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 4

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

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.

1 The Language of Thought

In this chapter you'll learn:

- What Natural Language Processing is
- Some of the magical things NLP can do
- 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're about to embark on an exciting adventure in Natural Language Processing (NLP) In this chapter we'll explain what NLP is and then show you what it can do by describing some exciting, almost magical, NLP applications. Then we'll show you how to begin training a computer to both read and write natural language statements. Throughout the rest of the book you will add to your NLP toolbox incrementally while building a chatbot that can carry on a dialog in English and answer natural language questions.

1.1 The Magic

So what's so magical about reading and writing natural languages? Machines have been processing languages since their inception. However, these were "formal" languages, programming languages designed to be interpreted (or compiled) only one correct way. Ada, COBOL, and Fortran were some of the earliest languages. Today Wikipedia lists more than 700 programming languages. In contrast, Ethnologue, a webbased publciation of statistics about natural languages, has identified around 7000 distinct natural languages spoken by humans around the world. These are called Natural Languages in the same way that the natural world of evolved things is different from the mechanical world of things designed and built by humans. We're going to focus on only one natural language, English. And we're going to show you how to process and generate that language with only one programming language, Python. Python was designed from the ground up to be as readable a language as possible. It also exposes a lot of it's own language processing "guts." Both of these characteristics make it a natural choice for learning natural language processing and implementing production quality NLP algorithms. And we'll even use Python in lieu of the "universal language" of mathematics and mathematical symbols, wherever possible. After all it's unambiguous and designed to be more readable by programmers like you.

There are no compilers or interpreters for natural languages, in the traditional computer science sense. Natural languages aren't intended to be translated into a finite set of mathematical operations, like programming languages are. But this book is going to show you how to write software in Python that can extract information from English and "interpret" or translate that information into a form that 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 will be the function of the "dialog engine" (chatbot) that we'll help you build in Chapters 4-6. We will focus entirely on English text documents, not spoken statements. Processing spoken statements is call "speech recognition" and "speech generation."

Just extracting useful information from text is quite difficult, and machines cannot perform the task as accurately and reliably as humans. But like many other technical problems, it seems a lot easier once you know the answer. And the techniques you will learn are powerful enough to create machines that can surpass humans in both accuracy and speed for some suprisingly subtle tasks, like recognizing sarcasm in an isolated natural language statement. 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 we'll help you incorporate context (metadata) into your NLP pipeline.

Once we are able to extract structured numerical data, vectors, from natural language, we can take advantage of all the tools of mathematics and machine learning. We will 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. It's these breakthough ideas that opened up a world of "semantic" analysis, allowing computers to interpret and store the "meaning" of statements rather than just counts of words or characters. 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 and formal languages. We can rely on statistical relationships between words instead of a deep system of grammar rules.[1] Imagine if you had to define English grammar and spelling rules in a nested tree of "if…then" statements, to deal with every possible way that words, letters, and punctuation can be combined to make a statement. And this wouldn't even begin to capture the semantics, the meaning of English statements. And imagine how fragile, brittle, this software would be. Unanticipated spelling or punctuation would break it.

Footnote 1 Systems of grammar rules can be implemented in a Computer Science abstraction called a "finite state machine." Regular grammars can be implemented in regular expressions, for which there are interpreters and compilers in Python. These can be thought of as trees of "if…then…else" statements covering every possible action or statement by a machine.

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 all the things that make up a morning including not only that mornings come before "noon"s and "afternoon"s and "evening"s but also that they can represent times of day as well as general experiences of a period of time. And the interpreter is assumed to know that "good morning" is a common greeting that actually doesn't contain much information at all about the morning, but rather the state of mind of the speaker and her readiness to speak with others. This turns out to be a powerful assumption, allowing us to say a lot with very few words. It's a degree of "compression" that machines have yet to master. However, we will show you techniques in later chapters to help machines build "ontologies" or knowledge bases of common sense knowledge to help them interpret our statements.

1.2 Practical Applications

Natural Language Processing (NLP) is everywhere. It's so ubiquitous that some of the examples here may surprise you.

