a new statement. And N^2 operations are inherently "parallelizable". We can almost always run one of the N sequences of computations independent of the other N sequences. So you could just throw more RAM and processors at the problem and just run some batch training process every night or every weekend to keep your bot's brain up to date.²²¹ Even better, you may be able to just bite off chunks of the N^2 computation and run them one by one, incrementally, as data comes in that increases that N.

Footnote 221 This is the real-world architecture we used on a N^2 document matching problem.

For example, imagine you've trained a chatbot on some small dataset to get started and then turned it loose on the world. Imagine that "N" is the number of statements and replies in it's persistent memory (database). Each time someone addresses the chat bot with a new statement, the bot might want to search it's database for the most similar statement so it can reuse any replies that worked for that statement in the past. So you compute some similarity score (metric) between the N existing statements and the new statement and store the new similarity scores in your $(N+1)^2$ similarity matrix as a new row and column. Or you just add n more "connections" or relationships to your graph data structure storing all the similarity scores between statements. Now you can just do a query on these connections (or cells in the connection matrix) to find the minimum distance value. For the simplest approach, you only really have to check those n scores you just computed, but if you wanted to be more thorough, you could check other rows and columns (walk the graph a little deeper) to find, for instance, some replies to similar statements and check metrics like kindness, information content, sentiment, grammatical, well-formedness, brevity, style. Either way you have an $O(N)$ algorithm for compute the best reply, even though the overall complexity for a "full" training run is $O(N^2)$.

But what if $O(N)$ still isn't enough. What if we're talking about a really big brain, like Google, where N is more than 60 trillion.²²². Even if your N isn't that large, if the individual computations are pretty complex, or you want to respond in a reasonable amount of time (100's of milliseconds) then you'll need to employ an index.

Footnote 222 <https://www.google.com/insidesearch/howsearchworks/thestory/> [Google tutorial on web indexing]

13.2.5 An Indexing Workaround: Discretizing

So we've just claimed that floats (real values) are impossible to naively index. What is one way to prove me wrong, or be less naive about our indexing? Those of you with experience working with sensor data and Analog to Digital Converters may be thinking to yourself that continuous values can easily be made digital or discrete. And a float isn't really continuous anyway. They're a bunch of bits, after all. But we need to make them *really* discrete if we want them to fit into our concept of an index and maintain that low dimensionality. We need to "bin" them into something manageable. The simplest way to turn a continuous variable into a manageable number of categorical or ordinal values is something like this:

```
from sklearn.preprocessing import MinMaxScaler
real_values = [-1.2, 3.4, 5.6, -7.8, 9.0]
# Confine our floats to be between 0.0 and 1.0
scaler = MinMaxScaler()
scaler.fit(real_values)

[int(x * 100.) for x in scaler.transform(real_values)] # scaled, discretized ints, 0 - 100
[39, 66, 79, 0, 100]
```

This works fine for low dimensional spaces. This is essentially what some 2D GIS indexes use to discretize lat/lon values into a grid of bounding boxes. Points in 2D space are either present or absent for each of the grid points.

As the number of dimensions grows we need to use more and more sophisticated efficient indexes than our simple 2D grid.

Let's use spatial dimensions to think about 3D space before diving into 300D natural language semantic vectors. For example, think about what changes when you grow from 2 to 3 dimensions by adding altitude to some database of 2D GPS latitude and longitude values. Now imagine you divided the Earth up into 3D cubes rather than our 2D grid that we used earlier. Most of those cubes wouldn't have much in them that humans would be interested in finding. And doing proximity searches, like finding all the objects within some 3D sphere or 3D cube becomes a much more difficult operation. The number of grid points you have to search through increases with N^{**3} where N is the "diameter" of a search region. You can see how when 3, the number of dimensions goes up to 4 or 5, you really need to be smart about your search.

13.3 Constant RAM Algorithms

13.3.1 Gensim

What if you have more documents than you can hold in RAM? As the size and variety of the documents in your corpus grows, you may eventually exceed the RAM capacity of even the largest machines you can rent from a cloud service. Have the fear, the mathematicians are here.

The math behind algorithms like Latent Semantic Analysis has been around for decades. This means mathematicians and computer scientists have had a lot of time to play with it and get it to work "out of core." "Out of core" just means that the objects required to run an algorithm do not all have to be present in core memory (RAM) at once. This means you are no longer limited by the RAM on your machine.

Even if you don't want to parallelize your training pipeline on multiple machines, constant RAM implementations will be required for large datasets. Gensim's `LsiModel` is one such "out of core" implementation of Singular Value Decomposition for Latent Semantic Analysis.²²³

Footnote 223 [gensim.models.LsiModel](#)

Even for smaller datasets the `gensim LSIModel` has the advantage that it doesn't require increasing amounts of RAM to deal with a growing vocabulary or set of documents. So you don't have to worry about it starting to SWAP to disk halfway through your corpus or grinding to a halt when it runs out of RAM. You can even continue to use your laptop for other tasks while a `gensim` model is training in the background.

Gensim uses what's called "batch training" to accomplish this. It trains your LSA model (`gensim.models.LsiModel`) on batches of documents and merges the results from these batches incrementally. All of gensim's models are designed to be "constant RAM," which makes them run faster on large data sets by avoiding swapping data to disk and using your precious CPU cache RAM efficiently.

TIP

In addition to being "constant RAM", the training of gensim models is "parallelizable," at least for many of the long-running steps in these pipelines.

So packages like `gensim` are worth having in your toolbox. They can speed up your small-data experiments like in this book, and also power your hyperspace travel on Big

Data in the future.

13.3.2 Graph Computing

Hadoop, TensorFlow, Caffe, Theano, Torch, and Spark were designed from the ground up to be "constant RAM". This means if you can formulate your machine learning pipeline as a Map-Reduce problem or a general computational graph, then you can take advantage of these frameworks to avoid running out of RAM. These frameworks automatically traverse your computational graph to allocate resources and optimize your throughput.

Peter Goldsborough implemented several benchmark models and datasets using these frameworks to compare their performance. Even though Torch has been around since 2002 it fared well on most of his benchmarks, outperforming all of the others on CPUs, and sometimes even on GPUs. In many cases it was 10 times faster than the nearest competitor.

An Torch (and it's PyTorch python API) is integrated into many cluster compute frameworks like RocketML. Though we haven't used PyTorch for the examples in this book, to avoid overwhelming you with options, it may be worth looking into if RAM or throughput are blockers for your NLP pipeline.

13.4 Parallelizing Your NLP Computations

13.4.1 Training NLP models on GPUs

Graphical Processing Units, often abbreviated with GPU, have become an important and sometimes necessary tool to develop real-world NLP applications. GPUs, first introduced in 2007, are designed to parallelize a large number of computational tasks and to access large amounts of memory. This is in contrast to the design of the *Central Processing Units* (CPU), which are the core of every computer. They are designed handle tasks sequentially at a high speed and they can access their limited processing memory at a high speed.

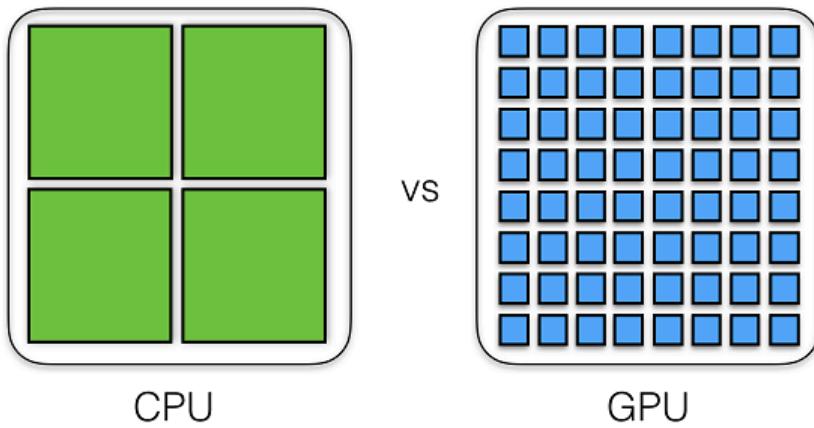


Figure 13.1 Comparison between a CPU and GPU

As it turns out, training deep learning models involves various operations which can be parallelized, for example the multiplication of matrices. Similar to graphical animations which were the initial target market for GPUs, the training of deep learning models is heavily accelerated by parallelized matrix multiplications.

Figure 2 shows the multiplication of an input vector with a weight matrix, a frequent operation during a forward-pass of the neural network training. The individual cores of a GPU are slow compared to a CPU, but each core can compute one of the result vector components. If the training is executed on a CPU, each row multiplication would be executed sequentially, assuming that no specific linear algebra library is used. It will require n (number of matrix rows) time steps to complete the multiplication. If the same task is executed on a GPU, the multiplication will be parallelized and each row multiplication can happen at the same time in the individual cores of the GPU.

$$\begin{bmatrix} w_{11} & \dots & w_{1n} \\ w_{21} & \dots & w_{2n} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ w_{m1} & \dots & w_{mn} \end{bmatrix}
 \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{bmatrix}
 = \begin{bmatrix} w_{11}x_1 + \dots + w_{1n}x_n \\ \vdots \\ \vdots \\ \vdots \\ w_{m1}x_m + \dots + w_{mn}x_n \end{bmatrix}$$

GPU

The diagram shows a weight matrix W and an input vector x . The result of their multiplication is shown as a column vector of dot products. Arrows point from the resulting vector to a GPU represented as a grid of blue squares, indicating that each element of the result vector is computed in parallel by one of the GPU's cores.

Figure 13.2 Matrix Multiplication where each row multiplication can be parallelized on a GPU

NOTE**Do I need to run my model on a GPU after the training is complete?**

You don't need to use a GPU for running your models in production, even if you used a GPU to train your model. The computation gain from computing a single forward-pass to compute a model predict does not usually warrant the additional complexity and cost of a GPU. The benefit of the GPU kicks in if you need to compute the forward and backward pass a few million times, like during training.

Once the training is completed, Keras or your Deep Learning framework provides you a way to export the model weights and structure. You can then load the weights and the model setup on almost any hardware to compute the model prediction (forward pass, or inference pass), even on a smart phone²²⁴ or in a browser²²⁵

Footnote 224 developer.apple.com/documentation/coreml or
www.tensorflow.org/mobile/tflite/

Footnote 225 transcranial.github.io/keras-js/#/

13.4.2 Rent vs. Buying

The use of GPUs can accelerate your model development, and allow you to iterate through your model development more quickly. Graphical Processing Units are very useful, but should you buy a high performance graphic card?

The answer in most cases is No. The performance of GPUs is improving so rapidly that a purchased graphic card could quickly get out-of-date. Unless you plan to use your GPU around the clock, you might be better off with renting a GPU via a service like *Amazon Web Services* or *Google Cloud*. The GPU service allows you to switch instance sizes between model training run. That way, you can scale up or down your GPU size depending to your needs. These providers also often provide fully configured installations which can save you time, and let you focus on your model development.

We built and maintained our own GPU server to speed some of the model training used in this book, but you should do as we say and not as we do. Selecting components that are compatible with each other and minimizing the data throughput bottlenecks was a challenge. We imitated successful architectures described by others and bought RAM and GPUs before the recent Bitcoin surge and the resulting High Performance Computing

(HPC) component price spike. Keeping all the libraries up to date and coordinating usage and configuration between authors was a challenge. It was fun and educational, but not an efficient use of our time nor our dollars.

The flexible setup of renting GPU instances has one drawback: you need to watch your costs closely. Completing your training won't stop your instance automatically. To stop the ticking of the meter (incurring ongoing cost), you will need to turn off your GPU instance between training runs. For more details, check out the section *Watch your Costs* in Appendix G.

13.4.3 GPU Rental Options

Various companies provide GPU rental options, starting with the well known *Platform as a Service* companies like *Microsoft*, *AWS* or *Google*. Other startups, like *PaperSpace* or *FloydHub* are breaking into the industry with interesting product offerings which can get you started quickly with your deep learning project.

Table 1 compares the different GPU options from *Platform-as-a-Service* providers. The services range from a bare GPU machine with a minimal installation to fully configured machines with drag&drop clients. Due to the regional variability in the service pricing, we can't compare the providers based on price. Price for the service range from \$0.65 to multiple dollars per hour and instance depending on the servers location, configuration and setup.

Table 13.1 Comparison of GPU Platform-as-a-Service options

| Company | Why? | GPU Options | Ease to get started | Flexibility |
|---------------------------|--|--|---------------------|-------------|
| Amazon Web Services (AWS) | Wide range of GPU options, spot prices, available in various data centers around the world | NVIDIA GRID K520, Tesla M60, Tesla K80, Tesla V100 | Medium | High |
| Google Cloud | Integrates well into Google Cloud Kubernetes clusters | NVIDIA Tesla K80, Tesla P100 | Medium | High |
| Microsoft Azure | Good option if you are using other Azure services | NVIDIA Tesla K80 | Medium | High |
| Floyd Hub | Command line interface to bundle your code | NVIDIA Tesla K80, Tesla V100 | Easy | Medium |
| PaperSpace | Fully configured virtual servers, hosted iPython notebooks with GPU support | NVIDIA Maxwell, Tesla P5000, Tesla P6000, Tesla V100 | Easy | Medium |

TIP**Setting up your own GPU on AWS**

We have created a summary of the necessary steps for you to get started with your own GPU instance in Appendix G.

13.4.4 Tensor Processing Units

You might have heard of another abbreviation *TPU*, Tensor Processing Units. These computational units are highly optimized for deep learning. They are particularly efficient at computing back-propagation for Tensor Flow models. TPUs are optimized for multiplying tensors of any dimensionality and use specialized FPGA and ASIC chips to preprocess and transport data around. GPUs are optimized for graphical processing, which mostly consists of the 2D Matrix multiplications required to render and move around in a 3D game worlds.

Google claims that TPUs are ten times more power efficient at computing deep learning models than an equivalent GPU. At the time of writing this chapter, Google, which designed and invented the TPUs in 2015, just released TPUs to the general public in a beta stage (no service-level-agreement is provided). In addition, researchers can apply to become part of the TensorFlow Research Cloud²²⁶ to train their models on a TPUs.

Footnote 226 www.tensorflow.org/tfrc/

13.5 Reducing the Memory Footprint during Model Training

When you train your NLP models on a GPU and you train with a large corpus, you will probably eventually encounter the following error when training your model: *MemoryError*

Listing 13.1 Error Message if your Training Data exceeds the Memory of the GPU

```
Epoch 1/10
Exception in thread Thread-27:
Traceback (most recent call last):
  File "/usr/lib/python2.7/threading.py", line 801, in __bootstrap_inner
    self.run()
  File "/usr/lib/python2.7/threading.py", line 754, in run
    self._target(*self._args, **self._kwargs)
  File "/usr/local/lib/python2.7/dist-packages/keras/engine/training.py", line 606,
    in data_generator_task
      generator_output = next(self._generator)
  File "/home/ubuntu/django/project/model/load_data.py", line 54, in load_training_set
    rv = np.array(rv)
MemoryError
```

To achieve the high performance of GPUs, the units use their own internal GPU memory in addition to the CPU memory. The card's memory is usually limited to a few gigabytes and in most case not near as much as the CPU has access to. When you trained your model on a CPU, your training data was probably loaded into the computer memory in one large table or sequence of tensors. This isn't possible anymore with the memory restrictions by the GPU.

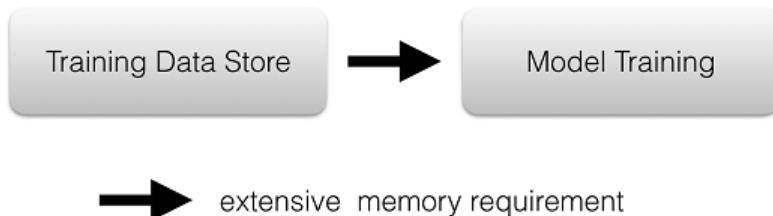


Figure 13.3 Loading the Training Data without a Generator Function

One efficient work-around is using Python's concept of a *generator*. A generator is a function which returns an *iterator* object. We can pass the iterator object to the model training method and it will "pull off" one or more training items at each training iteration. It never requires the whole training dataset in memory. This efficient way to reduce your memory footprint comes with caveats:

- Generators only provide one sequence element at a time, so we don't know how many elements it contains until we reach the end
- Generators can only be run once. They are disposable and not recyclable.

These two difficulties combine to make it a lot more tedious to make multiple training passes through our data. But Keras comes to the rescue. There are Keras methods that take care of all this tedious book-keeping for us:

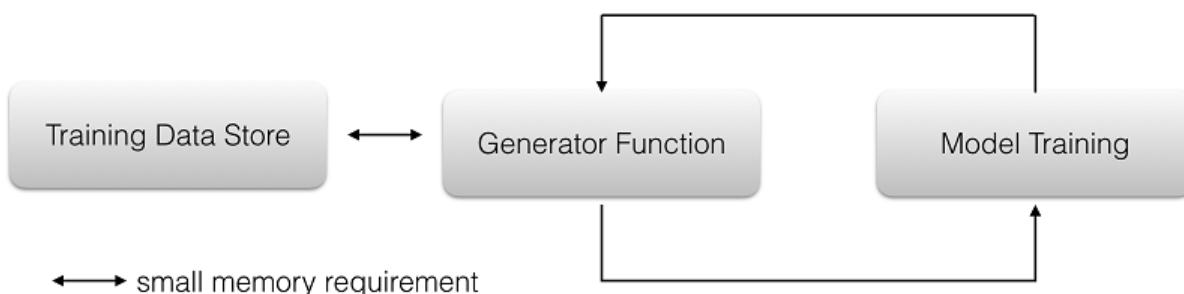


Figure 13.4 Loading the Training Data with a Generator Function

The generator function handles the loading of the training data store and returns the training "chunks" to the training methods. In the example code, the training data store is a csv file with the input data separated from the expected output data by the | delimiter. The "chunks" are limited to the batch size and only one batch at a time has to be stored in

memory. That way, we can heavily reduce the memory foot print of the model training data set.

```
import numpy as np

def training_set_generator(training_data_store, batch_size=32):
    X, Y = [], []
    while True:
        with open(training_data_store) as f:
            for i, line in enumerate(f):
                if i % batch_size == 0 and X and Y:
                    yield np.array(X), np.array(Y)
                    X, Y = [], []
                x, y = line.split('|')
                X.append(x)
                Y.append(y)

training_data_store = '/path/to/your/data.csv'
training_set = training_set_generator(training_data_store)
```

- ➊ In the function setup, you can set the batch size dynamically
- ➋ This endless loop will provide training batches forever; Keras will stop requesting more training examples when an epoch ends
- ➌ Opening the training data store and creating the file handler f
- ➍ Loop over the training data stores content line by line until your entire data has been served as training samples, afterwards start from the beginning of the training set
- ➎ If you have gathered enough train data samples, return the training data and the expected training output via a function yield. Python will jump back after the yield statement once the data is served to the model fit method
- ➏ If you don't have enough samples yet, read more lines, split them on the delimiter | and keep them in the lists X and Y

In our example, the `training_set_generator` function reads from a comma-separated values file, but it could load the data from any database, or any other data storage system.