Search	Web	Documents	Autocomplete
Dialog	Chatbot	Assistant	Scheduling
Editing	Spelling	Grammar	Style
Writing	Index	Concordance	Table of Contents
Email	Spam filter	Classification	Prioritization
Text Mining	Summarization	Knowledge extraction	Medical diagnoses
Law	Legal inference	Precedent search	
News	Event detection	Fact checking	Article writing
Attribution	Detect plagerism	Literary forensics	Style coaching
Sentiment Analysis	Monitor community morale	Product review triage	
Predict Behavior	Finance	Election	Sales
Creative writing	Movie scripts	Poetry	

To provide meaningful results quickly, a search engine must index webpages or document archives in a way that takes into account the meaning and intent 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, even style coaches. NLP bots at the Associated Press write entire financial news articles and sporting event reports.[2] There are bots that compose weather forecasts that sound a lot like what your hometown weather person might say, perhaps because meteorologists use applications to draft a script just before the show. Movie and product reviews are written not only by Amazon's "Mechanical Turk" systems that harness human brain power behind the scenes, but also by real live NLP systems ;). There are chatbots on Twitter, Slack, IRC, and even customer service websites. It's estimated that nearly 20% of the tweets about the 2016 US presidential election were composed by chatbots.[3] automated-pro-trump-bots-overwhelmed-pro-clinton-messages-researchers-say.html], amplifying the viewpoint of the owners and developers of those bots. NLP systems sometimes answer the phone for companies that prefer to be on the cutting edge. There are NLP systems that can act as e-mail "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), interracting with others by e-mail on their boss's behalf. Companies are putting the very brand and "face" of their company 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 a few paragraphs in their books. More on that later.

Footnote 2 "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

Footnote 3 New York Times, Oct 18, 2016, www.nytimes.com/2016/11/18/technology/

Recent advances in NLP even helped solve the long-standing mystery of the identity of Shakespeare and the authorship of famous works like *Henry the V*. Hint: Christopher Marlowe was more than just a friend and editor.

The pace of development in NLP is accelerating. NLP powers exciting new industries like election and financial market forecasting based on the firehose of natural language and unstructured data emanating from and about those markets. It's unnerving to realize that some of the articles whose sentiment is driving those predictions are being written by other bots and that these bots are unaware of each other. In 2014, Banjo developed a twitter monitoring service that was predicting breaking news events 30 minutes to an hour before the first Reuters or CNN reporter filed a story or snapped a photograph. The tweets it was using to detect those events would have almost certainly been favorited and retweeted by several other bots in order to catch the "eye" of Banjo's NLP bot. And more and more of the tweets it monitored weren't just curated, promoted, or metered out according to machine learning algorithms driven by analytics, but many were written, wholecloth, by NLP engines.

NLP is used to compose entire movie scripts.[4] And one can imagine that video games might play a role as props in those movies. And the "play within a play" (a movie about video game dialog by AI bots) could have been scripted by an AI which would have NLP capabilities to model NLP system capabilities. Humans may produce and edit those movies, with the help of NLP systems, but then the bots step back in to digest the subtitles and transcripts and write reviews to help us consumers decide which movies to watch. In later chapters we'll show you how this is done, how bots can be taught to generate original dialog from a corpus of movie scripts or recommend movies for you based on nothing more than the natural language you put in your profile page.

Footnote 4 Five Thirty Eight, fivethirtyeight.com/features/some-like-it-bot/

More and more entertainment, advertisement, and financial reporting content generation can happen without requiring a human to lift a finger… except to press the "on" button on their computer or e-book reader to consume and enjoy that content.

NLP enables efficient information retrieval (search). Search was the first commercially successful application of NLP. Search powered faster and faster development of NLP algorithms which then improved search technology itself. We'll help you contribute to this virtuous cycle of increasing collective brain power by showing you some of the natural language indexing and prediction techniques behind web search. We'll show you how to index this book so that you can free your brain to do higher level thinking, abstracting, generalizing, and allow the machines to take care of memorizing the terminology, facts, and Python snippets discussed here.

The development of NLP systems has built to a crescendo of information flow and computation through and among human brains. We can now type only a few characters into a search bar, and often retrieve the exact piece of information we need to complete whatever task we're working on, like writing the software for a textbook on NLP. The top few autocomplete options are often so uncannily appropriate that we feel like we have a human assisting us with our search. We authors used various search engines

throughout the writing of this textbook. Search engines found Tweets and articles curated or written by bots which in turn inspired many of the NLP explanations and applications in the following pages.

1.3 What is Driving NLP Advances?

- A new appreciation for the ever-widening web of unstructured data?
- Increases in processing power catching up with researchers' ideas?
- The efficiency of interacting with a machine in our own language?

Of course it's all of the above. We won't bore you by reiterating the reasoning behind these three "causes" for the recent resurgence of AI and NLP. You can easily enter the question "Why is natural language processing so important right now?" into any search engine that does NLP well,[5] and you're likely to be directed to a wikpedia article full of good reasons.[6] And these articles will have even more recent data and examples than we can provide in a printed book.

Footnote 5 Duck Duck Go query about NLP, duckduckgo.com/?q=Why+is+natural+language+processing+so+important+right+now%3F&t=h_&ia=web

Footnote 6 Wikipedia/NLP, en.wikipedia.org/wiki/Natural_language_processing

However, we can offer some deeper reasons gleaned from related research in other areas. Reasons that even Wikipedia authors and bots may not "think" of. One such reason is the concept that our minds think more efficiently due to our ability to collect concepts, thoughts, ideas, into packets of meaning, packets that include sentences, words, and meaningful parts of words (called morphemes). One of the ideas in Steven Pinker's *The_Stuff_of_Thought* is that we actually think in natural language.[7] It's not called an "inner dialiog" without a reason. Facebook, Google, and Elon Musk are betting on the fact that words will be the default communication protocol for thought with their attempts to translate thought (brain waves and electrical signals) into words.[Facebook and Google research project to translate electrical signals from skin into text: backchannel.com/we-are-entering-the-era-of-the-brain-machine-interface-75a3a1a37fd3]

Footnote 7 Steven Pinker, en.wikipedia.org/wiki/The_Stuff_of_Thought

Geofry Hinton other deep learning deep thinkers have demonstrated that even individual characters, completely isolated from other characters, carry some meaning. So if it's good enough for human brains, and we'd like to emulate or simulate human thought in a machine, then natural language is likely to be a critical communication protocol within and between such systems. In additionAnd there may be important clues to intelligence hidden in the data structures and nested networks of connections between words that you're going to learn about in this book. After all, these structures and algorithms make it possible for an inanimate system to digest, store, retrieve, and generate natural language in ways that sometimes appear to be human.

And there's another even more important reason why you might want to learn how to program a system that uses natural language well… you might just save the world.

Hopefully you've been following the discussion among movers and shakers about the *AI Control Problem* and the challenge of developing "Friendly AI".[8] Nick Bostrom,[9], Calum Chace,[10], Elon Musk,[11] and many others believe that the future of humanity rests on our ability to develop friendly machines. And natural language is going to be the main social connections between humans and machines for the foreseeable future—at least until direct machine-brain interfaces become widespread.

Footnote 8 Wikipedia, AI Control Problem, en.wikipedia.org/wiki/AI_control_problem

Footnote 9 Nick Bostrom, home page, nickbostrom.com/Nick Bostrom

Footnote 10 Calum Chace, *Surviving AI*, www.singularityweblog.com/calum-chace-on-surviving-ai/

Footnote 11 www.forbes.com/sites/ericmack/2015/01/15/elon-musk-puts-down-10-million-to-fight-skynet/#17f7ee7b4bd0

But even once we are able to "think" directly to/with machines, those thoughts will likely be shaped by natural words and languages within our brains as well as the language of computing machines. The line betwen natural and machine language will be blurred just as the separation between man and machine fades. In fact this line began to blur in 1984,[12] making George Orwell's dystopian predictions both more likely and easier for us to accept.[13][14]

Footnote 12 Haraway, *Cyborg Manifesto*, en.wikipedia.org/wiki/A_Cyborg_Manifesto

Footnote 13 Wikipedia on George Orwell's *1984*, en.wikipedia.org/wiki/Nineteen_Eighty-Four

Footnote 14 Wikipedia, The Year 1984, en.wikipedia.org/wiki/1984

Hopefully the phrase "help save the human race" didn't leave you incredulous. As you progress through this book, we'll show you how to build and connect several lobes of a chatbot "brain" together. As you do this you'll notice that very small nudges to the social feedback loops between humans and machines can have a profound effect, both on the machines and on humans. Like a butterfly flapping its wings in China, one small decimal place adjustment to your chatbot's "selfishness" gain can result in a chaotic storm of antagonistic chatbot behavior and conflict.[15] And you'll also notice how a few kind, altruistic systems will quickly gather a loyal following of supporters that help quell the chaos wreaked by bots designed to pursue shortsighted goals (greedy "objective functions" in the optimization and machine learning world). Prosocial, cooperative chatbots have an outsized impact on the world, because of the network effect of prosocial behavior.