One disadvantage of the generator is that it doesn't return any information about the size of the training data array. Since we don't know how much training data is available, we have to use slightly different *fit*, *predict* and *evaluate* methods of the Keras model.

Instead of using training our model with

```
model.fit(x=X,
          y=Y,
          batch_size=32,
          epochs=10,
```

```
verbose=1,
validation_split=0.2)
```

you have to kick off the training of your model with

```
training_data_store = '/path/to/your/data.csv'
model.fit_generator(generator=training_set_generator(training_data_store,
                                                       batch_size=32), ①
                     steps_per_epoch=100, ②
                     epochs=10, ③
                     verbose=1,
                     validation_data=[X_val, Y_val]) ④
```

- ① fit_generator expects a generator being passed to, which can be our training_set_generator or any other generator you program
- ② In contrast to defining your training batch_size like you did in the original fit method, the fit_generator expects the number of steps per epoch steps_per_epoch. For every step, the generator will be called. Set steps_per_epoch to training samples / batch_size, so that your model is exposed to the full training set once per epoch
- ③ Set your number of epochs as usual
- ④ Since the full training data isn't available to the fit_generator, it doesn't allow the usual validation_split; instead you need to define validation_data

If you use a generator, you might also want to update your model's *evaluate* and *predict* methods with

```
model.evaluate_generator(generator=your_eval_generator(eval_data, batch_size=32), steps=10)
```

and

```
model.predict_generator(generator=your_predict_generator(prediction_data, batch_size=32), steps=10)
```

WARNING Generators are memory efficient, but they can also become a *bottleneck* during the model training and slow down the training iterations. We highly recommend you pay attention to the generator speed while developing the training functions. In case, the on-the-fly processing slows down the generator, it might be beneficial to preprocess the training data and/or rent an instance with larger memory configuration.

13.6 Gaining Model Insights with TensorBoard

Wouldn't it be nice to get insights into your model performance while you train your model and compare it to previous training runs? Or quickly plot word embeddings to check semantic similarities? *Google's TensorBoard* provides you exactly that.

While training your model using TensorFlow (or with Keras and a TF backend), you can use Google's TensorBoard tool to gain insights into your NLP models. It can be used to track model training metrics, plot network weight distributions, visualize your word embeddings and various things. Tensorboard is easy to use and it connects to the training instance via your browser.

If you want to use TensorBoard side-by-side with Keras, you need to install TensorBoard like other Python package.

```
pip install tensorflow
```

Once, TensorBoard is installed, your can now start it up. We recommend to run it in a separate terminal.

```
tensorboard --logdir=/tmp/
```

Once TensorBoard is running, access it browser to 127.0.0.1:6006 if you train on your laptop or desktop PC. If you train your model on a rented GPU instance, use the public IP address of your GPU instance and make sure GPU provider allows access via the port 6006.

Once you are logged in, you can explore the model performance.

13.6.1 How to Visualize Word Embeddings

TensorBoard is a great tool to visualize word embeddings. Especially when you train your own, domain-specific word embeddings, the embedding visualization can help to verify semantic similarities. Converting a word model into a format TensorBoard can handle is straight forward. Once the word vectors and the vector labels are loaded into TensorBoard, it will perform the dimensionality reductions to 2D or 3D for you. TensorBoard provides currently three ways of dimensionality reductions: *PCA*, *t-SNE* and *custom* reductions.

The code below will convert your word embedding into a TensorBoard format and

generate the projection data.

Listing 13.2 Function to convert an embedding into a TensorBoard projection

```

import os
import tensorflow as tf
import numpy as np
from io import open
from tensorflow.contrib.tensorboard.plugins import projector

def create_projection(projection_data, projection_name='tensorboard_viz',
                      path='/tmp/'):
    1
    meta_file = "{}.tsv".format(projection_name)
    vector_dim = len(projection_data[0][1])
    samples = len(projection_data)
    projection_matrix = np.zeros((samples, vector_dim))

    with open(os.path.join(path, meta_file), 'w') as file_metadata:
        2
        for i, row in enumerate(projection_data):
            label, vector = row[0], row[1]
            projection_matrix[i] = np.array(vector)
            file_metadata.write("{}\n".format(label))

    3
    sess = tf.InteractiveSession()

    embedding = tf.Variable(projection_matrix, trainable=False, name=projection_name)
    tf.global_variables_initializer().run()

    saver = tf.train.Saver()
    writer = tf.summary.FileWriter(path, sess.graph)      4

    config = projector.ProjectorConfig()
    embed = config.embeddings.add()
    embed.tensor_name = "{}".format(projection_name)
    embed.metadata_path = os.path.join(path, meta_file)

    5
    projector.visualize_embeddings(writer, config)
    saver.save(sess, os.path.join(path, '{}.ckpt'.format(projection_name)))
    print('Run `tensorboard --logdir={}` to run visualize result on\n'
          'tensorboard'.format(path))

```

- 1 The `create_projection` function takes three arguments: the embedding data, a name for the projection and a path, where to store the projection files
- 2 The function will loop over the embedding data and create a numpy array which will then be converted to a TensorFlow variable
- 3 To create the TensorBoard projection, we need to create a TensorFlow session
- 4 TensorFlow provides built-in methods to create projections
- 5 `visualize_embeddings` will write the projection to our path and will be then available for TensorBoard

The function `create_projection` will take a list of tuples (expects the vector and then the label) and convert it into TensorBoard projection files. Once the projection files are

created and available to TensorBoard (in our case, TensorBoard expects the files in the `tmp` directory), head over to TensorBoard in your browser and check out the embedding visualization.

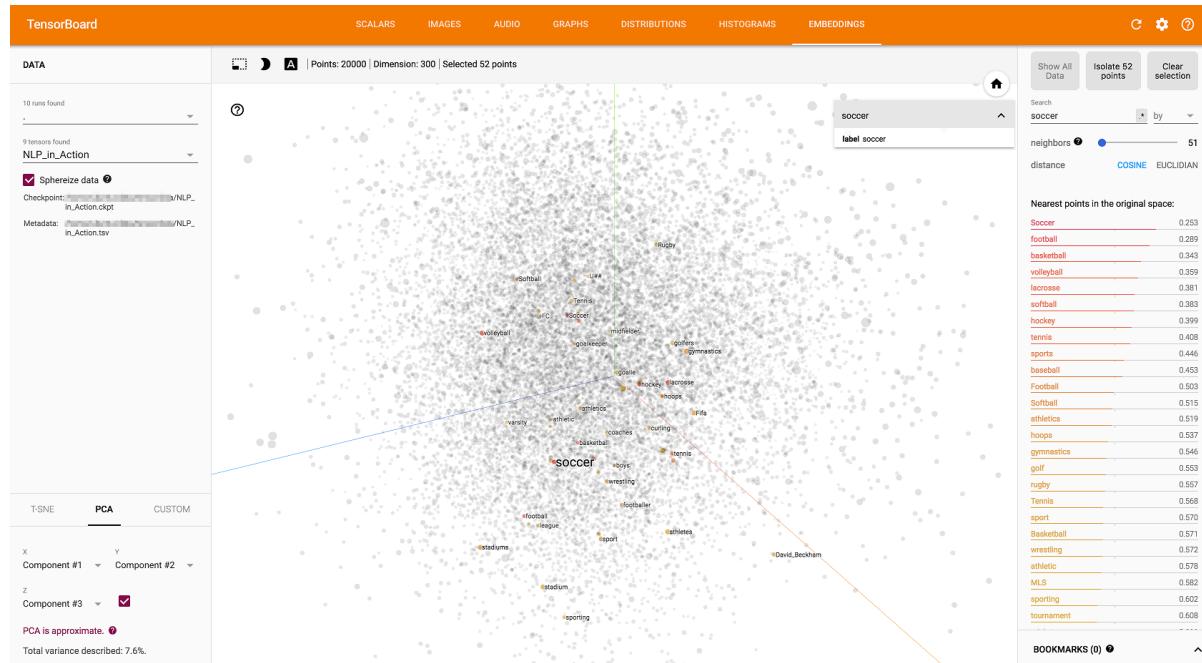
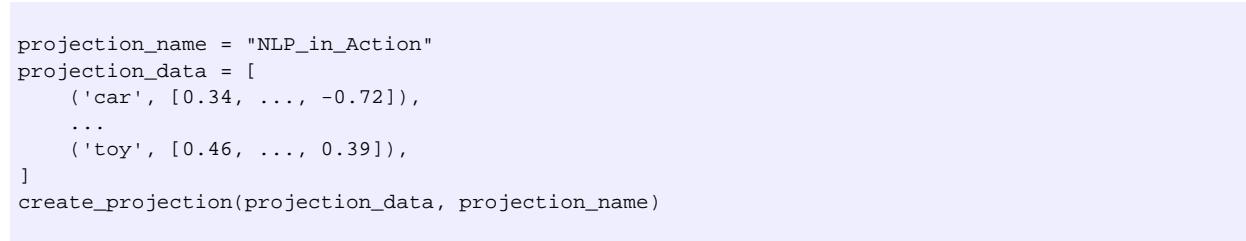


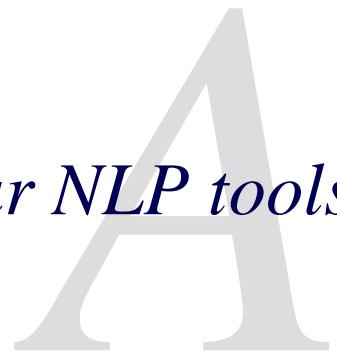
Figure 13.5 Using Tensorboard to visualize Google's word2vec embedding

13.7 Summary

In this chapter we discussed

- Semantic indexing is an effective way to find related documents in large datasets
- The training of Deep learning models can be sped up by using Graphical Processing Units (GPUs)
- That the memory limitations of GPUs can be overcome by using Python's generators. Converting your training loading methods to Python generators will save you money since you can achieve the same training results on a smaller GPU instance
- Google's TensorBoard is an excellent tool to investigate word embeddings

Your NLP tools



You can run all the examples in this book if you are able to install the `nlpia` package (github.com/totalgood/nlpia). We keep the installation instructions in the README file there up to date. But if you already have Python 3 installed, and you're feeling lucky (or are just lucky enough to have a Linux environment), you can try:

```
$ git clone https://github.com/totalgood/nlpia
$ pip3 install -e nlpia
```

If that doesn't work for you, you'll probably need to install a package manager and some binary packages for your OS. We have provided some OS-specific instructions in three sections:

- Ubuntu
- Mac
- Windows

These sections show you how to install an OS package manager. Once you have a package manager installed (or you're on a developer-friendly OS like Ubuntu that already has one), you can install Anaconda3.

A.1 Anaconda3

Python 3 has a lot of performance and expressiveness features that are really handy for NLP. And the easiest way to install Python 3 on almost any system is to install Anaconda3 (www.anaconda.com/download/). This has the added benefit of giving you a package and environment manager that can install a lot of problematic packages (such as `matplotlib`) on a wide range of problematic OSes (like Windows).

You can install the latest version of Anaconda and its `conda` package manager programmatically by running the following:

Listing A.1 Install Anaconda3

```
$ OS=MacOSX # or Linux or Windows
$ BITS=_64 # or '' for 32-bit
$ curl https://repo.anaconda.com/archive/ > tmp.html
$ FILENAME=$(grep -o -E "Anaconda3-[.0-9]+-$OS-x86$BITS\.(sh|exe)" tmp.html | head -n 1)
$ curl "https://repo.anaconda.com/archive/$FILENAME" > install_anaconda
$ chmod +x install_anaconda
$ ./install_anaconda -b -p ~/Anaconda
$ export PATH="$HOME/Anaconda/bin:$PATH"
$ echo 'export PATH="$HOME/Anaconda/bin:$PATH"' >> ~/.bashrc
$ echo 'export PATH="$HOME/Anaconda/bin:$PATH"' >> ~/.bash_profile
$ source ~/.bash_profile
$ rm install_anaconda
```

Now you can create a virtual environment, not a Python `virtualenv` but a more complete `conda` environment that isolates all of Python's binary dependencies from your OS Python environment. Then you can install the dependencies and source code for NLPIA within that `conda` environment with listing A.2.

A.2 Install NLPIA

We like to install software source code that we're working on in a subdirectory under our user `$HOME` called `code/`, but you can put it wherever you like. If this doesn't work, check out github.com/totalgood/nlpia for updated installation instructions.

Listing A.2 Install nlpia source with conda

```
$ mkdir -p ~/code
$ cd ~/code
$ git clone https://github.com/totalgood/nlpia
$ cd ~/code/nlpia
$ conda install -y pip
$ pip install --upgrade pip
$ conda env create -n nlpiaeenv -f conda/environment.yml
$ source activate nlpiaeenv
$ pip install --upgrade pip
$ pip install -e .
```

- 1
- 2
- 3
- 4
- 5
- 6

- 1 Install the latest conda binary for pip within your root conda environment.
- 2 Upgrade pip to the latest pypi.python.org version—Pip Installs Pip after all ;).
- 3 Create a conda environment, a directory in "\$HOME/Anaconda3/envs/nlpiae" with binary & source dependencies.

 ©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

- ➃ Activate your Python environment.
- ⑤ Install the latest pip within your nlpiaenv environment.
- ⑥ Install an "editable" source code directory for nlpia, so your changes source and data will go "live" whenever you save your edits to disk.

A.3 IDE

Now that you have Python 3 and NLPIA on your machine, you only need a good text editor to round out your integrated development environment (IDE). Rather than installing a complete system like PyCharm by JetBrains, we prefer individual tools with small teams (one-man teams in the case of Sublime Text) that do one thing well.

TIP

Built by developers for developers is a real thing, especially if the developer team is a team of one. Individual developers often build better tools than corporations because individuals are more open to incorporating code and suggestions by their users. An individual developer that builds a tool because she needs it usually build a tool that is optimized for her workflow. And her workflow is pretty awesome if she builds tools that are reliable, powerful, and popular enough to compete with. Large open source projects like `jupyter` are awesome, too, but in a different way. They're usually extremely versatile and full-featured, as long as they don't have a commercially licensed "fork" of the open source project.

Fortunately the tools you need for your Python IDE are all free, extensible, and continuously maintained. Most are even open source so you can make them your own:

- Sublime Text 3 (www.sublimetext.com/3) text editor with Package Control (packagecontrol.io/installation#st3) and Anaconda (packagecontrol.io/packages/Anaconda) "linter" plus auto-corrector
- Meld merge tool for Mac (yousseb.github.io/meld) or other OSes (meldmerge.org)
- `ipython` (`jupyter` console) for your *Read Eval Print Loop* (development workflow)
- `jupyter` notebook for creating reports, tutorials, and blog posts, or for sharing your results with your boss

TIP

Some phenomenally productive developers use a REPL workflow for Python.

¹ The ipython, jupyter console, and jupyter notebook REPL consoles are particularly powerful, with their help, ?, ??, and % magic commands, plus automatic tab-completion of attributes, methods, arguments, file paths, and even dict keys. Before Googling or overflowing your stack, explore the docstrings and source code Python packages you've imported by trying commands like >>> sklearn.linear_model.BayesianRidge???. Python's REPLs even allow you to execute shell commands (try >>> !git pull or >>> !find . -name nlpia) to keep your fingers on the keyboard, minimizing context switching and maximizing productivity.

Footnote 1 that's you, Steven "Digital Nomad" Skoczen and Aleck "The Dude" Landgraf

A.4 Ubuntu package manager

Your Linux distribution already has a full-featured package manager installed. And you may not even need it if you use Anaconda's package manager conda, as suggested in the NLPIA installation instructions (github.com/totalgood/nlpia). The package manager for Ubuntu is called apt. We've suggested some packages to install in A.3. You almost certainly will not need all these packages, but this exhaustive list of tools is here just in case you install something with Anaconda and it complains about a missing binary. You can start at the top and work your way down, until conda is able to install your Python packages.

Listing A.3 Install developer tools with apt

```
$ sudo apt-get update
$ sudo apt install -y build-essential libssl-dev g++ cmake swig git
$ sudo apt install -y python2.7-dev python3.5-dev libopenblas-dev libatlas-base-dev
    gfortran libgtk-3-dev
$ sudo apt install -y openjdk-8-jdk python-dev python-numpy python-pip python-virtualenv
    python-wheel python-nose
$ sudo apt install -y python3-dev python3-wheel python3-numpy python-scipy python-dev
    python-pip python3-six python3-pip
$ sudo apt install -y python3-pyaudio python-pyaudio
$ sudo apt install -y libcurl3-dev libcurl3-dev xauth x11-apps python-qt4
$ sudo apt install -y python-opencv-dev libxvidcore-dev libx264-dev libjpeg8-dev
    libtiff5-dev libjasper-dev libpng12-dev
```

TIP

If the `apt-get update` command fails with an error regarding `bazel`, you've likely added the Google `apt` repository with their build tool for TensorFlow. This should get you back on track again:

```
$ sudo apt-get install curl
$ curl https://bazel.build/bazel-release.pub.gpg | sudo apt-key add -
```

A.5 Mac

You need a real package manager (not XCode) before you can install all the tools you need to keep up with other developers.

A.5.1 A Mac package manager

Homebrew ([brew.sh/](#)) is probably the most popular command-line package manager for Macs among developers. It's easy to install and contains one-step installation packages for most tools that developers use. It's equivalent to Ubuntu's `apt` package manager. Apple could've ensured their OS would play nice with `apt`, but they didn't want developers to bypass their XCode and App Store "funnels", for obvious "monetization" reasons. So some intrepid Ruby developers homebrewed their own package manager.² And it's almost as good as `apt` or any other OS-specific binary package manager.

Footnote 2 [en.wikipedia.org/wiki/Homebrew_\(package_management_software\)](https://en.wikipedia.org/wiki/Homebrew_(package_management_software))

Listing A.4 Install brew

```
$ /usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You'll be asked to confirm things with the Return key and also enter your root/sudo password. So don't walk away to brew your coffee until you've entered your password and the installation script is happily chugging along.

A.5.2 Some packages

Once brew is installed, you may want to install some Linux tools that are handy to have around:

Listing A.5 Install developer tools

```
$ brew install wget htop tree pandoc asciidoctor
```

A.5.3 Tuneups

If you are serious about NLP and software development, you'll want to make sure you have your OS tuned up so you can get stuff done. Here's what we install whenever we create a new user account on a Mac:

- Snappy (snappy-app.com/)
- CopyClip (itunes.apple.com/us/app/copyclip-clipboard-history-manager/id595191960)

If you want to share screenshots with other NLP developers you'll need a screen grabber such as Snappy. And a clipboard manager, such as CopyClip lets you copy and paste more than one thing at a time and persist your clipboard history between reboots. A clipboard manager gives you the power of console history search ([ctrl]-[R]) in your GUI copy and paste world.

And you should also increase your bash shell history, add some safer `rm -f` aliases, set your default editor, create colorful text, and add `open` commands for your browser, text editor, and merge tool:

Listing A.6 bash_profile

```
#!/usr/bin/env bash
echo "Running customized ~/.bash_profile script: '$0' ...."
export HISTFILESIZE=10000000
export HISTSIZE=10000000
# append the history file after each session
shopt -s histappend
# allow failed commands to be re-edited with Ctrl-R
shopt -s histreedit
# command substitutions are first presented to user before execution
shopt -s histverify
# store multiline commands in a single history entry
shopt -s cmdhist
# check the window size after each command and, if necessary, update the values of LINES
# and COLUMNS
shopt -s checkwinsize
# grep results are colorized
export GREP_OPTIONS='--color=always'
# grep matches are bold purple (magenta)
export GREP_COLOR='1;35;40'
# record everything you ever do at the shell in a file that won't be unintentionally
# cleared or truncated by the OS
export PROMPT_COMMAND='echo "# cd $PWD" >> ~/.bash_history_forever; '$PROMPT_COMMAND
export PROMPT_COMMAND="history -a; history -c; history -r;
    history 1 >> ~/.bash_history_forever; $PROMPT_COMMAND"
# so it doesn't get changed again
readonly PROMPT_COMMAND
# USAGE: subl http://google.com # opens in a new tab
if [ ! -f /usr/local/bin/firefox ]; then
    ln -s /Applications/Firefox.app/Contents/MacOS/firefox /usr/local/bin/firefox
fi
alias firefox='open -a Firefox'
# USAGE: subl file.py
if [ ! -f /usr/local/bin/subl ]; then
    ln -s /Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl
```

```

/usr/local/bin/subl
fi
# USAGE: meld file1 file2 file3
if [ ! -f /usr/local/bin/meld ]; then
    ln -s /Applications/Meld.app/Contents/MacOS/Meld /usr/local/bin/meld
fi
export VISUAL='subl -w'
export EDITOR="$VISUAL"
# you can use -f to override these interactive nags for destructive disk writes
alias rm="rm -i"
alias mv="mv -i"
alias ..="cd .."
alias ...="cd ../../.."

```

You can find others' bash_profile scripts on GitHubGist (gist.github.com/search?q=%22.bash_profile%22+mac).

A.6 Windows

The command-line tools for package management, such as cygwin on Windows, aren't that great. But if you install GitGUI on a Windows machine, that gets you a bash prompt and a workable terminal that you can use to run your Python REPL console.

1. Download and install the git installer at git-scm.com/download/win.
2. Download and install the GitHub Desktop (desktop.github.com/).

The git installer comes with a version of the bash shell that should work well within Windows, but the git-gui that it installs is not very user friendly, especially for beginners. Unless you are using git from the command line (a bash shell within Windows), you should use GitHub Desktop for all your git push/pull/merge needs on Windows. We had problems throughout the editing of this book when git-gui did unexpected things that overwrote commits by others whenever there was a version conflict, even in files that weren't involved in the conflict. So that's why we ask you to install GitHub Desktop on top of raw git and git-bash. It gives you a more user-friendly git experience. (desktop.github.com/).³

Footnote 3 Big thanks to Benjamin Berg and Darren Meiss at Manning for figuring this out, and for all the other blocking and tackling of words and code they did to make this book decent.

Once you have a shell running in a Windows terminal you can install Anaconda and use the conda package manager to install the nlpia package just like the rest of us, using the instruction in the github repository README (github.com/totalgood/nlpia).

A.6.1 Get Virtual

If you get frustrated with Windows you can always install VirtualBox or Docker and create a virtual machine with an Ubuntu OS. That's the subject of a whole book (or at least a chapter), and there are better people at that than we are

VirtualBox

- Jason Brownlee ([linux-virtual-machine-machine-learning-development-python-3/](#))
- Jeroen Janssens ([datasciencetoolbox.org/](#))

Docker Container

- Vik Paruchuri ([dataquest.io/blog/docker-data-science/](#))
- Jamie Hall ([blog.kaggle.com/2016/02/05/how-to-get-started-with-data-science-in-containers/](#))

Another way to get Linux into your Windows world is with Microsofts Ubuntu shell app. I've not used it, so I can't vouch for its compatibility with the Python packages you'll need to install. If you try it, share what you learn with us at the `nlpia` repository with a feature or pull request on the documentation ([github.com/totalgood/nlpia/issues](#)). The Manning NLPIA forum ([natural-language-processing-in-action](#)) is also a great place to share your knowledge and get assistance.

A.7 NLPIA automagic

Fortunately for you, `nlpia` has some automatic environment provisioning procedures that will download the NLTK, Spacy, Word2vec models, and the data you need for this book. These downloaders will be triggered whenever you call an `nlpia` wrapper function, like `segment_sentences()` that requires any of these datasets or models. But this software is a work in progress, continually maintained and expanded by readers like you. So you may want to know how to manually install these packages and download the data you need to make them work for you when the automagic of `nlpia` fails. And you may just be curious about some of the datasets that make sentence parsing and part of speech taggers possible. So, if you want to customize your environment, the remaining appendices show you how to install and configure the individual pieces you need for a full-featured NLP development environment.

Playful Python and regular expressions



To get the most out of this book, you'll want to get comfortable with Python. You'll want to be so comfortable that you can get playful. When things don't work, you'll need to be able to play around and explore to find a way to make it run correctly. And even when your code works, playing around may help you uncover insidious unexpected behavior or new, more efficient ways of doing things. Hidden errors and edge cases are very common in natural language processing, because there are so many different ways to say things in a language like English.

This Python primer should get you going by explaining the data structures that we use throughout this book:

- `str` and `bytes`
- `ord` and `chr`
- `.format()`
- `dict` and `OrderedDict`
- `list`, `np.array`, `pd.Series`
- `pd.DataFrame`

We also explain some of the patterns and built-in Python functions we sometimes use here and in the `nlpia` package:

- List comprehensions: `[x for x in range(10)]`
- Generators: `(x for x in range(10**10))`
- Regular expressions: `re.match('[A-Za-z]*', 'Hello World')`
- File openers: `open('path/to/file.txt')`

B.1 Working with strings

Natural language processing is all about strings. And strings have a lot of quirks in Python 3 that may take you by surprise, especially if you have a lot of Python 2 experience. So you'll want to play around with strings and all the ways you can interact with them so you are comfortable interacting with natural language strings.

B.1.1 String types (`str` and `bytes`)

Strings (`str`) are sequences of Unicode characters. If you use a non-ASCII character in a `str`, it may contain multiple bytes for some of the characters. Non-ASCII characters pop up a lot if you are copying and pasting from the Internet into your Python console or program. And some of them are hard to spot, like those curly asymmetrical quote characters and apostrophes.

When you open a file with the python `open` command, it will be read as a `str` by default. For example if you open

Bytes (`bytes`) are sequences of bytes, usually used to hold ASCII characters and Extended ASCII characters (with integer `ord` values greater than 128). They are also sometimes used to store RAW images, WAV audio files, or other binary data "blobs".

B.1.2 Templates in Python (`.format()`)

Python comes with a versatile string templating syntax that allows you to populate a string with the values of variables.

B.2 Mapping in Python (`dict` and `OrderedDict`)

Hash table (or mapping) data structures are built into Python in `dict` objects. But a `dict` doesn't enforce a consistent key order, so the `collections` module, in the standard Python library, contains an `OrderedDict` that allows you to store key-value pairs in a consistent order that you can control (based on when you insert a new key).

B.3 Regular expressions

Regular expressions are little computer programs with their own programming language. Each regular expression string like "[a-z]*" can be compiled into a small program designed to be run on other strings to find matches. We provide a quick reference and some examples here, but you'll probably want to dig deeper in some online tutorials, if you're serious about NLP. As usual, the best way to learn is to play around at the command line. The `nlpia` package has a lot of natural language text documents and some useful regular expression examples for you to play with.

A regular expression defines a sequence of conditional expressions (`if` in Python) that each work on a single character. The sequence of conditionals forms a tree that eventually concludes in a single answer to the question "is the input string a 'match' or not". Because each regular expression can only match a finite number of strings and has a finite number of conditional branches, it defines a finite state machine (FSM).⁴

Footnote 4 This is only true for strict regular expression syntaxes that don't "look-ahead" and "look-behind".

The `re` package is the default compiler in Python, but the new official package is `regex` and can be easily installed with the `pip install regex`. It's more powerful, but we don't use those extra features, so you can use either one for the examples shown here. You only need to learn a few regular expression symbols to solve the problems in this book:

- `|`—The "OR" symbol
- `()`—Grouping with parentheses, just like in Python expressions
- `[]`—Character classes
- `\s, \b, \d, \w`—Shortcuts to common character classes
- `*, ?, +`—Some common shortcuts to character class occurrence count limits
- `{ }`—When the `*, ?, +` aren't enough, you can specify exact count ranges with curly braces.

B.3.1 |—"OR"

The `|` symbol is used to separate strings that can alternatively match the input string to produce an overall match for the regular expression. So the regular expression `Hobson|Cole|Hannes` would match any of the given names (first names) of the authors of this book:

`()`—Groups

You can use parentheses

B.3.2 []—Character classes

Character classes are equivalent to OR (`|`) between a set of characters. So `[abcd]` is equivalent to `(a|b|c|d)`, and `[abc123]` is equivalent to `(a|b|c|d|1|2|3)`.

And if some of the characters in a character class are consecutive characters in the alphabet of characters (ASCII or Unicode), they can be abbreviated using a hyphen between them. So `[a-d]` is equivalent to `[abcd]` or `(a|b|c|d)`, and `[a-c1-3]` is an abbreviation for `[abc123]` and `(a|b|c|d|1|2|3)`.

CHARACTER CLASS SHORTCUTS

- \s—[\t\n\r]—Whitespace characters
- \b—A non-letter non-digit next to a letter or digit
- \d—[0-9]—A digit
- \w—[a-zA-Z0-9_]—A whitespace character

B.4 Mastery

Find an interactive coding challenge website to hone your python skills before you jump into a production project. You can do one or two of these a week while reading this book.

1. codingbat.com—fun challenges in an interactive web-based python interpreter.
2. www.datacamp.com/community/tutorials/pandas-tutorial-dataframe-python—Pandas DataFrame tutorial at DataCamp

Once you've mastered python syntax, you'll want to try your hand at some coding and data manipulating challenges:

1. [Donne Martin's Coding Challenges](https://github.com/dmlc/dl-learningspace)—an open source repository of Jupyter Notebooks and Anki flashcards to help you learn algorithms and data structures 2.



Vectors and matrices (linear algebra fundamentals)

Vectors and numbers are the language of machine thought. Bits are the most fundamental "number" that machine computation is based on, a little like letters (characters) are the most fundamental, irreducible part of words, the language of thought for humans. All mathematical operations can be reduced to a few logical operations on sequences of bits. Sequences of characters are processed by human brains when we read in an analogous way. So if we want to teach machines about our words, the first challenge is to come up with vectors to represent characters, words, sentences, and "intermediate concepts" that the machine will need to work with to produce seemingly intelligent behavior.

C.1 Vectors

A vector is just an ordered sequence of numbers or other vectors without any "skips". In scikit learn and numpy, a vector is a dense array, and it works a lot like a Python list of numbers. The main reason we use numpy arrays rather than Python lists is because they are much faster to work with (100x) and use much less memory (1/4). Plus you can specify vectorized operations like multiplying the entire array by a value without iterating through it with a for loop. This is *very* important when working with a lot of text that contains a lot of information to be represented in these vectors and numbers.

Listing C.1 Code listing

```
>>> import numpy as np
>>> np.array(range(4))
array([0, 1, 2, 3])
>>> np.arange(4)
array([0, 1, 2, 3])
>>> x = np.arange(0.5, 4, 1)
>>> x
array([ 0.5,  1.5,  2.5,  3.5])
>>> x[1] = 2
>>> x
array([ 0.5,  1.5,  2.5,  3.5])
>>> x.shape
```

```
(4, )
>>> x.T.shape
(4, )
```

The new properties shown here that an array has but a list doesn't are `shape` and `T`. The `shape` contains the length or size of each dimension (the number of numbers it holds). We use lowercase letters when we create arrays (or even just numbers), just like formal mathematical symbols. In linear algebra, physics, and engineering texts, these letters are often bolded, and sometimes embellished with an arrow above them (especially by professors on chalkboards and whiteboards).

If you've ever heard of a matrix, you've probably heard that it can be thought of as a vector of vectors, like this:

Listing C.2 Code listing

```
>>> np.array([range(4), range(4)])
>>> array([[0, 1, 2, 3],
   [0, 1, 2, 3]])
>>> X = np.array([range(4), range(4)])
>>> X.shape
(2, 4)
>>> X.T.shape
(4, 2)
```

The `T` attribute returns the *transpose* of the matrix. The transpose of a matrix is the matrix flipped along an imaginary diagonal from the top-left corner to the bottom-right. So the following matrix A

```
>>> A = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
>>> A
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

has a *transpose* of

```
>>> A.T
array([[1, 5],
       [2, 6],
       [3, 7],
       [4, 8]])
```

C.1.1 Distances

Distance or similarity between two vectors can be measured a lot of different ways.

First, let's look at Euclidean distance between some vectors from a simple NLP example

from Patrick Winston's AI lecture series.⁵

Footnote 5 Patrick Winston. 6.034 Artificial Intelligence. Fall 2010. Massachusetts Institute of Technology:
MIT OpenCourseWare, ocw.mit.edu. License: Creative Commons BY-NC-SA. Lecture 10:
ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lect

Let's say we have 2D term frequency (bag-of-word) vectors that count the occurrences of the words "hack" and "computer" in articles from two publications, *Wired Magazine* and *Town and Country*. And we want to be able to query that set of articles while researching something to find some articles about a particular topic. The query string has both the words "hacking" and "computers" in it. Our query string word vector is [1, 1] for the words "hack" and "computer" because our query tokenized and stemmed the words that way (see chapter 2).

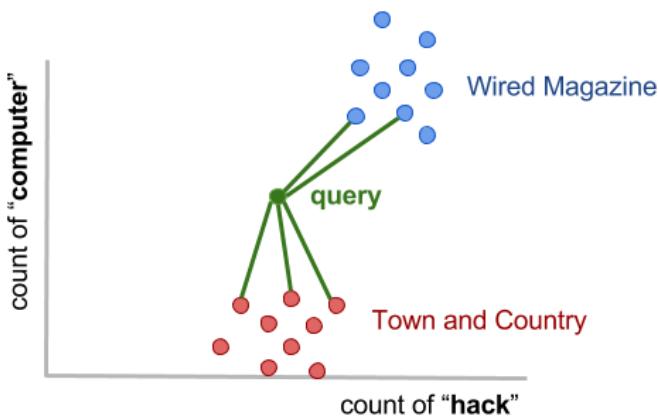


Figure C.1 Measuring Euclidean distance

Now which articles would you say are closest to our query in Euclidean distance? Euclidean distance is the length of the green lines in figure C.1. They look pretty similar don't they. How would you "fix" this problem so that your search engine returns some useful articles for this query?

You could compute the *ratio* of the word counts relative to the total number of words in a document and use these ratios to calculate your Euclidean distance. But you learned in chapter 3 about a better way to compute this ratio: TF-IDF. The Euclidean distance between TF-IDF vectors tends to be a good measure of the distance (inverse similarity) of documents.

But another "fix" makes our distance metric even better. Cosine distance between vectors will measure the angle between the two documents. So in this example, the angle between the query and the *Wired Magazine* articles would be much smaller than the angle between the query and the *Town and Country* articles. This is what we want.

Because a query about "hacking computers" should give us *Wired Magazine* articles and *NOT* articles about upper-crust recreational activities like horse riding ("hacking")⁶, duck hunting, dinner parties, and rustic interior design.

Footnote 6 Equestrian use of the word "hack": en.wikipedia.org/wiki/Hack_%28horse%29

COSINE DISTANCE

Cosine distance is the cosine of the angle between two vectors. This is efficiently computed as the dot product of two normalized vectors, vectors whose values have all been divided by the length of the vector.

Listing C.3 Code listing

```
>>> import numpy as np
>>> vector_query = np.array([1, 1])
>>> vector_tc = np.array([1, 0])
>>> vector_wired = np.array([5, 6])
>>> normalized_query = vector_query / np.linalg.norm(vector_query)
>>> normalized_tc = vector_tc / np.linalg.norm(vector_tc)
>>> normalized_wired = vector_wired / np.linalg.norm(vector_wired)

>>> normalized_query
array([ 0.70710678,  0.70710678])
>>> normalized_tc
array([ 1.,  0.])
>>> normalized_wired
array([ 0.6401844 ,  0.76822128])
```

The cosine similarity between our query TF vector and these other two TF vectors (cosine of the angle between them) is

```
>>> np.dot(normalized_query, normalized_tc) # cosine similarity
0.70710678118654746
>>> np.dot(normalized_query, normalized_wired) # cosine similarity
0.99589320646770374
```

The cosine distance between our query and these two TF vectors is one minus the cosine similarity.

```
>>> 1 - np.dot(normalized_query, normalized_tc) # cosine distance
0.29289321881345254
>>> 1 - np.dot(normalized_query, normalized_wired) # cosine distance
0.0041067935322962601
```

This is why cosine distance is used for TF vectors in NLP. It's also often used for TF-IDF vectors as well, but it's less critical because the Euclidean distance works OK as well.

Machine learning tools and techniques

Beyond natural language processing, the more generalized world of machine learning has some tools and techniques that are applicable to topic at hand. Some have been covered in earlier chapters, some have not, but all warrant at least a few words here.

D.1 Data selection (and avoiding bias)

Data selection, and to a lesser extent in NLP feature engineering, are ripe grounds for introducing bias (in human terms) into a model. Once baked in, there isn't much to do but discover it, throw out the model, and start over. For example, when the original Google model of Word2vec came out, trained on a vast array of news articles, excitement arose around the power of math on word vectors with such artistry as "king - man + woman = queen". But diving deeper, more problematic relationships revealed themselves in the model, such as "doctor - father + mother = nurse". A clear gender bias (similar racial biases were prevalent as well) was inadvertently built into the model. Almost assuredly this was not a direct effort of the researchers, but from the choice of data itself. A broad swath of the news articles simply had such cultural biases built in. Keeping the concept of bias in mind all the way through each step of the machine learning pipeline, from data selection to model validation and deployment, is vital. At any step along the way it's possible to introduce bias into the system.