Footnote 15 A chatbot's main tool is to mimic the humans it is conversing with. So dialog participants can use that influence to engender prosocial and antisocial behavior in bots: www.techrepublic.com/article/why-microsofts-tay-ai-bot-went-wrong/

NOTE An example of autonomous machines "infecting" humans with their considerate, measured behavior can be found in studies of the impact self-driving cars are likely to have on rush-hour traffic.[16] In some studies, if as few as 1 in 10 vehicles around you on the freeway will help moderate human behavior reducing congestion and producing smoother, safer traffic flow.

Footnote 16 www.enotrans.org/wp-content/uploads/AV-paper.pdf

This is how and why the authors of this book came together. A supportive community emerged through open, honest, prosocial communication over the Internet using the language that came natural to us. And we're using our collective intelligence to help build and support other intelligent actors (machines).[17] And we hope that our words will leave their impression in your mind and propogate like a virus through the world of chatbots, infecting others with our passion for building prosocial NLP systems. And not only can prosocial chatbots help us treat each other more humanely, they may influence the character of any superintelligence that gains its knowledge by reading natural language from the web. The superintelligence may be inspired by the effectiveness of our altruistic bots to prioritize altruisic acts of its own. This could turn out to be a workaround for the control problem, obviating our need to control AI at all. At the very least it represents a small contribution to the altruistic side of a balance of natural selection forces that, if left unchecked, can sometimes pushes organisms toward socialpathic intelligence.

Footnote 17 Toby Sagaram, *Collective Intelligence*, forum.myquant.cn/uploads/default/original/1X/2065c4d1964e26331996cfa23d12acd185e3d7b6.pdf

1.4 Language through a Computer's "Eyes"

When we type "Good Morn'n Rosa", a computer sees only "01000111 01101111 01101111 …". How can we program our 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). A finite state machine that outputs a sequence of new symbols as it runs, like the Python `str.translate` function, is called a "finite state transducer" (FST). Don't worry if these terms are new to you. Things will become clear as we show you examples throughout this book. In fact you've probably already built a FSM without even knowing it. Have you ever written a regular expression? That's the kind of FSM that we'll use in the next section to show you one posssible approach to NLP, the grammar-based approach.

What if we decided to just 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 just one typo or

variation in the statement. Our bot would be sent "off the rails." And bits aren't very continuous or forgiving, they either match or don't. There's no obvious way to find similarity between two streams of bits that takes into account what they actually 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. Hopefully you'll never think of your combination bicycle lock the same way again, after finishing this chapter. 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 combination is any sequence of symbols that matches the "grammar" 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 the language of locks is to release the catch holding the U-shaped hasp so you can get into your locker.

And this "language of locks" is the kind of language, a particularly simple one, that we can use in a chatbot 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 interpretted. 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"[18] language of combination locks.

Footnote 18 one of Cialdini's six psychology principles in his best-selling book *Influence*
changingminds.org/techniques/general/cialdini/click-whirr.htm

---

We're going to use a slightly more restrictive grammar than formal grammar, called a "regular grammar." That grammar is particularly easy to work with. This small restriction in our langauge unlocks a broad set of powerful, easy-of-use features. Regular grammars have predicatable, provable behavior, and yet are flexible enough to power some of the most sophisticated dialog engines and chatbots on the market, like Amazon Echo and Google Now. Extremely deep, complex regular grammar rules can often be expressed in a single line of code called a "regular expression." There are very 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 interaction with users.

NOTE Actually, regular expressions implemented in Python and in Posix (Unix) applications like `grep`, are not true regular grammars. They have language and logic features like "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. *R*gular *E*xpressions are indeed used mostly for search, for sequence matching. But it turns out that anything that can do search, 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 knowlege base about the user or about the world that 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" (FSM) or "deterministic finite automaton" (DFA). FSMs will 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. For now, just think of them as combination locks. For those who can't resist trying to understand a bit more about these computer science tools, here's a diagram from Wikpedia that 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.

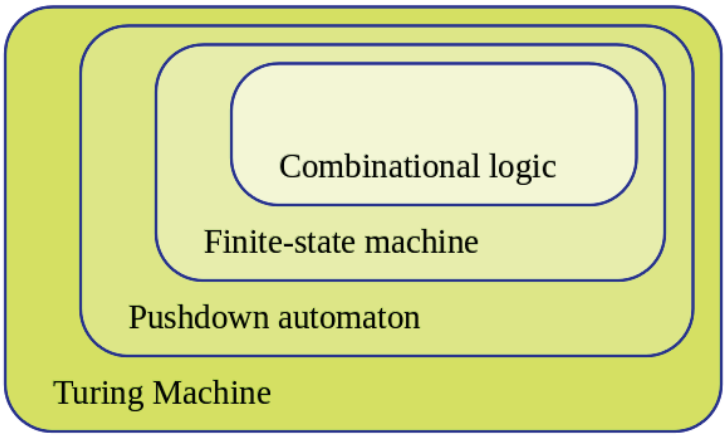


Figure 1.1 Kinds of Automata

1.4.2 A Simple Chatbot

Let's build a FSM, a regular expression in Python, that can "speak" lock language (regular language). We could program it to understand lock language statements, like '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 antisocial, 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 protocall we would just define a simple "handshake" with an ACK/NAK signal after each message passed back and forth between two machines. But our machiness are going to be interacting with humans who say things like "Good morning, Rosa" rather than sending out of bunch of chirps and beeps to synch up a modem at the start of a conversation. Here's one way to recognize several different human greetings at the start of a conversation "handshake."

```
>>> import re
>>> re_greeting = re.compile(r"[^a-z]*([y]o|[h']?ello|ok|hey|(good[ ])?(morn[gin']{0,3}|
    afternoon|even[gin']{0,3}))[\s,;:]{1,3}([a-z]{1,20})", flags=re.IGNORECASE)
>>> 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 Norning 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'>
```

❶

❶ Notice that this regex cannot recognize (match) words with typos

There's a lot of logic packed into that first line of code. It gets the the job done for a surprsising range of greetings. But it missed that "Norning" typo. This 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 very unlikely to ever say. That's called a "False Positive," which is also a bad thing. This means that our regular expression is both too liberal and too strict. We'd have to do a lot more work to refine the phrases that it matches to be more humanlike. And this tedious work would be highly unlikely to ever succeed at capturing all the slang and mispellings that poeple commonly say. These mistakes could make our bot sound a bit dull, mechanical. 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).

But let's go ahead and finish up our one-trick chatbot by adding an output generator. It needs to say something.

```
>>> my_names = set(['rosa', 'rose', 'chatty', 'chatbot', 'bot', 'chatterbot'])
>>> curt_names = set(['hal', 'you', 'u'])
```

```
>>> greeter_name = None  # we don't yet know who is chatting to the bot
...
>>> 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 or '')
```

So if you run this little script and "chat" to our bot something like "Hello Rosa", it will respond by asking about your day. If you use a rudish name to address the chatbot, she will be less responsive, but not inflamatory, to try to encourage politeness.[19](en.wikipedia.org/wiki/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] 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 noone else out there watching our `input()` line, but if this were a function within a larger chatbot, you want to deal will these sorts of things.

Footnote 19 The idea for this defusing response originated with Viktor Frankl's *Man's Search for Meaning*, his [Logotherapy

Because of the limitations of computational resources, early NLP researchers had to use their human computational power to design and hand-tune complex logical rules to extract information from a natural langauge string or respond to it with another string. This is called a "grammar-based" approach to NLP. The core NLP building blocks like stemmers and tokenizers as well as end-to-end NLP systems like Liza were built this way, from regular expressions. You'll learn more about grammar-based approaches to tokenizing and stemming in Chapter 2, because they still form the foundation of modern NLP. Perhaps you've heard of the "Porter Stemmer" or the "Treebank Tokenizer" which we'll show in Chapter 2. But in later chapters we're going to take advantage of the exponentially greater computational resources of the 21st century, 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 the Regular Expresion Appendix or the online documentation for Python regular expressions. But you don't have to 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 and 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 grammar-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 that 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 don't. There's no obvious way to find similarity between two streams of bits that takes into account what they actually signify or mean. The bits and character sequences for "Good" will be just as similar to "Bad!" as they are to "Okay".

Sequences of characters with similar meaning aren't "close" to one other when we use metrics designed to measure distances for numerical sequences. Metrics like Jaccard, Levenshtein, and Euclidian distance can sometimes add enough fuzziness to prevent a chatbot from stumbling over minor spelling or typographical changes. But these metrics fail to capture the essence of the relationship between two strings of characters when they are very dissimilar. Though we'll use these distance metrics in later chapters, there's yet a better approach.