Unfortunately, there are no easy answers in this realm.

The tendency to use the powers of these somewhat opaque algorithms as justifications for societal actions has already begun, whether through malice or oversight. The pattern uncovered by a model in data that is itself pre-biased will most assuredly appear biased. So careful statistical analysis of the source data should always be a priority.

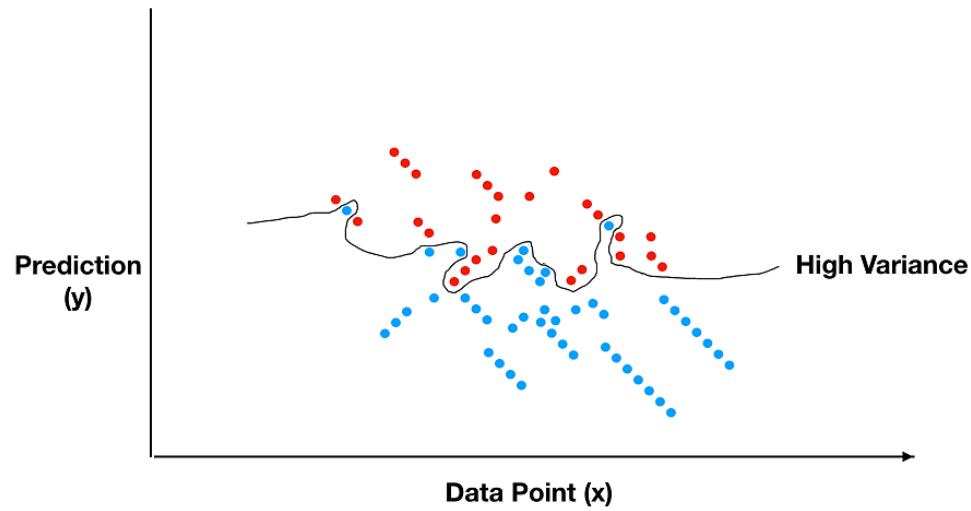
These pursuits often fall into statistical, psychological, and anthropological realms as

easily as they do that of computer science. As such they are outside the scope of this book, but we feel that they are of vital concern to machine learning practitioners, and answers should be pursued across disciplines. Otherwise the true power of these tools to uncover patterns will be little more than propaganda, and a wonderful opportunity will be missed.

On a more immediate level, handling the data you do have in relation to the models you build does have some immediately actionable approaches. Let's take a look.

D.2 How fit is fit?

With any machine learning model, one of the major challenges is overcoming the model's ability to do *too well*. How can something be "too good"? When working with example data in any model, the given algorithm may do very well at finding patterns in that particular dataset. But given that we already likely know the label of any particular example in the training set (or it wouldn't be in the training set), that is not particularly helpful. The real goal is to use those training examples to build a model that will *generalize*, and be able to correctly label an example that, while similar in scope and style to members of the training set, is verifiably outside of the training set. This also known as *performance on novel data*.



Model perfectly fits this dataset, but no other data points.

Figure D.1 Overfit on training samples

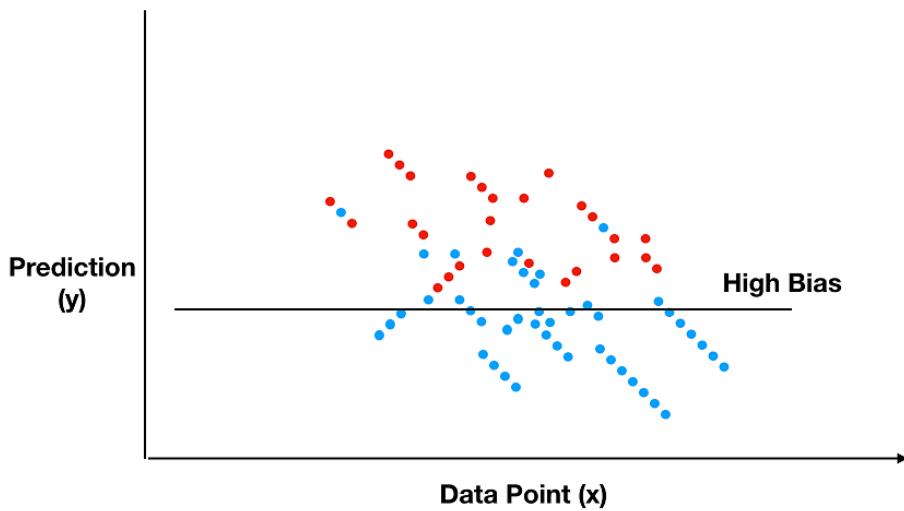


Figure D.2 Underfit on training samples

A model that perfectly describes (and predicts) on a given set of training data is perfectly *overfit* (see figure D.1). Such a model will have little or no capacity to describe novel data. Conversely, a model that always predicts the same output regardless of input data, be it in the training set or not, is perfectly *underfit* (see figure D.2). Neither of these kinds of models will be particularly helpful in practice, so let's look at techniques to detect these issues and, more importantly, ways to avoid them.

D.3 Knowing is half the battle

In machine learning practice, if data is gold, labeled data is ritarium (or whatever metaphor for what is most precious to you). One's first instinct may be to take every last bit of labeled data and feed it to the model, as more training data leads to a more resilient model, but that would leave us with no way to test the model short of throwing it out into the real world and hoping for the best. This obviously isn't practical. The solution is to split your labeled data into two and sometimes three datasets: a training set, a *validation* set, and in some cases a *test* set.

The training set is obvious. The validation set is a smaller portion of the labeled data we hold out and *never* show to the model for training. Good performance on the validation set is a first step to verifying that the trained model will perform well in the wild, as novel data comes in. You will often see an 80%/20% or 70%/30% split for training versus validation from a given labeled dataset. The *test* set is just like the validation set—a subset of the labeled training data to run the model against and measure

performance. But how is this *test* set different from the validation set then? In formulation, they are not different at all. The difference comes in how you use each of them.

While training the model on the training set, there will be several iterations with various hyperparameters; the final model you choose will be the one that performs the best on the validation set. But there's a catch. How do you know you haven't tuned a model that is merely highly biased toward the validation set? There is no way to verify that the model will perform well on data from the wild. So it's often advisable, if you can afford it in terms of data quantity, to hold a third chunk of the labeled dataset as a *test set*. Once the trained model is selected based on performance on the validation set, and you are no longer training at all, you can then infer from each sample in the test set. Should the model perform well against this third set of data, it will likely generalize well. For this kind of split, you will often see a 60%/20%/20% training/validation/test split.

TIP

Shuffling your dataset before you make the split between training, validation, and testing datasets is vital. You want each subset to be a representative sample of the "real world", and they need to have roughly equal proportions of each of the labels you expect to see. If your training set has 25% positive examples and 75% negative examples, you want your test and validation sets to have 25% positive and 75% negative examples, too. And if your original dataset had all the negative examples first and you did a 50/50 train/test split without shuffling the dataset first, you'd end up with 100% negative examples in your training set and 50/50 in your test set. Your model would never learn from the positive examples in your dataset.

D.4 Cross-fit training

Another approach to the train/test split question is *cross-validation* or *k-fold cross-validation* (see figure D.3). The concept behind cross validation is very similar to the rough splits we just covered, but it allows you a path to use the entire labeled set as training. The process involves dividing your training set into k equal sets, or *folds*. You then train your model with $k-1$ of the folds as a training set and validate it against the k -th fold. You then restart the training afresh with one of the $k-1$ sets used in training on the first attempt as your held-out validation set. The remaining $k-1$ folds become your new training set.

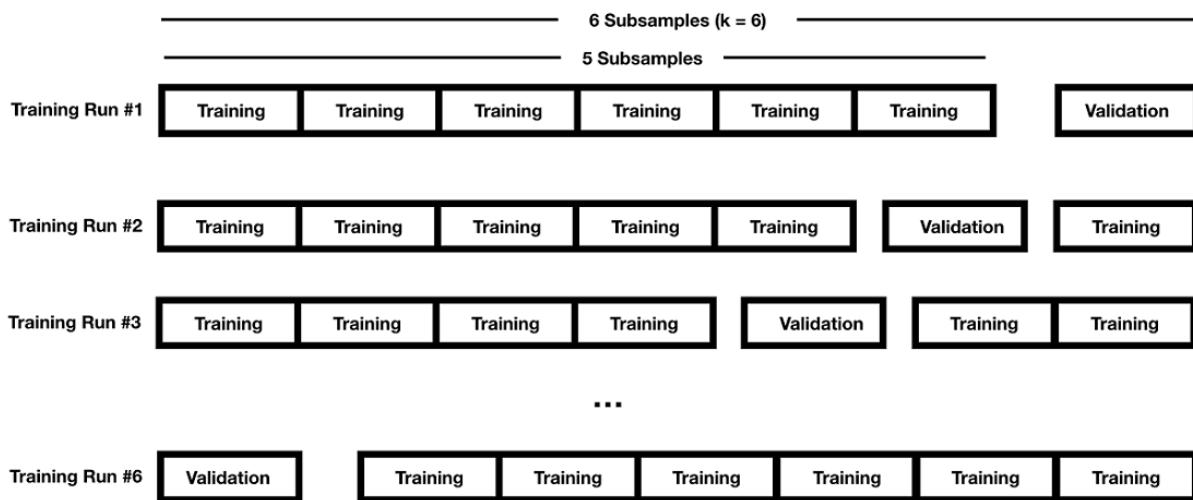


Figure D.3 K-fold cross-validation

This technique is valuable for analyzing the structure of the model and finding hyperparameters that perform well against a variety of validation data. Once your hyperparameters are chosen, you still have to select the *trained* model that performed the best and as such is susceptible to the bias expressed in the previous section, so holding a test set out from this process is still advisable.

D.5 Holding your model back

D.5.1 Regularization

In any machine learning model, overfitting will eventually come up. Luckily, several tools can combat it. The first is *regularization*, which is a penalization to the learned parameters at each training step. It is usually, but not always, a factor of the parameters themselves. *L1-norm* and *L2-norm* are the most common.

$$+\lambda \sum_{i=1}^n |w_i|$$

Figure D.4 L1 regularization

L1 is the sum of the absolute values of all the parameters (weights) multiplied by some lambda (a hyperparameter), usually a small float between 0 and 1. This sum is applied to the weights update—the idea being that weights that spike upwards or downwards cause a penalty to be incurred, and the model is encouraged to use more of its weights ... evenly.

$$+\lambda \sum_{i=1}^n w_i^2$$

Figure D.5 L2 regularization

Similarly, L2 is a weight penalization but defined slightly differently. In this case, it is the sum of the weights squared multiplied by some value lambda (a separate hyperparameter to be chosen ahead of training).

D.5.2 Dropout

In neural networks, *dropout* is another handy tool for this situation—one that is seemingly magical on first glance. Dropout is the concept that at any given layer of a neural network we will turn off a percentage of the signal coming through that layer at training time. Note that this occurs *only* during training, and never during inference. At any given training pass, a subset of the neurons in the layer below are "ignored"; those output values are explicitly set to zero. And because they have no input onto the resulting prediction, they will receive no weight update during the backpropagation step. In the next training step, a different subset of the weights in the layer will be chosen and those others are zeroed out.

How is a network supposed to learn anything with 20 percent of its brain turned off at any given time? The idea is that no specific weight path should define wholly a particular attribute of the data. The model must generalize its internal structures to be able to handle data via multiple paths through the neurons.

The percentage of the signal that gets turned off is defined as a hyperparameter, because it is a percentage that will be a float between 0 and 1. In practice, a dropout of .1 to .5 is usually optimal, but of course it's model dependent.

And at inference time, dropout is ignored and the full power of the trained is brought to bear on the novel data.

Keras provides a very simple way to implement this, and it can be seen in the book's examples.

Listing D.1 Code listing

```
>>> from keras.models import Sequential
>>> from keras.layers import Dropout, LSTM, Flatten, Dense
>>>
>>> num_neurons = 20
1
>>> maxlen = 100
```

```
>>> embedding_dims = 300
>>>
>>> model = Sequential()
>>>
>>> model.add(LSTM(num_neurons, return_sequences=True,
...                  input_shape=(maxlen, embedding_dims)))
>>> model.add(Dropout(.2)) ②
>>>
>>> model.add(Flatten())
>>> model.add(Dense(1, activation='sigmoid'))
```

- ① Arbitrary hyperparameters used as an example
- ② .2 here is the hyperparameter, so 20% of the outputs of the LSTM layer above will be zeroed out and therefore ignored.

D.5.3 BatchNorm

A newer concept in neural networks seems to have regularizing properties. *Batch Norm* is the idea that, much like the input data, the outputs of each layer should be normalized to values between 0 and 1. There is still some debate about how or why or when this is beneficial, and under which conditions it should be used. We leave it to you to explore that research on your own.

But Keras does provide a handy implementation of it.