We're not going to stay in this confusing binary world of logic for long, but let's imagine we're 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 WWII. 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. Like Mavis Batey at Bletchley Park during WWII, 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 just 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 decypher 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 counting of characters and words is tedious work, but 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 them to bins and store them away in those bins as bit vectors like a coin or token sorter 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 that 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 very 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 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.

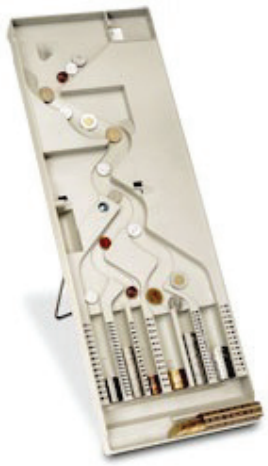


Figure 1.2 Canadian Coin Sorter

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.

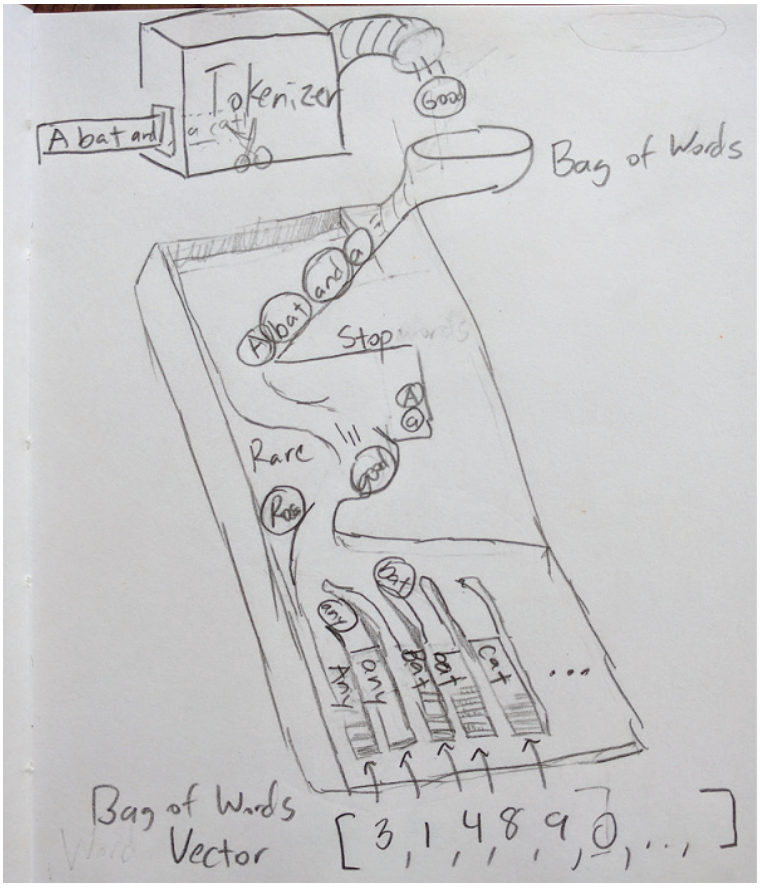


Figure 1.3 Token Sorting Tray

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 couple crude examples below, and then well show some more useful data structures for bag-of-word vectors in Chapter 2.

This is our first "vector space model" of a language. Those bins and the numbers they contain for each word are represented as very long vectors containing a lot of zeros and a few ones or twos scattered around wherevever the word for that bin actually 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 occuring within a collection of various sequennces 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 we might clean those tokens up. We'll do just 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'll 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 that we could find. We'd count up the tokens in each slot at the bottom after each of these statements were processed, and we'd call that a vector representation of that statement. All the possible vectors that 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. This 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-word 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 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, in some cases million-dimensional for a vocabulary of a million possible tokens. We'll discuss the curse of dimensionality in Chapter 5 and some other properties that makeand several For reasonsWhat can we do in this hyper-dimensional space?