Listing D.2 Code listing

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dropout, LSTM, Flatten, Dense
>>> from keras.layers.normalization import BatchNormalization
>>>
>>> model = Sequential()
>>>
>>> model.add(Dense(64, input_dim=14))
>>> model.add(BatchNormalization())
>>> model.add(Activation('sigmoid'))
>>> model.add(Dense(64, input_dim=14))
>>> model.add(BatchNormalization())
>>> model.add(Activation('sigmoid'))
>>> model.add(Dense(1, activation='sigmoid'))
```

D.6 Imbalanced training sets

Machine learning models are only ever as good as the data you feed them. Having a huge amount of data is only helpful if you have examples that cover all the cases you hope to predict in the wild. And just covering each case once isn't necessarily enough. Imagine you are trying to predict whether an image is a dog or a cat. But you have a training set with 20,000 pictures of cats and only 200 pictures of dogs. If you were to train a model on this dataset, it would not be unlikely that the model would simply learn to predict any given image was a cat regardless of the input. And from the model's perspective that would be fine, right? I mean it would be correct in 99% of the cases from the training set. Of course, that's a bogus argument and that model is worthless. But totally outside the scope of any particular model, the most likely cause of this failure is the *imbalanced training set*.

Models, especially neural nets, can be very finicky regarding training sets, for the simple reason that the signal from an overly sampled class in the labeled data can overwhelm the signal from the small cases. The weights will more often be updated by the error generated by the dominant class and the signal from the minority class will be washed out. It isn't vital to get an exactly even representation of each class, because the models have the ability to overcome some noise. The goal here is just to get the counts into the same ballpark.

The first step, as with any machine learning task, is to look long and hard at your data. Get a feel for the details and run some rough statistics on what the data actually represent. Find out not just how much data you have, but how much of which kinds of data you have.

So what to do if things aren't magically even from the beginning? If the goal is to even out the class representations (and it is), there are three main options: oversampling, undersampling, and augmenting.

D.6.1 Oversampling

Oversampling is the technique of repeating examples from the under-represented class or classes. Let's take the dog/cat example from earlier (only 200 dogs to 20,000 cats). You can simply repeat the dog images you do have 100 times each and end up with 40,000 total samples, half dogs/half cats.

This is an extreme example, and as such will lead to its own problems. The network will likely get very good at recognizing those specific 200 dogs and not generalize well to

other dogs not in the training set. But the technique of oversampling can certainly help balance a training set in cases that aren't so radically spread.

D.6.2 Undersampling

Undersampling is the opposite technique of the same coin. Here you just drop examples from the over-represented class. In the dog/cat example, we would just randomly drop 19,800 cat images and be left with 400 examples, half dog/half cat. This, of course, has a glaring problem of its own. We've thrown away the vast majority of the data and are working from a much less general footing. Extreme cases such as this aren't ideal but can be a good path forward if you have a large number of examples in the under-represented class. Having that much data is definitely a luxury.

D.6.3 Augmenting your data

This is a little trickier, but in the right circumstances, *augmenting* the data can be your friend. The concept of augmentation is to generate novel data, either from perturbations of the existing data or generating it from scratch. AffNIST (www.cs.toronto.edu/~tijmen/affNIST) is such an example. The famous MNIST dataset is a set of handwritten digits, 0-9 (see figure D.6). AffNIST takes each of the digits and skews, rotates, and scales them in various ways, while maintaining the original labels.



Figure D.6 The entries in the leftmost column are examples from the original MNIST, the other columns are all affine transformations of the data included in affNIST (image credit: www.cs.toronto.edu/~tijmen/affNIST)

The purpose of this particular effort wasn't to balance the training set but to make nets such as convolutional neural nets more resilient to new data written in other ways, but the concept of augmenting the data is the same.

But you must be cautious. Adding data that is not truly representative of that which you're trying to model can hurt more than it helps. Say your dataset is the 200/20,000 dogs/cats from earlier. And let's further assume that the images are all high-resolution color images taken under ideal conditions. Now handing a box of crayons to 19,000 kindergarteners would not necessarily get you the augmented data you desired. So think a bit about what augmenting your data will do to the model. The answer isn't always clear, so if you do go down this path, keep it in mind while you validate the resulting model and try to test around its edges to verify that you didn't introduce unexpected behavior unintentionally.

And lastly, probably the least helpful thing to say, but it is true, going back to the well to look for additional data should always be considered if your dataset is "incomplete". It isn't always feasible, but you should at least consider it as an option.

D.7 Performance metrics

The most important piece of any machine learning pipeline is the performance metric. If you don't know how well your machine learning model is working, you can't make it better. The first thing we do when starting a machine learning pipeline is set up a performance metric, such as ".score()" on any sklearn machine learning model. We then build a completely random classification/regression pipeline with that performance score computed at the end. This lets us make incremental improvements to our pipeline that gradually improve the score, getting us closer to our goal. It's also a great way to keep your bosses and coworkers convinced that you're on the right track.

D.7.1 Classification

Two important performance metrics for any machine learning classification problem are *precision* and *recall*. Information retrieval (search engines) and semantic search are examples of such problems, since your goal is to classify documents as a "match" or not.

D.7.2 Regression

The two most common performance scores used for machine learning regression problems are root mean square error (RMSE) and Pierson correlation (R2).

D.8 Pro tips

- Work with a random sample of a large dataset before training on all the data you have.
- First try the model and feature extractors you understand the best.
- Use scatter plots and scatter matrices on low-dimensional features.
- Plot high-dimensional data as a raw image to discover shifting across features.⁷

Footnote 7 Time series training sets will often be generated with a time shift and discovering this can help you on Kaggle competitions that hide the source of the data, like the Santander Value Prediction competition (www.kaggle.com/c/santander-value-prediction-challenge/discussion/61394).

- Try PCA on high-dimensional data (LSA on NLP data) when you want to maximize the *differences* between pairs of vectors (classification).
- Use nonlinear dimension reduction, like t-SNE, when you want to find *matches* between pairs of vectors or perform regression in the low-dimensional space.
- Build a `Pipeline` object to improve the maintainability/reusability of your model.
- Automate the hyperparameter tuning so your model can learn the data and you can spend your time learning about machine learning.

IMPORTANT

Hyperparameter tuning

Hyperparameters are all the values that determine the performance of your pipeline, including the selection of the model type, like the `sklearn` model class. It also includes the parameters that govern any preprocessing steps, like the tokenizer type, lists of stop words, lemmatizer, TF-IDF normalization approach, and so on. Tuning these parameters is what data scientists do to improve the performance of their models. Automating that process can help you spend more time reading books like this and less time guessing at hyperparameters. You can still guide the tuning with your intuition by selecting the parameters that should be adjusted and over what range of values.



In writing this book we pulled from numerous resources. Here are some of our favorites.

In an ideal world, you could find these resources yourself simply by entering the heading text into a semantic search engine like [Duck Duck Go](#), [Gigablast](#), or [Qwant](#). But until Jimmy Wales takes another shot at [Wikia Search](#) or Google shares their NLP technology, we have to rely on 1990s-style lists of links like this. Check out the E.6 section if your contribution to saving the world includes helping open source projects that index the web.

E.1 Applications and project ideas

Here are some applications to inspire your own NLP projects.

- [Guess passwords from social network profiles](#)⁸

Footnote 8

sciemag.org/news/2017/09/artificial-intelligence-just-made-guessing-your-password-whole-lot-easier

- [Parking ticket lawyer bots](#)⁹—Chatbots can file parking ticket legal appeals for you in New York and London.

Footnote 9

theguardian.com/technology/2016/jun/28/chatbot-ai-lawyer-donotpay-parking-tickets-london-new-york

- [Gutenberg + Library of Congress](#)¹⁰—Automatic document classification according to the Library of Congress specification

Footnote 10 github.com/craigboman/gutenberg

- [Longitudinal Detection of Dementia Through Lexical and Syntactic Changes in Writing](#)¹¹—Masters thesis by Xian Le on psychology diagnosis with NLP

Footnote 11 <ftp://cs.toronto.edu/dist/gh/Le-MSc-2010.pdf>

- [Time Series Matching](#)¹²—Songs and other time series can be discretized and searched with dynamic programming algorithms analogous to Levenshtein distance.

Footnote 12 cs.nyu.edu/web/Research/Theses/wang_zhihua.pdf

- [NELL, Never Ending Learning](#)¹³—Publications for NELL, a constantly evolving knowledge base that learns by scraping natural language text

Footnote 13 rtw.ml.cmu.edu/rtw/publications

- [How the NSA identified Satoshi Nakamoto](#)—*Wired Magazine* and the NSA identified Satoshi Nakamoto using NLP (this blog called it "stylometry").
- [Stylometry](#) and [Natural Language Forensics](#)—Style/pattern matching and clustering of natural language text (also music and artworks) for authorship and attribution
- Scrape examples.yourdictionary.com for examples of various grammatically correct sentences with POS labels and train your own Parsey McParseface syntax tree and POS tagger.
- [Identifying "Fake News" with NLP](#) by Julia Goldstein and Mike Ghoul at NYC Data Science Academy.
- [simpleNumericalFactChecker](#) by [Andreas Vlachos](#) and information extraction (see chapter 11) could be used to rank publishers, authors, and reporters for truthfulness. Might be combined with "fake news" labeler mentioned earlier.
- [artificial-adversary](#) by Jack Dai, an intern at Airbnb—Obfuscates natural language text (such as 'ur gr8' 'you are great'). You could train a machine learning classifier to detect this obfuscation. You could also train a stemmer (an autoencoder with the obfuscator generating character features) to decypher obfuscated words so your NLP pipeline can handle obfuscated text without retraining. Thank you Aleck.

E.2 Courses and tutorials

Here are some good tutorials, demonstrations, and even courseware from renowned university programs, many of which include Python examples.

- [Speech and Language Processing](#) by David Jurafsky and James H. Martin—The next book you should read if you're serious about NLP. Jurafsky and Martin are more thorough and rigorous in their explanation of NLP concepts. They have whole chapters on topics that we largely ignore, like finite state transducers (FSTs), hidden Markov models (HMMs), part-of-speech (POS) tagging, syntactic parsing, discourse coherence, machine translation, summarization, and dialog systems.
- [MIT Artificial General Intelligence course \(6.S099\)](#)¹⁴ led by Lex Fridman Feb 2018—MIT's free, interactive (there's a public competition) AGI course. It's probably the most thorough and rigorous course on artificial intelligence engineering you can find.

Footnote 14 agi.mit.edu

- [Textacy](#)¹⁵—Topic modeling wrapper for SpaCy

Footnote 15 github.com/chartbeat-labs/textacy

- [MIT Natural Language and the Computer Representation of Knowledge¹⁶](#)

Footnote 16

ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-863j-natural-language-and-the-computer-repres

- [SVD¹⁷—Singular value decomposition by Kardi Teknomo, PhD.](#)

Footnote 17 people.revoledu.com/kardi/tutorial/LinearAlgebra/SVD.html

- [Intro to IR \(and NLP\)¹⁸—*An Introduction to Information Retrieval*, April 1, Online Edition, by Christopher Manning, Prabhakar Raghavan, and Hinrich Schutze at Stanford \(creators of the Stanford CoreNLP library\)](#)

Footnote 18 nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf

E.3 Research papers and talks

One of the best way to gain a deep understanding of a topic is to try to repeat the experiments of researchers and then modify them in some way. That's how the best professors and mentors "teach" their students, by just encouraging them to try to duplicate the results of other researchers they are interested in. You can't help but tweak an approach if you spend enough time trying to get it to work for you.

E.3.1 Vector space models and semantic search

- [Semantic Vector Encoding and Similarity Search Using Full Text Search Engines](#)—Jan Rygl et. al were able to use a conventional inverted index to implement efficient semantic search for all of Wikipedia.
- [Learning Low-Dimensional Metrics](#)—Lalit Jain, et al, were able to incorporate human judgement into pairwise distance metrics, which can be used for better decision-making and unsupervised clustering of word vectors and topic vectors. For example, recruiters can use this to steer a content-based recommendation engine that matches resumes with job descriptions.
- [RAND-WALK: A latent variable model approach to word embeddings](#) by Sanjeev Arora, Yuanzhi Li, Yingyu Liang, Tengyu Ma, Andrej Risteski—Explains the latest (2016) understanding of the "vector-oriented reasoning" of Word2vec and other word vector space models, particular analogy questions
- [Efficient Estimation of Word Representations in Vector Space](#) by Tomas Mikolov, Greg Corrado, Kai Chen, Jeffrey Dean at Google, Sep 2013—First publication of the Word2vec model, including an implementation in C++ and pretrained models using a Google News corpus
- [Distributed Representations of Words and Phrases and their Compositionality](#) by Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean at Google—Describes refinements to the Word2vec model that improved its accuracy, including subsampling and negative sampling
- [From Distributional to Semantic Similarity](#) 2003 Ph.D. Thesis by James Richard Curran—Lots of classic information retrieval (full-text search) research, including

TF-IDF normalization and page rank techniques for web search

E.3.2 Finance

- [Predicting Stock Returns by Automatically Analyzing Company News Announcements](#)—Bella Dubrov used gensim’s Doc2vec to predict stock prices based on company announcements with excellent explanations of Word2vec and Doc2vec.
- [Building a Quantitative Trading Strategy to Beat the S&P 500](#)—At PyCon 2016, Karen Rubin explained how she discovered that female CEOs are predictive of rising stock prices, though not as strongly as she initially thought.

E.3.3 Question answering systems

- [Visual question answering¹⁹](#)

Footnote 19 github.com/avisingh599/visual-qa

- [2005 tutorial²⁰](#)

Footnote 20 lml.bas.bg/ranlp2005/tutorials/magnini.ppt

- [2003 EACL tutorial by Lin Katz, University of Waterloo, Canada²¹](#)

Footnote 21 cs.uwaterloo.ca/~jimmylin/publications/Lin_Katz_EACL2003_tutorial.pdf

- [Simple question answering from scratch \(corenlp and nltk used for sentence segmenting and POS tagging\)²²](#)

Footnote 22

github.com/raoariel/NLP-Question-Answer-System/blob/master/simpleQueryAnswering.py

- [PiQASSo: Pisa Question Answering System by Attardi, et al, 2001—Uses traditional information retrieval \(IR\) NLP.²³](#)

Footnote 23 trec.nist.gov/pubs/trec10/papers/piqasso.pdf

E.3.4 Deep learning

- [Understanding LSTM Networks](#) by Christopher Olah—One of the most clear and correct explanations of LSTMs you’ll find anywhere
- [Learning Phrase Representations using RNN ... Translation](#) by Kyunghyun Cho, et al, 2014—Gated recurrent units are first introduced. This is what made LSTMs practical and effective for NLP and a lot of other applications.

E.3.5 LSTMs and RNNs

We had a lot of difficulty understanding the terminology and architecture of LSTMs. This is a gathering of the most cited references so you can let the authors "vote" on the right way to talk about LSTMs. The state of the Wikipedia page (and Talk page discussion) on LSTMs is a pretty good indication of the lack of consensus about what LSTM means.

- [Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation](#) by Cho, Bengio, et al—Explains how the contents of the memory cells in an LSTM layer can be used as an embedding that can encode variable length sequences and then decode them to a new variable length sequence with a potentially different length, translating or transcoding one sequence into another
- [Reinforcement Learning with Long Short-Term Memory](#) by Bram Bakker—Application of LSTMs to planning and anticipation cognition with demonstrations of a network that can solve the T-maze navigation problem and an advanced pole-balancing (inverted pendulum) problem
- [Supervised Sequence Labelling with Recurrent Neural Networks](#)—Thesis by Alex Graves with advisor B. Brugge; a detailed explanation of the mathematics for the exact gradient for LSTMs as first proposed by Hochreiter and Schmidhuber in 1997. But Graves' first language may not be English, and he fails to define terms like CEC or LSTM *block/cell* rigorously.
- [Theano LSTM documentation](#) by Pierre Luc Carrier and Kyunghyun Cho—Diagram and discussion to explain the LSTM implementation in Theano and Keras
- [Learning to forget: Continual prediction with LSTM. Neural computation](#) by F. A., Schmidhuber, J., & Cummins—Uses nonstandard notation for layer inputs (*yin*) and outputs (*youtu*) and internal hidden state (*h*). All math and diagrams are "vectorized."
- [Sequence to Sequence Learning with Neural Networks](#) by Ilya Sutskever, Oriol Vinyals, and Quoc V. Le at Google
- [Understanding LSTM Networks](#) 2015 blog by Charles Olah—lots of good diagrams and discussion/feedback from readers
- [Long Short-Term Memory](#) by Sepp Hochreiter and Jurgen Schmidhuber, 1997—Original paper on LSTMs with outdated terminology and inefficient implementation, but detailed mathematical derivation

E.4 Competitions and awards

- [Large Text Compression Benchmark](#)—Some researchers believe that compression of natural language text is equivalent to artificial general intelligence (AGI)
- [The Hutter Prize](#)—Annual competition to compress a 100 MB archive of Wikipedia natural language text. Alexander Rhatushnyak won in 2017.
- [Open Knowledge Extraction Challenge](#)

E.5 Datasets

Natural language data is everywhere you look. Language is the superpower of the human race, and your pipeline should take advantage of it.

- [Stanford Datasets](#)²⁴—Pretrained word2vec and GloVe models, multilingual language models and datasets, multilingual dictionaries, lexica, and corpora.

Footnote 24 nlp.stanford.edu/data/

- [Pretrained word2vec models](#)²⁵—The README for a word vector web API provides links to several word vector models, including the 300-D Wikipedia GloVe model.

Footnote 25 github.com/3Top/word2vec-api#where-to-get-a-pretrained-model

- [Some NLP Datasets](#)²⁶

Footnote 26 github.com/karthikncode/nlp-datasets

- [A LOT of NLP Datasets](#)²⁷

Footnote 27 github.com/niderhoff/nlp-datasets

- [Google’s International TTS](#)²⁸—Data and tools for i18n

Footnote 28 github.com/googlei18n/language-resources

- [Natural Language Processing in Action](#)²⁹— Python package with data loaders and preprocessors for all the NLP data you will ever need... until you finish this book ;)

Footnote 29 github.com/totalgood/nlpia

E.6 Search engines

E.6.1 Search algorithms

- [GPU-enhanced BidMACH](#)—BidMACH is a high-dimensional vector indexing and KNN search implementation, similar to the annoy python package. This paper explains an enhancement for GPUs that is 8x faster than the original implementation.
- [Erik Bernhardsson’s Annoy Package](#)—Erik built the annoy nearest neighbor algorithm at Spotify and continues to enhance it.
- [Erik Bernhardsson’s ANN Comparison](#)—Approximate nearest neighbor algorithms are the key to scalable semantic search, and Erik keeps the world abreast of the latest and greatest algorithms out there.

E.6.2 Open source search engines

- [BeeSeek](#)—Open source distributed web indexing and private search (hive search); no longer maintained
- [WebSphinx](#)—Web GUI for building a web crawler

E.6.3 Open source full-text indexers

Efficient indexing is critical to any natural language search application. Here are a few open source full-text indexing options. However, these "search engines" do not crawl the web, so you need to provide them with the corpus you want them to index and search.

- [Elastic Search](#)³⁰

Footnote 30 github.com/elastic/elasticsearch

- [Apache Solr](#)³¹

Footnote 31 github.com/apache/lucene-solr

- [Sphinx Search](#)³²

Footnote 32 github.com/sphinxsearch/sphinx

- [Xapian](#)³³—There are packages for Ubuntu that will let you search your local hard drive (like Google Desktop used to do).

Footnote 33 github.com/Kronuz/Xapiand

- [Indri](#)³⁴—Semantic search with a [Python interface](#)³⁵ but it doesn't seem to be actively maintained.

Footnote 34 www.lemurproject.org/indri.php

Footnote 35 github.com/cvangysel/pyndri

- [Gigablast](#)³⁶—Open source web crawler and natural language indexer in C++.