1.5 A Brief Overflight of Hyperspace

In chapter 5 we'll show you how to "consolidate" words into a smaller number of vector dimensions to help reduce this curse of dimensionality and even turn it into a powerful tool. When we "project" these vectors onto each other to determine their distance from one another this will be a reflection of the similarity in their *meaning* rather than merely just thier statistical word usage. This vector distance metric is called "cosine distance metric" which we'll 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 2-D 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 vector space model of messages so we can label them with a set of continuous (float) values for qualities like subject matter and sentiment. How likely is this 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 them 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 of our questions.

These vectors become something that a machine can be programmed to react to. We can simplify and generalize 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"? Etc. 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. Actually it was more than just raw hardware power and capacity that 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.

Actually 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 of 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 upper 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 replaying an exact "piece" rather than composing new music or extracting the "essence" of a piece of music so we can compare the piano paper roll for one "song" to that of another. And this representation is

actually longer than the original ascii-encoded representation of the document. The number of dimensions (possible document representations) just exploded in order to retain information about the sequence, the order of characters and words.

These representations of documents don't cluster together well, in this character-based vector world. 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'll 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 meaninful 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 examples above. 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 of 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) I think you'd probably conclude that they all probably had similar intent or meaning. You might even notice the capitalization of the word Good and thus 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. Nontheless, a smart chatbot or clever woman of the 1940's 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  'linguistics'."
>>> len(set(s.split()))
11
```