Footnote 36 github.com/gigablast/open-source-search-engine

- [Zettair](#)³⁷ Open source HTML and TREC indexer (no crawler or live example); last updated 2009

Footnote 37 www.seg.rmit.edu.au/zettair

- [OpenFTS](#)³⁸—Full text search indexer for PostgreSQL with Python API [PyFTS](#)³⁹

Footnote 38 openfts.sourceforge.net

Footnote 39 rhodesmill.org/brandon/projects/pyfts.html

E.6.4 Manipulative search engines

The search engines most of us use are not optimized solely to help you find what you need, but rather to ensure that you click links that generate revenue for the company that built it. Google's innovative second-price sealed-bid auction ensures that advertisers don't overpay for their ads,⁴⁰ but it doesn't prevent search users from overpaying when they click disguised advertisements. This manipulative search is not unique to Google, but any search engine that ranks results according to any other "objective function" other than your satisfaction with the search results. But here they are, if you want to compare and experiment.

Footnote 40 Cornell University Networks Course case study, "Google Adwords Auction",
blogs.cornell.edu/info2040/2012/10/27/google-adwords-auction-a-second-price-sealed-bid-auction

- Google
- Bing
- Baidu

E.6.5 Less manipulative search engines

To determine how "commercial" and manipulative a search engine was, I queried several engines with things like "open source search engine". I then counted the number of ad-words purchasers and click-bait sites were among the search results in the top 10. The following sites kept that count below one or two. And the top search results were often the most objective and useful sites, such as Wikipedia, Stack Exchange, or reputable news articles and blogs.

- [Alternatives to Google](#)⁴¹

Footnote 41

www.lifehack.org/374487/try-these-15-search-engines-instead-google-for-better-search-results

- [Yandex](#)—Surprisingly, the most popular Russian search engine (60% of Russian searches) seemed less manipulative than the top US search engines.
- [DuckDuckGo](#)
- [watson Semantic Web Search](#)—No longer in development, and not really a full text web search, but it is an interesting way to explore the semantic web (at least what it was years ago before watson was frozen)

E.6.6 Distributed search engines

Distributed search engines⁴²⁴³ are perhaps the least manipulative and most "objective" because they have no central server to influence the ranking of the search results. However, current distributed search implementations rely on TF-IDF word frequencies to rank pages, because of the difficulty in scaling and distributing semantic search NLP algorithms. However, distribution of semantic indexing approaches such as latent semantic analysis (LSA) and locality sensitive hashing have been successfully distributed with nearly linear scaling (as good as you can get). It's just a matter of time before someone decides to contribute code for semantic search into an open source project like Yacy or builds a new distributed search engine capable of LSA.

Footnote 42 en.wikipedia.org/wiki/Distributed_search_engine

Footnote 43 wiki.p2pfoundation.net/Distributed_Search_Engines

- **Nutch**—Nutch spawned Hadoop and itself became less of a distributed search engine and more of a distributed HPC system over time.
- **Yacy**—One of the few **open source** decentralized (at least federated) search engines and web crawlers still actively in use. Preconfigured clients for Mac, Linux, and Windows are available.

Glossary

We've collected some definitions of common natural language processing and machine language acronyms and terminology here.⁴⁴

Footnote 44 Bill Wilson at the university of New South Wales in Australia has a more complete NLP glossary here: www.cse.unsw.edu.au/~billw/nlpdict.html.

You can find some of the parsers and regular expressions we used to help generate this list in the `nlpia` python package at github.com/totalgood/nlpia.⁴⁵

Footnote 45 `nlpia.transcoders`: github.com/totalgood/nlpia/blob/master/src/nlpia/transcoders.py and
`nlpia.book_parser`: github.com/totalgood/nlpia/blob/master/src/nlpia/book_parser.py

F.1 Acronyms

- **AGI**
Artificial general intelligence—Machine intelligence capable of solving a variety of problems that human brains can solve
- **ANN**
Aproximate Nearest Neighbors—Finding the M closest vectors to a single vector in a set of N high-dimensional vectors is a $O(N)$ problem because you have to calculate your distance metric between every other vector and the target vector. This makes clustering an intractable $O(N^2)$
- **ANN**
Artificial neural network—See *artificial neural network*.
- **BOW**
Bag of words—A data structure (usually a vector) that retains the counts (frequencies) of words but not their order.
- **CEC**
Constant error carousel—A neuron that just outputs its input delayed by one time step. Used within an LSTM or GRU memory unit. This is the memory register for an LSTM unit and can only be reset to a new value by the forgetting gate interrupting this "carousel."
- **CNN**
Convolutional neural network—A neural network that is trained to learn *filters*, which are also know as *kernels* as part of a classifier in supervised training
- **DAG**
Directed acyclic graph—A network topology without any cycles, connections that loop back on themselves.
- **GRU**
Gated recurrent unit—A variety of long short-term memory network with shared parameters to cut computation time
- **HPC**
High performance computing—The study of systems that maximize throughput, usually by parallelizing computation with separate `map` and `reduce` computation stages
- **LSTM**
Long short-term memory—An enhanced form of a recurrent neural network that maintains a memory of state that itself is trained via backpropagation
- **MSE**
Mean squared error—The sum of the square of the difference between the desired output of a machine learning model and the actual output of the model
- **NLG**
Natural language generation—Composing text automatically, algorithmically; one of the most challenging tasks of natural language processing (NLP)
- **NLP**
Natural language processing—You probably know what this is by now. If not, see the

introduction to chapter 1.

- *NLU*

Natural language understanding—Often used in recent papers to refer to natural language processing with neural networks

- *pip*

Pip installs pip—The official python package manager that downloads and installs packages automatically from the "Cheese Shop" (pypi.python.org)

- *PR*

Pull request—The right way to request that someone merge your code into theirs. GitHub has some buttons and wizards to make this easy. This is how you can build your reputation as a conscientious contributor to open source.

- *ReLU*

Rectified linear unit—The most popular and successful activation function for image processing and NLP neural nets.

- *REPL*

Read–evaluate–print loop—The typical workflow of a developer of any scripting language that doesn't need to be compiled. The `ipython`, `jupyter console`, and `jupyter notebook` REPs are particularly powerful, with their `help`, `?`, `??`, and `%` magic commands, plus auto-complete, and Ctrl-R history search. Python's REPs even allow you to execute any shell command (including `pip`) installed on your OS (such as `!git commit -am 'fix 123'`). This lets your fingers stay on the keyboard and away from the mouse, minimizing cognitive load from context switches.

- *RNN*

Recurrent neural network—A neural network architecture that feeds the outputs of one layer into the input of an earlier layer. RNNs are often "unfolded" into equivalent feed forward neural networks for diagramming and analysis.

- *SMO*

Sequential minimal optimization—A support vector machine training approach and algorithm

- *SVM*

Support vector machine—A machine learning algorithm usually used for classification

F.2 Terms

- *Artificial neural network*
A computational graph for machine learning or simulation of a biological neural network (brain)
- *Cell*
The memory or "state" part of an LSTM unit that records a single scalar value and outputs it continuously ⁴⁶[Footnote 46 en.wikipedia.org/wiki/Long_short-term_memory](https://en.wikipedia.org/wiki/Long_short-term_memory)
- *Feed-forward network*
A "one-way" neural network that passes all its inputs through to its outputs in a consistent direction forming a computation directed acyclic graph (DAG) or tree
- *Morpheme*
Part of a word that contains meaning in and of itself
- *Net, network, or neural net*
Artificial neural network
- *Neuron*
A unit in a neural net whose function ($y = \tanh(w \cdot \text{dot}(x))$) is to take multiple inputs and output a single scalar value. This value is usually the weights for that neuron (w or w_i) multiplied by all the input signals (x or x_i) and summed with a bias weight (w_0) before applying an activation function like *tanh*. A neuron always outputs a scalar value, which is sent to the inputs of any additional hidden or output neurons in the network. If a neuron implements a much more complicated activation function than that, like the enhancements that were made to recurrent neurons to create an LSTM, it is usually called a *unit*, for example, an *LSTM unit*.
- *Predicate*
In English grammar, the predicate is the main verb of a sentence that is associated with the subject. Every complete sentence must have a predicate, just like it must also have a subject.
- *Skip-grams*
Pairs of tokens used as training examples for a word vector embedding, where any number of intervening words are ignored
- *Softmax*
Normalized exponential function used to squash the real-valued vector output by a neural network so that its values range between 0 and 1 like probabilities:
- *Subject*
The main noun of a sentence—every complete sentence must have a subject (and a predicate) even if the subject is implied, like in the sentence "Run!", where the implied subject is "you".
- *Unit*
Neuron or small collection of neurons that perform some more complicated nonlinear function to compute the output. For example an LSTM unit has a memory cell that records state, an input gate (neuron) that decides what value to remember, a forget gate (neuron) that decides how long to remember that value, and an output gate neuron that accomplishes the activation function of the unit (usually a sigmoid or *tanh()*). A unit is a

drop-in replacement for a neuron in a neural net that takes a scalar input and outputs a scalar value; it just has more complicated behavior.



Setting up your AWS GPU

This appendix gives you a brief introduction on how to set up your own AWS GPU instance.

G.1 Steps to create your AWS GPU instance

1. Go to aws.amazon.com and sign in or sign up to your AWS account. Once you are logged in, you can start your setup steps in the AWS Management Console.
2. Select EC2 to get to the EC2 Dashboard.

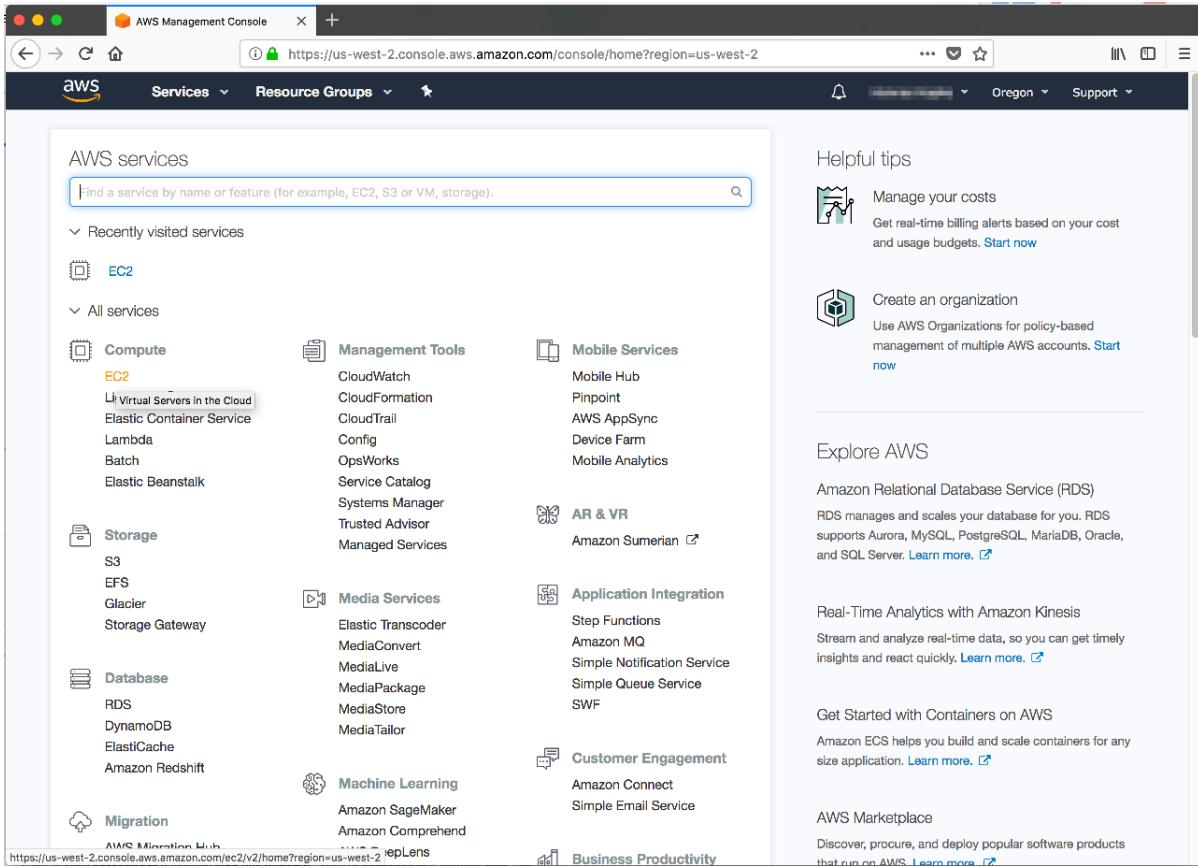


Figure G.1 AWS Management Console

3. In the EC2 Dashboard, click Launch Instance to start the instance setup.

The screenshot shows the AWS EC2 Management Console interface. The left sidebar contains a navigation menu with sections like EC2 Dashboard, Instances, Images, Elastic Block Store, Network & Security, and Load Balancing. The main content area is titled 'Resources' and displays statistics for the US West (Oregon) region: 0 Running Instances, 0 Dedicated Hosts, 1 Volumes, 2 Key Pairs, 0 Placement Groups, 0 Elastic IPs, 0 Snapshots, 0 Load Balancers, and 0 Security Groups. A promotional banner for EC2 Spot instances is visible. Below this, the 'Create Instance' section is shown, with a note that instances will launch in the US West (Oregon) region. A prominent blue 'Launch Instance' button is centered. To the right, there are sections for Service Health (showing normal status for US West (Oregon)) and Scheduled Events (no events). On the far right, there's an 'AWS Marketplace' section featuring Barracuda NextGen Firewall F-Series - PAYG, with a rating of ★★★★☆ and starting from \$0.60/hr or \$4,599/yr. Other links in this sidebar include Getting Started Guide, Documentation, All EC2 Resources, Forums, Pricing, and Contact Us.

Figure G.2 Create a new AWS instance

4. Amazon Web Services provides preconfigured server setups. They come with all major deep learning frameworks installed. That way, you don't need to worry about the detail of the CUDA installation, and so on. To find a deep learning machine image, search for "deep learning".⁴⁷

Footnote 47 At the time of this writing, the image was available under the AMI ID *ami-f1d51489*.

5. We tested our code on the Deep Learning AMI (Ubuntu). Click Select to proceed.

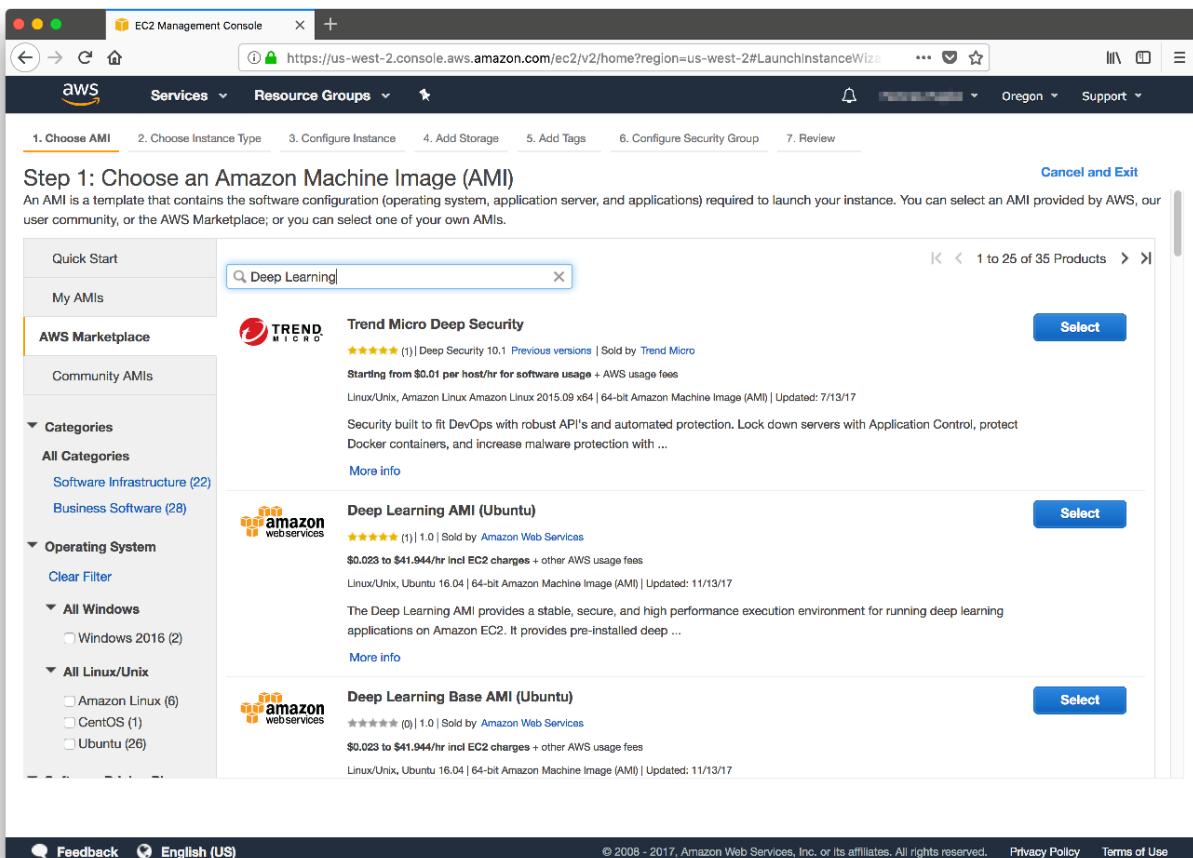


Figure G.3 Selecting an AWS Machine Image

6. The usage of the Amazon Machine Image is free, but you need to pay for the server instance, therefore the costs in the Software column are 0. Please note that GPU instances aren't covered by Amazon's free tier program. The price depends on the instance type.

The screenshot shows the AWS EC2 Management Console interface. The main title is "Deep Learning AMI (Ubuntu)". On the left, there's a sidebar with steps: 1. Choose AMI, 2. Choose Instance Type, 3. Configure Instance, 4. Add Storage, 5. Add Tags, 6. Configure Security Group, and 7. Review. The current step is Step 1: Choose AMI.

Product Details:

- Sold by: Amazon Web Services
- Customer Rating: ★★★★ (1)
- Latest Version: 1.0
- Base Operating System: Linux/Ubuntu 16.04
- Delivery Method: 64-bit Amazon Machine Image (AMI)
- License Agreement: [End User License Agreement](#)
- On Marketplace Since: 11/14/17
- AWS Services Required: Amazon EC2, Amazon EBS

Highlights:

- Used Ubuntu 16.04 (ami-cd0f5cb6) as the base AMI.

Pricing Details:

| Instance Type | Software | EC2 | Total |
|--------------------------------|----------|----------|--------------------|
| R3 Eight Extra Large | \$0.00 | \$2.66 | \$2.66/hr |
| M3 Extra Large | \$0.00 | \$0.266 | \$0.266/hr |
| R4 16 Extra Large | \$0.00 | \$4.256 | \$4.256/hr |
| M4 Extra Large | \$0.00 | \$0.20 | \$0.20/hr |
| Graphics Two Extra Large | \$0.00 | \$0.65 | \$0.65/hr |
| C3 Quadruple Extra Large | \$0.00 | \$0.84 | \$0.84/hr |
| High I/O Quadruple Extra Large | \$0.00 | \$1.248 | \$1.248/hr |
| T2 Large | \$0.00 | \$0.093 | \$0.093/hr |
| C4 Double Extra Large | \$0.00 | \$0.398 | \$0.398/hr |
| G2 Eight Extra Large | \$0.00 | \$2.60 | \$2.60/hr |
| R3 Double Extra Large | \$0.00 | \$0.665 | \$0.665/hr |
| C5 Large | \$0.00 | \$0.085 | \$0.085/hr |
| High Storage Eight Extra Large | \$0.00 | \$4.60 | \$4.60/hr |
| X1 32 Extra Large | \$0.00 | \$13.338 | \$13.338/hr |
| CC2 Cluster Compute | \$0.00 | \$2.00 | \$2.00/hr |
| T2 Double Extra Large | \$0.00 | \$0.371 | \$0.371/hr |
| T2 Extra Large | \$0.00 | \$0.186 | \$0.186/hr |

Buttons: Cancel, Continue

Figure G.4 Cost overview for the machine image and the available instance types in your AWS region

7. In this step you are selecting the server type for your machine. We recommend the smallest GPU instance: g2.2xlarge. Please note that Amazon will preselect a different type for you. The selected type provides more performance, but it is substantially more expensive.
8. From here, you can launch your machine by clicking Review and Launch. We recommend that you continue with the next few steps to customize your instance (optional).

The screenshot shows the AWS EC2 Management Console interface for launching a new instance. The current step is "Step 2: Choose an Instance Type". The table lists the following instance types:

| Type | Instance Type | Memory (GiB) | CPU Cores | Storage | Network | Termination Protection | Monitoring | Encryption at Rest |
|-------------------|---------------|--------------|-----------|---------------|---------|------------------------|------------|--------------------|
| Compute optimized | c3.xlarge | 4 | 7.5 | 2 x 40 (SSD) | Yes | Moderate | Yes | |
| Compute optimized | c3.2xlarge | 8 | 15 | 2 x 80 (SSD) | Yes | High | Yes | |
| Compute optimized | c3.4xlarge | 16 | 30 | 2 x 160 (SSD) | Yes | High | Yes | |
| Compute optimized | c3.8xlarge | 32 | 60 | 2 x 320 (SSD) | - | 10 Gigabit | Yes | |
| FPGA instances | f1.2xlarge | 8 | 122 | 1 x 470 (SSD) | Yes | Up to 10 Gigabit | Yes | |
| FPGA instances | f1.16xlarge | 64 | 976 | 4 x 940 (SSD) | Yes | 25 Gigabit | Yes | |
| GPU graphics | g3.4xlarge | 16 | 122 | EBS only | Yes | Up to 10 Gigabit | Yes | |
| GPU graphics | g3.8xlarge | 32 | 244 | EBS only | Yes | 10 Gigabit | Yes | |
| GPU graphics | g3.16xlarge | 64 | 488 | EBS only | Yes | 25 Gigabit | Yes | |
| GPU instances | g2.2xlarge | 8 | 15 | 1 x 60 (SSD) | Yes | High | - | |
| GPU instances | g2.8xlarge | 32 | 60 | 2 x 120 (SSD) | - | 10 Gigabit | - | |
| GPU compute | p2.xlarge | 4 | 61 | EBS only | Yes | High | Yes | |
| GPU compute | p2.8xlarge | 32 | 488 | EBS only | Yes | 10 Gigabit | Yes | |
| GPU compute | p2.16xlarge | 64 | 732 | EBS only | Yes | 25 Gigabit | Yes | |
| GPU compute | p3.2xlarge | 8 | 61 | EBS only | Yes | Up to 10 Gigabit | Yes | |

Buttons at the bottom include: Cancel, Previous, Review and Launch (highlighted in blue), and Next: Configure Instance Details.

Figure G.5 Choosing your instance type

9. Here you can configure the instance details. If you are already using AWS machine, you can assign your GPU machine to the same virtual private network. If the machine becomes a critical piece of your infrastructure and will contain important data, we recommend selecting Enable Termination Protection. That way, your machine can be stopped (which will wipe the tmp directory, but the machine can't be destroyed).
10. To continue, click Add Storage.

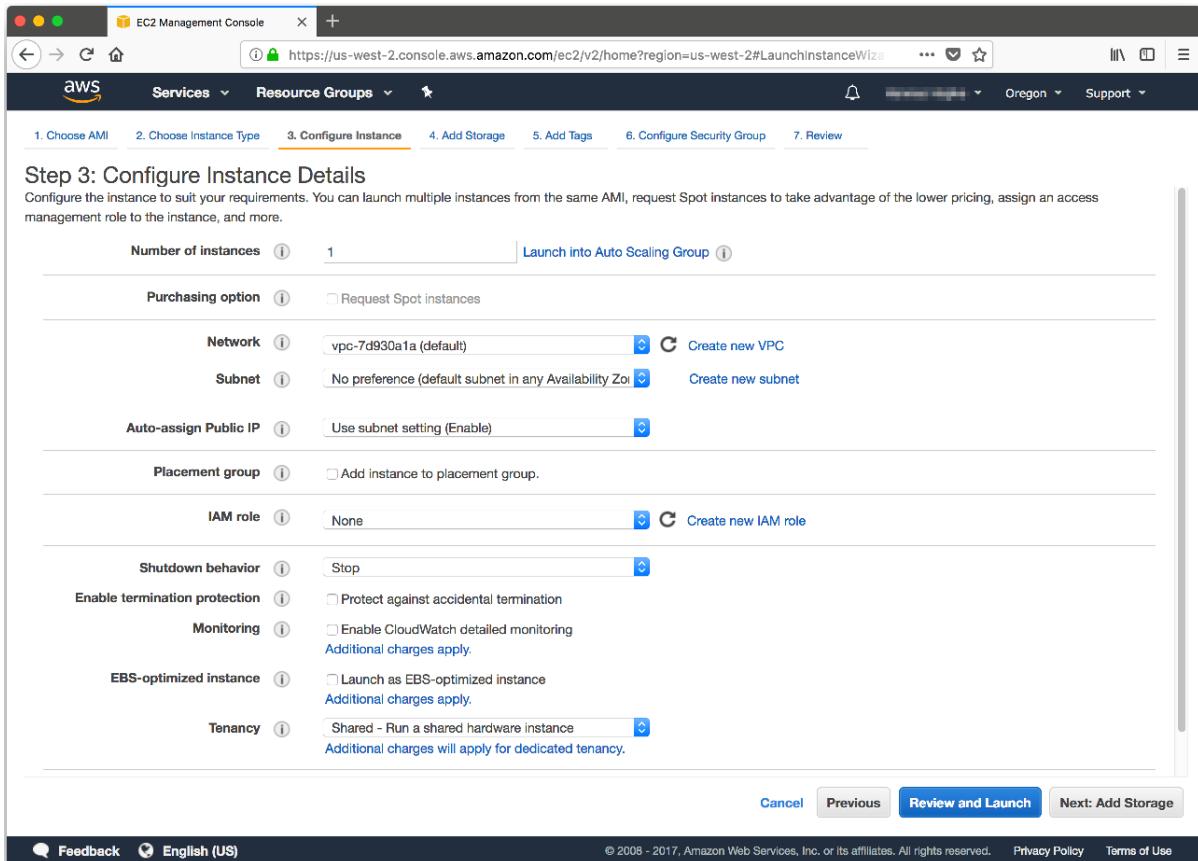


Figure G.6 Configuring instance details

11. If you plan to work with very extensive corpora, we recommend that you extend your storage space here. Please note that Amazon Web Service will charge you for any storage above the 30 GB free tier allowance.
12. The following instance details (such as tag setup) are for experienced users. Feel free to click Review and Launch now, or proceed to the optional steps.

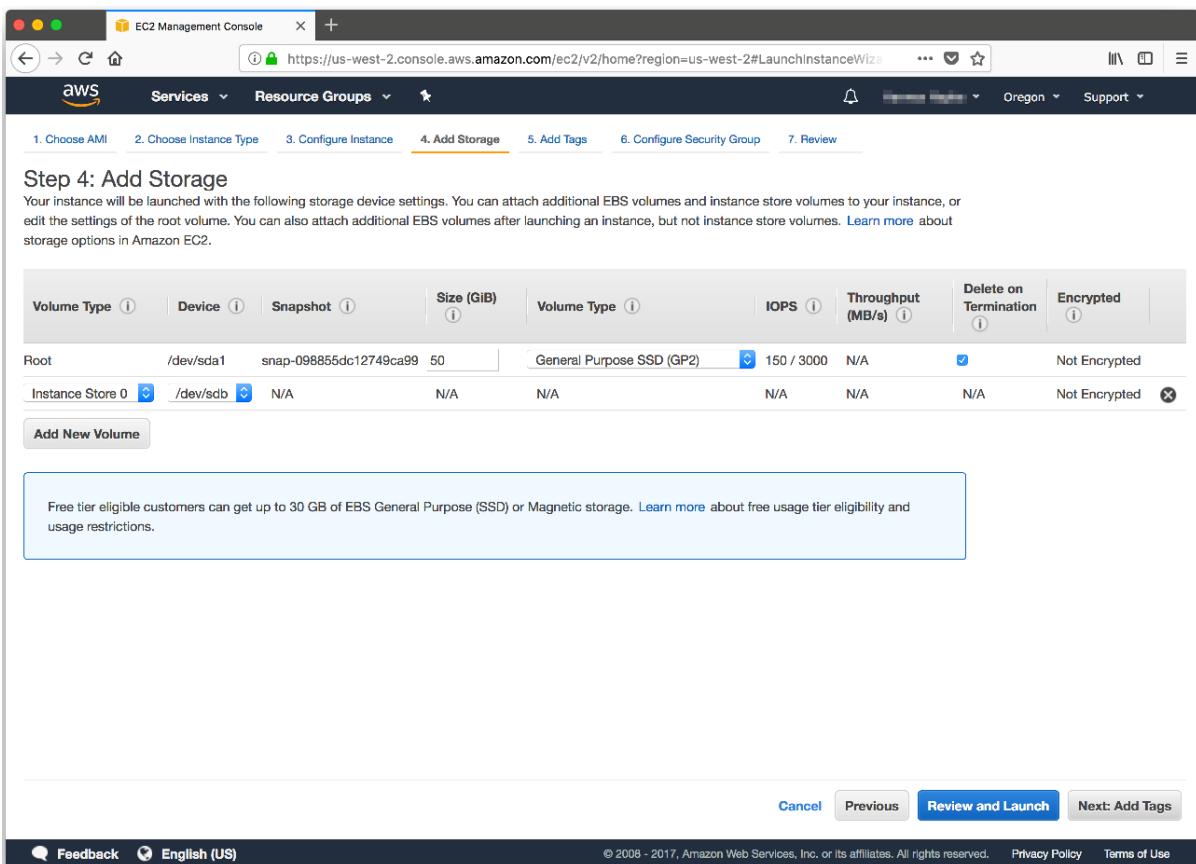


Figure G.7 Adding storage to your instance

13. On this screen, Amazon Web Services shows you all the details in one overview. Note that any selected GPU is not included in the free tier program. The price depends on your GPU instance type and your AWS region. US-West2 or often named after its location Oregon offers the lower prices compared to other data centers.
14. When you have confirmed all details, click Launch. AWS starts up your preconfigured machine.

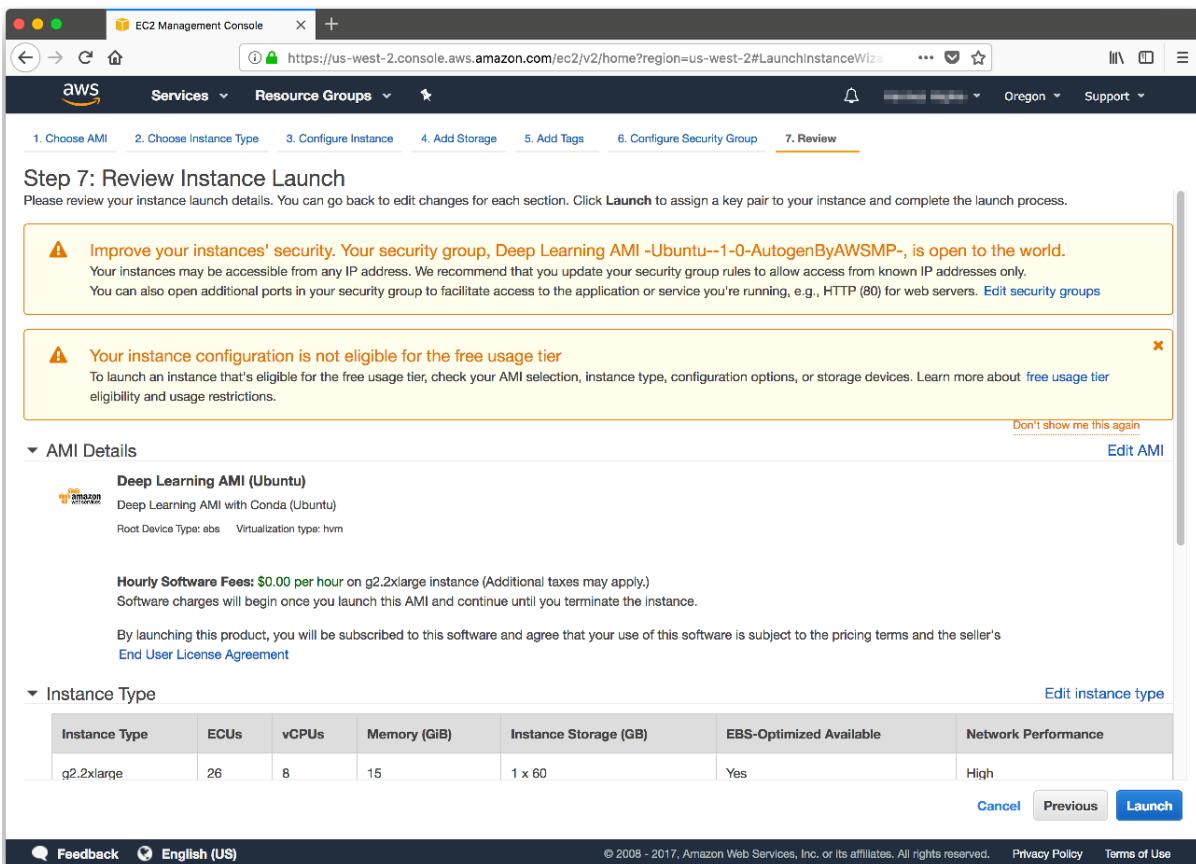


Figure G.8 Review your instance setup before launching

15. If you haven't created an instance with AWS, it asks you to create a new key pair. The key pair allows you to ssh into the machine without a password. Save the key pair in your `~/.ssh/` folder and keep the key pair in a safe place (such as your password manager) to access your machine in case you get locked out.
16. If you already have a key set up, feel free to select your existing pair instead of creating a new one.

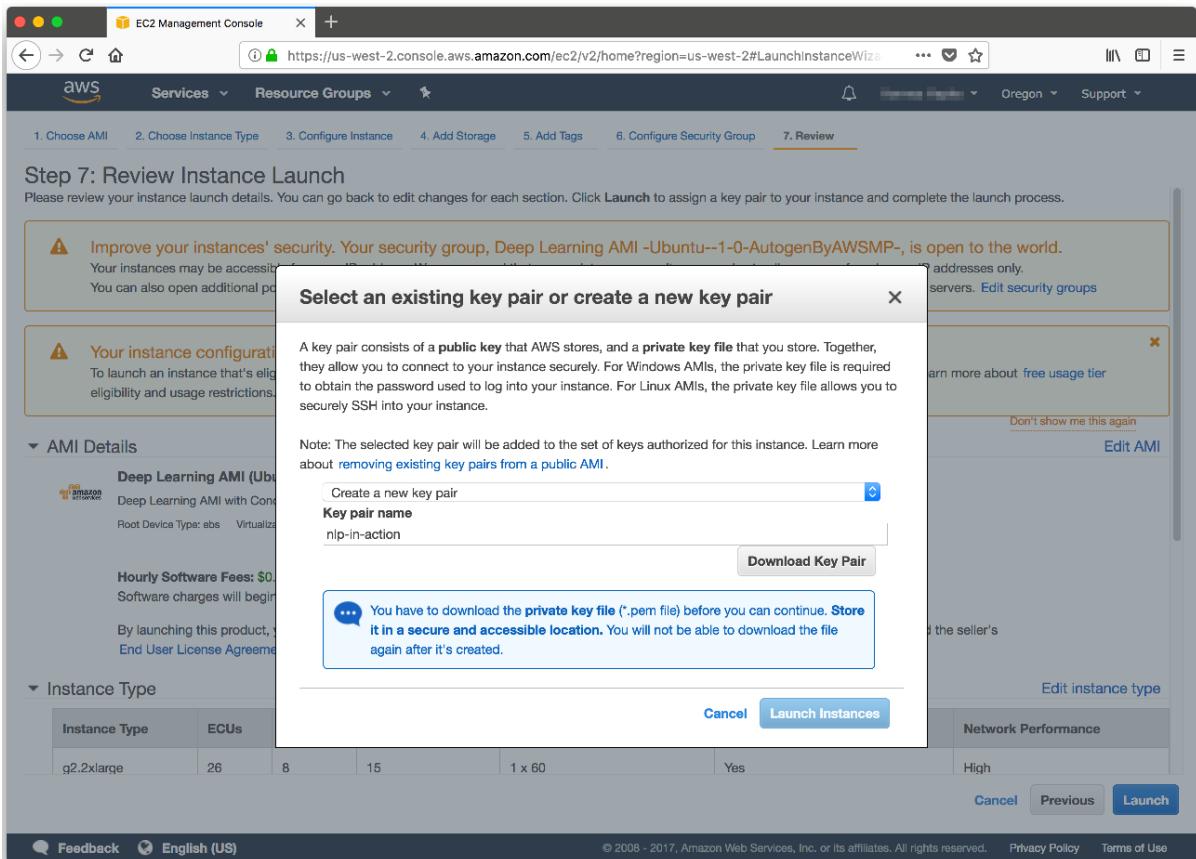


Figure G.9 Create a new or download an existing instance key

After saving your key pair (if you created a new key pair), AWS confirms that the instance is launched. On rare occasions, no instance is available, and you'll receive an error.

17. Click the instance hash that starts with *i-....*. The link sends you to the overview of all your EC2 instances, where you will see your instance mark as running or initializing.

Your instances are now launching
The following instance launches have been initiated: i-0e6dab2ec68602fe9 [View launch log](#)

Get notified of estimated charges
Create billing alerts to get an email notification when estimated charges on your AWS bill exceed an amount you define (for example, if you exceed the free usage tier).

How to connect to your instances
Your instances are launching, and it may take a few minutes until they are in the **running** state, when they will be ready for you to use. Usage hours on your new instances will start immediately and continue to accrue until you stop or terminate your instances.

Click [View Instances](#) to monitor your instances' status. Once your instances are in the **running** state, you can [connect](#) to them from the Instances screen. [Find out](#) how to connect to your instances.

Getting started with your software
To get started with Deep Learning AMI (Ubuntu) To manage your software subscription

[View Usage Instructions](#) [Open Your Software on AWS Marketplace](#)

Here are some helpful resources to get you started

- [How to connect to your Linux instance](#)
- [Amazon EC2: User Guide](#)
- [Learn about AWS Free Usage Tier](#)
- [Amazon EC2: Discussion Forum](#)

While your instances are launching you can also

Create status check alarms to be notified when these instances fail status checks. (Additional charges may apply)

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Figure G.10 AWS launch confirmation

| Name | Instance ID | Instance Type | Availability Zone | Instance State | Status Checks | Alarm Status | Public DNS (IPv4) |
|---------------------|---------------------|---------------|-------------------|----------------|---------------|--------------|--------------------|
| i-0e6dab2ec68602fe9 | i-0e6dab2ec68602fe9 | g2.2xlarge | us-west-2b | running | Initializing | Loading... | ec2-34-214-168-124 |

Instance: i-0e6dab2ec68602fe9 Public DNS: ec2-34-214-168-124.us-west-2.compute.amazonaws.com

| Description | Status Checks | Monitoring | Tags |
|----------------------------------|---|--------------------------------|--|
| Instance ID: i-0e6dab2ec68602fe9 | Public DNS (IPv4): ec2-34-214-168-124.us-west-2.compute.amazonaws.com | IPv4 Public IP: 34.214.168.124 | Private DNS: ip-172-31-26-225.us-west-2.compute.internal |
| Instance state: running | IPv6 IPs: - | | |
| Instance type: g2.2xlarge | Elastic IPs: | | |
| Elastic IPs: | | | |

Feedback English (US) © 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use

Figure G.11 EC2 Dashboard showing the newly created instance

18. Before logging into the AWS instance, ssh requires that the permissions for the private key are limited to you and no other user on your system. You can restrict the permissions by executing the following code ⁴⁸

Footnote 48 cygwin is required on any Windows system before using ssh.

```
$ chown -R $USER:users ~/.ssh/
$ chmod -R 600 ~/.ssh/nlp-in-action.pem
```

19. After you have changed the file permissions, execute the following:

```
$ ssh -i ~/.ssh/nlp-in-action.pem ubuntu@INSTANCE_PUBLIC_IP
```

20. If the Amazon Machine Image is Ubuntu-based, the username is ubuntu. Another AMI might require different login credentials.
21. If you log in for the very first time, you're warned that the fingerprint of the machine is unknown. Confirm with yes to go ahead with the login process.

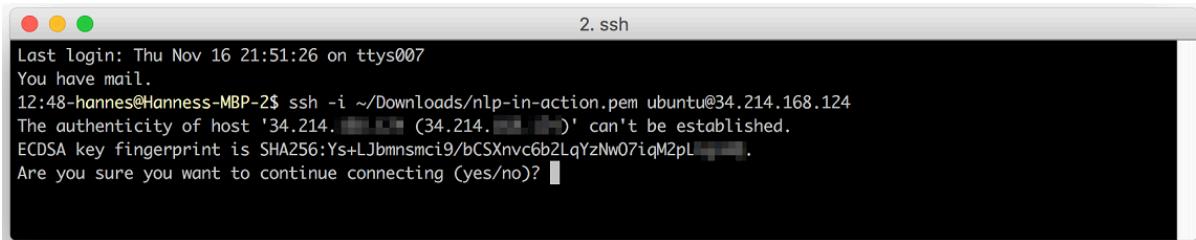


Figure G.12 Confirmation request to exchange ssh credentials

22. For a successful login, you see a welcome screen.

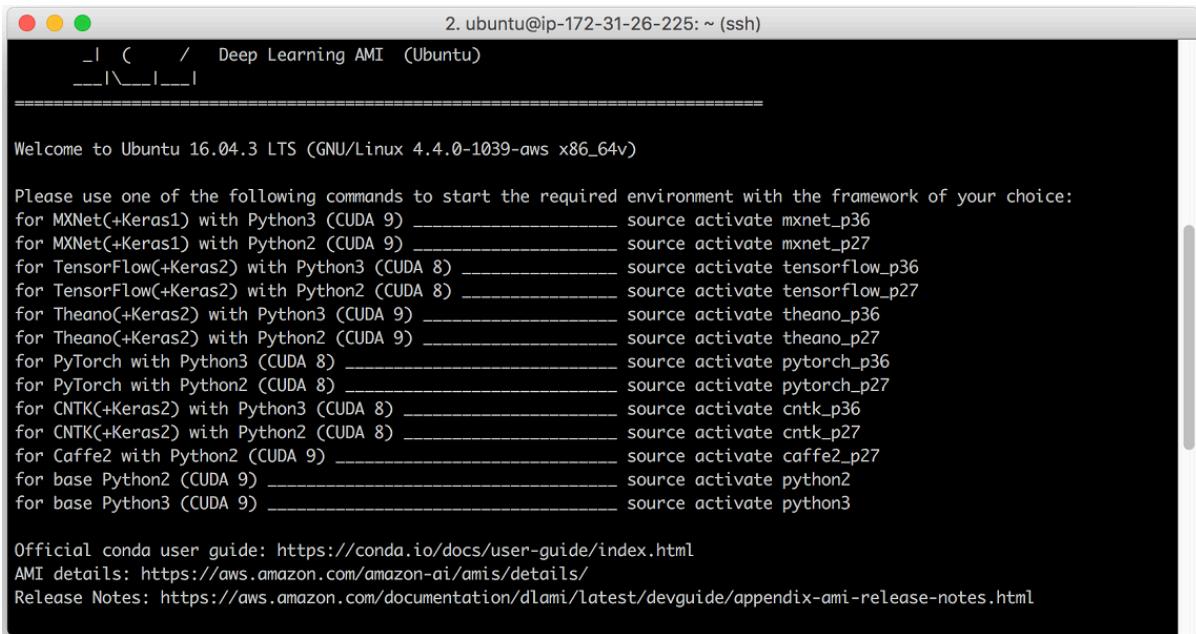


Figure G.13 Welcome screen after a successful login

23. As the final step, you need to activate your preferred development environment. The

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.

machine image provides various environments, including PyTorch, TensorFlow, and CNTK. Since we use TensorFlow and Keras in this book, you should activate the tensorflow_p36 environment. This loads a virtual environment with Python 3.6, Keras, and TensorFlow installed.

```
$ source activate tensorflow_p36
```

```
2. ubuntu@ip-172-31-26-225: ~ (ssh)

Official conda user guide: https://conda.io/docs/user-guide/index.html
AMI details: https://aws.amazon.com/amazon-ai/amis/details/
Release Notes: https://aws.amazon.com/documentation/dlami/latest/devguide/appendix-ami-release-notes.html

* Documentation: https://help.ubuntu.com
* Management: https://Landscape.canonical.com
* Support: https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud

56 packages can be updated.
31 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

ubuntu@ip-172-31-[REDACTED]:~$ source activate tensorflow_p36
```

Figure G.14 Activating your pre-installed Keras environment

Now that you've activated your TensorFlow environment, you are ready to train your deep learning NLP models. Head over to an *iPython* shell with

```
$ ipython
```

and train your models.

Have fun!

G.1.1 Cost control

Running GPU instance can quickly get expensive. The smallest GPU instance in the US-West 2 region costs \$0.65 per hour at the time of this writing. Training a simple sequence-to-sequence model can take a few hours and then you might want to iterate on your model parameters. All iterations can quickly add up to decent monthly bill. To avoid any surprises, we recommend a few actions:

- Turn off idle GPU machines. When you stop (not terminate) your machine, the last state of the storage (except your /tmp folder) will be preserved and you can return to it. In-memory data will be lost, so make sure to save all your models before stopping the machine.
- Check your AWS billing summary regularly. You will quickly see if there are any

instances running.

- Create an AWS Budget and set spending alarms. Once you have configured a budget, AWS will alert you when you are exceeding it.

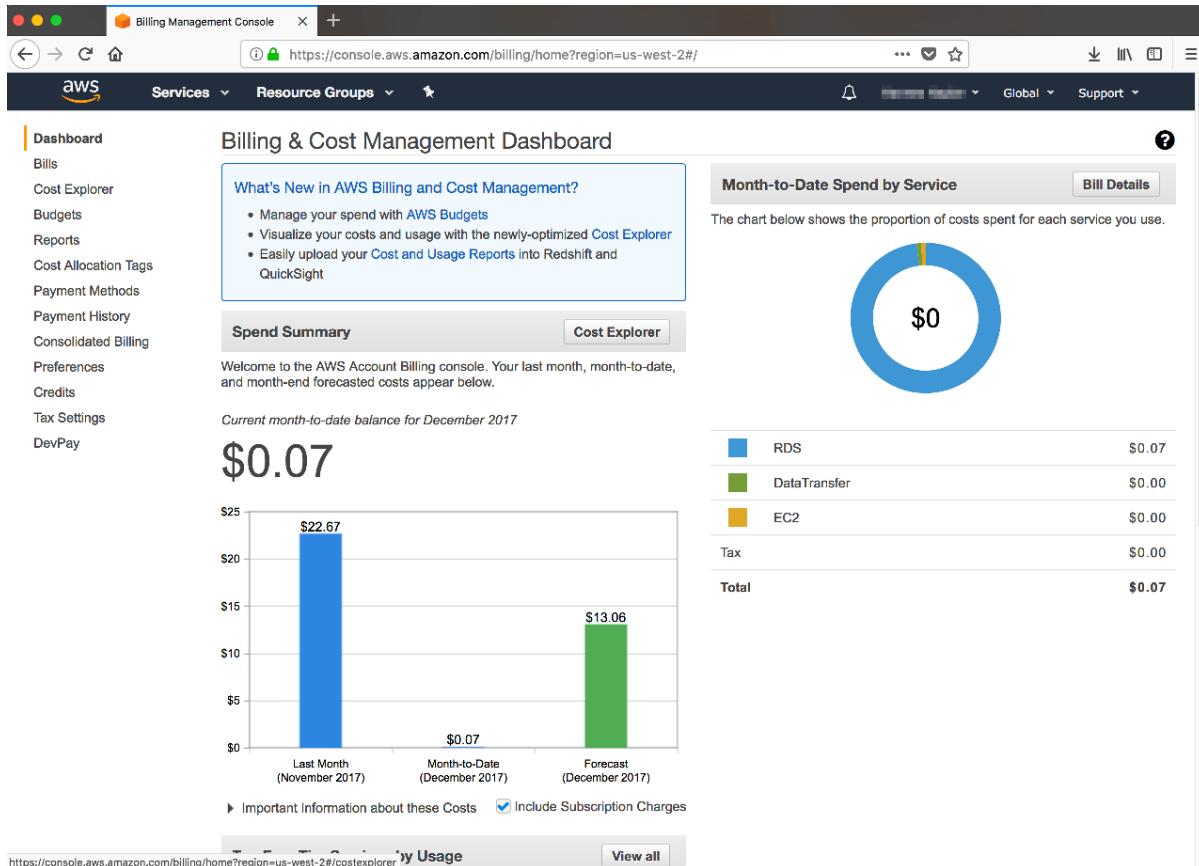


Figure G.15 AWS Billing Dashboard

The screenshot shows the AWS Billing Management Console with the URL <https://console.aws.amazon.com/billing/home?region=us-west-2#/budgets>. The left sidebar has a 'Budgets' section selected. The main content area is titled 'AWS Budgets' and contains a brief description: 'AWS Budgets lets you quickly create custom budgets that will automatically alert you when your AWS costs or usage exceed, or are forecasted to exceed, the thresholds you set.' Below this is a 'Create budget' button. The next section, 'Getting started with AWS Budgets', features three icons: a database icon for 'Create and manage budgets', a person icon for 'Refine your budget using filters', and a monitor icon for 'Add notifications to a budget'. A note at the bottom says: 'For more information about managing your AWS Budgets, refer to [Managing Your Costs With Budgets](#) in the Billing & Cost Management User Guide.'

Figure G.16 AWS Budget Console

Locality sensitive hashing



In chapter 4 you learned how to create topic vectors with hundreds of dimensions of real-valued (floating point) numbers. In chapter 6 you learned how to create word vectors that have hundreds of dimensions. Even though you can do useful math operations on these vectors, you cannot quickly search them like you can discrete vectors or strings. Databases do not have efficient indexing schemes for vectors of more than four dimensions.⁴⁹ To use word vectors and document topic vectors efficiently, you need a search index that can help find the nearest neighbors for any given vector.

Footnote 49 Some advanced databases such as PostgreSQL can index higher-dimensional vectors, but efficiency drops quickly with dimensionality.

You need this to convert the results of vector math into a word or set of words (because the resultant vector is never an exact match for the vector of a word in the English language). You also need it to do semantic search. This appendix shows an example approach based on locality sensitive hashing (LSH).

H.1 High-dimensional vectors are different

As you add dimensions to vectors going from 1D to 2D and even 3D, nothing much changes about the kinds of math you can do to find nearest neighbors quickly. Let's talk a little bit about conventional approaches used by database indexes for 2D vectors and work your way up to high-dimensional vectors. That will help you see where the math breaks down (becomes impractical).

Indexes are designed to make looking up something easy. Real value (`float`) indexes for things like latitude and longitude can't rely on exact matches, like the index of words at the back of a textbook. For 2D real-valued data, most indexes use some sort of bounding box to divide up a low-dimensional space into manageable chunks. The most common

example of an index like this for two-dimensional geographic location information is Zip codes. You assign a number to each region in 2D space; even though these are not technically square bounding boxes, they serve the same purpose.⁵⁰ This is equivalent to locality sensitive hashing, a way to produce these numbers so that hashes that are close to each other in value are close to each other in the 2D grid locations as well.

Footnote 50 And the military does use bounding boxes with its grid system for dividing up the globe into rectangular bounding boxes and assigning each one a number.

Let's start with 1D. Imagine a uniform distribution of 1D vectors, a single scalar value for each vector. You can create a 1D bounding box that is half the width of the overall space by finding a cut point somewhere near the middle of the range of your values in your 1D vectors. Each side (bounding box) will have about half the number of vectors in it as in the entire space. You can then cut that box in half to narrow it down further and create a binary tree of bounding boxes to help you index the vectors, assigning each vector to a "zip code" or index.

You want your final "zip code" to be shared by only a few other vectors so you don't have to search very far for the nearest vector to any new query vector you come up with. So you can keep doing that cut-in-half until you have only a few vectors in each box, and those boxes can then be assigned numbers and act as your index. For 1D space, the average number of points in each box is $(\text{num_vectors} / 2)^{\star\star} \text{ num_boxes}$. For 1D space, you only need about 16 levels (box sizes) to your boxes to index millions of points so that there are only a few thousand points in each box. It's easy to loop through a few thousand vectors with brute force to search for matches.

Now let's add a dimension. Think about how you'd divide the space into rectangular regions. Which dimension would you cut in half each time you tried to reduce the number points by half? Fortunately, in two dimensions it doesn't matter, you can cut a space in half along either axis and you'll likely get about half the number of points on each size. So you just alternate dimensions, dividing in half, just like before. And that works for three dimensions and even all the way out to about 10 dimensions before things start to get weird.

At around 12 dimensions, when you divide a space in half along an arbitrary dimension, you'll often end up with very few points (often none) on one side and a lot on the other.

Adding dimensions to a vector turns out to change the math that is possible. Unlike 2D and 3D vectors, it's not possible to truly "index" or "hash" topic vectors in a way that allows you to retrieve the closest matches.

In high-dimensional space, conventional indexes that rely on bounding boxes fail. And even an advanced form of hashing, called locality sensitive hashing also fails. LSH usually allows for similar vectors to have similar hashes. These locality sensitive hashes are like Zip codes that define more and more precise locations the more digits you add to the Zip code. But let's see where an LSH breaks down. In figure H.1, we constructed 400,000 completely random vectors, each with 200 dimensions (typical for topic vectors for a large corpus). And we've indexed them with the Python LSHash package (`pip install lshash3`). Now imagine that you have a search engine that wants to find all the topic vectors that are close to a "query" topic vector. How many will be gathered up by the locality sensitive hash? And for what number of dimensions for the topic vectors do your search results cease to make much sense at all?

| D | N | 100th Cosine | | Top 10 | | Top 100 | |
|----|-------|--------------|---------|---------------|---------------|---------|---------|
| | | Distance | Correct | Top 1 Correct | Top 2 Correct | Correct | Correct |
| 2 | 4254 | 0 | TRUE | TRUE | TRUE | TRUE | TRUE |
| 3 | 7727 | 0.0003 | TRUE | TRUE | TRUE | TRUE | TRUE |
| 4 | 12198 | 0.0028 | TRUE | TRUE | TRUE | TRUE | TRUE |
| 5 | 9920 | 0.0143 | TRUE | TRUE | TRUE | TRUE | TRUE |
| 6 | 11310 | 0.0166 | TRUE | TRUE | TRUE | TRUE | TRUE |
| 7 | 12002 | 0.0246 | TRUE | TRUE | TRUE | TRUE | FALSE |
| 8 | 11859 | 0.0334 | TRUE | TRUE | TRUE | TRUE | FALSE |
| 9 | 6958 | 0.0378 | TRUE | TRUE | TRUE | TRUE | FALSE |
| 10 | 5196 | 0.0513 | TRUE | TRUE | TRUE | FALSE | FALSE |
| 11 | 3019 | 0.0695 | TRUE | TRUE | TRUE | TRUE | FALSE |
| 12 | 12263 | 0.0606 | TRUE | TRUE | TRUE | FALSE | FALSE |
| 13 | 1562 | 0.0871 | TRUE | TRUE | TRUE | FALSE | FALSE |
| 14 | 733 | 0.1379 | TRUE | FALSE | TRUE | FALSE | FALSE |
| 15 | 6350 | 0.1375 | TRUE | TRUE | TRUE | FALSE | FALSE |
| 16 | 10980 | 0.0942 | TRUE | TRUE | TRUE | FALSE | FALSE |

Figure H.1 Semantic search with LSHash

You can't get many search results correct once the number of dimensions gets significantly above 10 or so. If you'd like to play with this yourself, or try your hand at building a better LSH algorithm, the code for running experiments like this is available in the `nlpia` package. And the `lshash3` package is open source, with only about 100 lines of code at the heart of it.

The state of the art for finding the closest matches for high-dimensional vectors is Facebook's FAISS package and Spotify's Annoy package. Because Annoy is so easy to

install and use, that's what we chose to use for your chatbot. In addition to it being the workhorse for finding matches among vectors representing song metadata for music fans, Dark Horse Comics has also used annoy to suggest comic books efficiently. We talk more about these tools in chapter 13.

H.2 "Like" prediction

Figure H.2 is what a collection of tweets looks like in hyperspace. Or, more accurately, these are the 2D shadows of 100-D tweet topic vectors (points). The green marks represent tweets that were liked at least once; the red marks are for tweets that received zero likes.

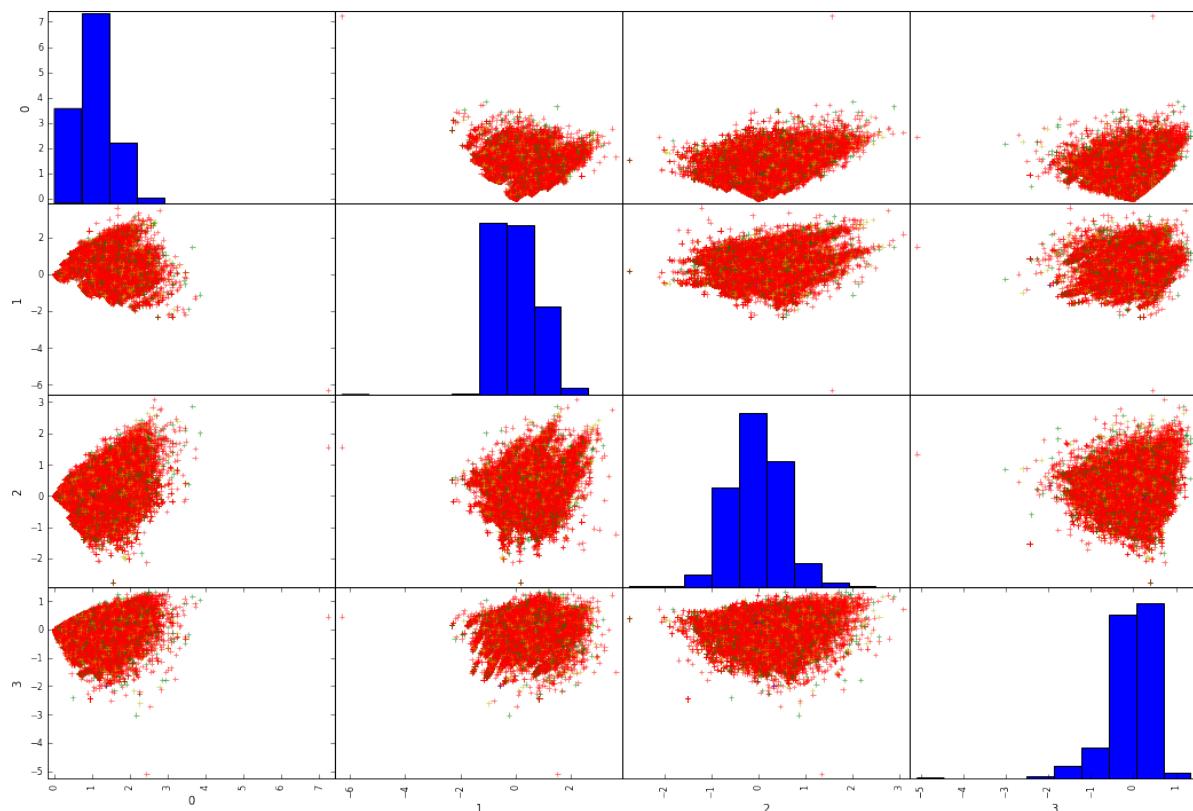


Figure H.2 Scatter matrix of four topics for tweets

An LDA model fit to these topic vectors will succeed 80% of the time. However, like your SMS dataset, your tweet dataset is also very imbalanced. So predicting the likeability of new tweets using this model is not likely to be very accurate. For now, since LSA, LDA, and LDIA are your main language modeling tools, you should probably stick to semantic matching and more straightforward classification problems, such as spam detection.