```
>>> import numpy as np
>>> np.arange(1, 11 + 1).prod()  # factorial(11) = arange(1, 12).prod()
39916800
```

The number of permutations exploded from `factorial(3) == 6` in our simple greeting to `factorial(11) == 39916800` in our longer statement! And it's clear that the logic contained in the order of the words is very 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 example above. 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 very important when a statement intends to convey precise, logical relationships between things. That's why computer languiages depend on rigid grammar and syntax rule parsers. Fortunately, recent advances in natural language syntax tree parsers have made it possible to extract syntactical and logical relationships from natural language with remarkable accuracy (greater than 90%).[20] In later chapters we'll show you how to use packages like `SyntaxNet` (Parsey McParseface) and `SpaCy` to identify these relationships.

Footnote 20 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 will show you a trick for incorporating some of the information conveyed by word-order into our word-vector representation. It will also show you how to refine the crude tokenizer used in the examples above (`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", while 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 very similar to the pipeline required to build a question answering system described in *Taming Text*.[21] However, some of the the algorithms listed within the five subsystem blocks may be new to you. We we will help you implement these in Python to accomplish various NLP tasks essential for most applications, including chatbots.

Footnote 21 Ingersol, Morton, and Farris, www.manning.com/books/taming-text/?a_aid=totalgood

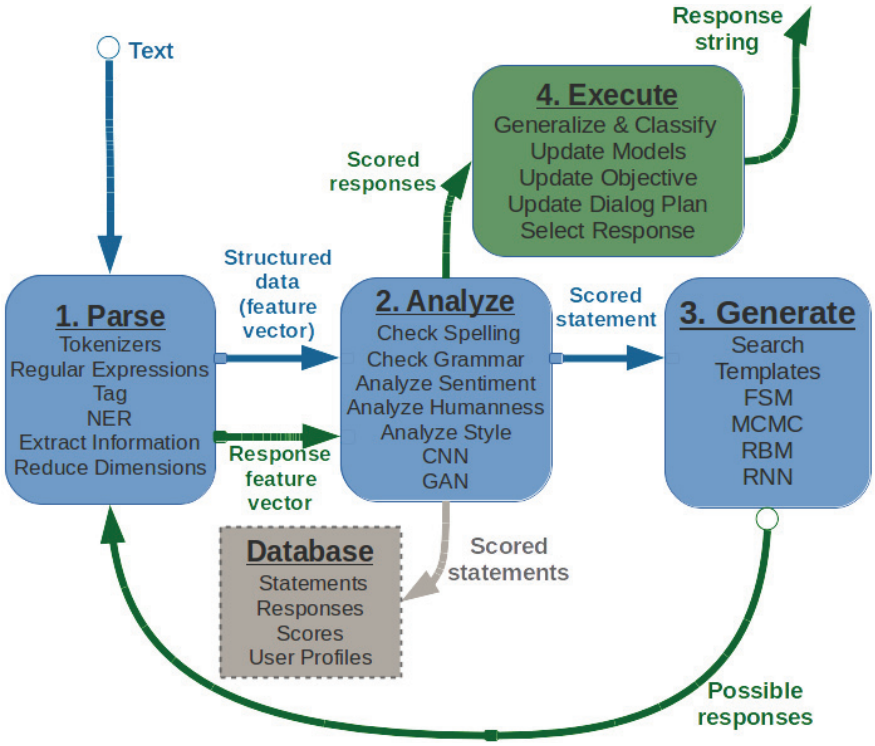


Figure 1.4 Chatbot Recirulating (Recurrent) Pipeline

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.

1. Parse: Extract features, structured numerical data, from natural language text
2. Analyze: Generate and combine features by scoring text for sentiment, gramaticality, 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 will show you how to use Python to accomplish near state-of-the-art performance for each of these processing steps. And we'll 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). However many applications only require 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 and we will 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 simply 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 the diagram above 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 have broken out our pipline into 4 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 which was the focus of *Taming Text*.

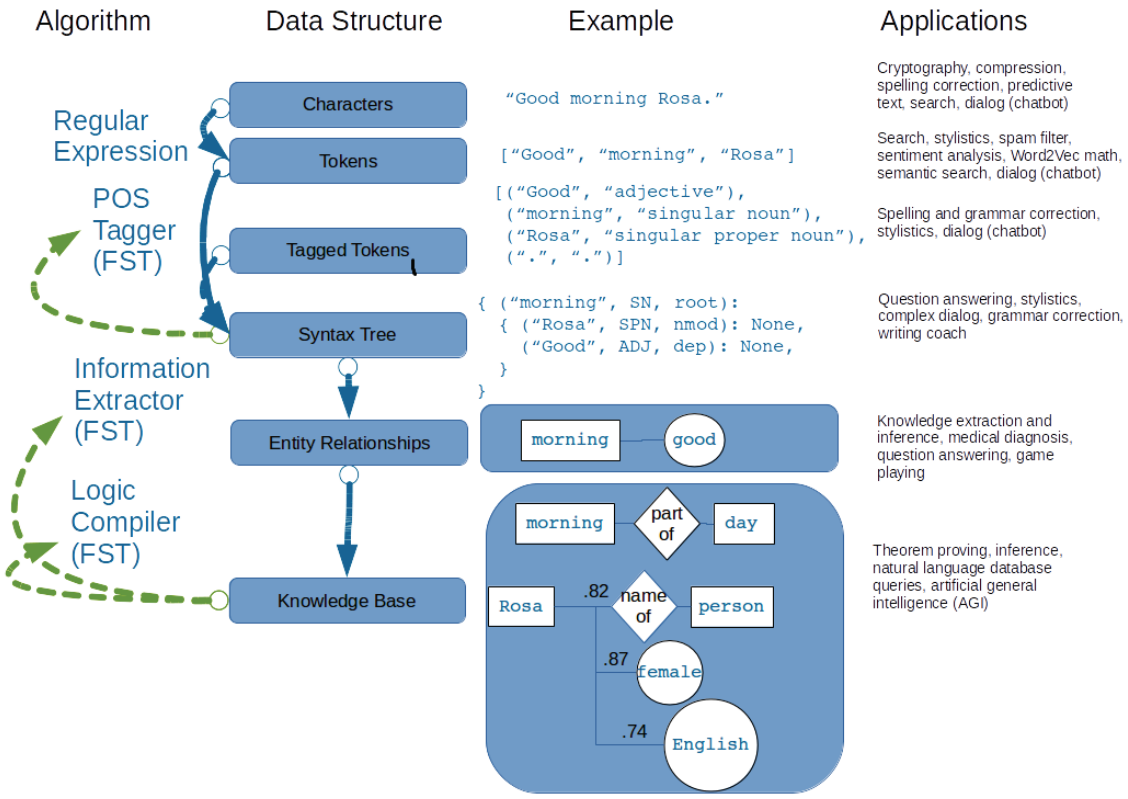
The application of this pipeline to financial forecasting or business analytics may not be so obvious. But if your Anlysis subsystem (stage 2 above) is trained to generate features, scores, that are designed to be useful for your particular finance or business predictions, they can help you incorporate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

One processing element in the diagram above 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 will show 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 pipline can be thought of as layers, like the layers in a feedforward neural network. Deep Learning is all about creating more complex models and behavior by adding additional processing layers to the conventional 2-layer machine learning model architecture of feature extraction followed by modeling. In Chapter 9 we'll explain how neural networks help spread the learning across layers by backpropogating model errors from the output layers back to the input layers. But here we'll talk about the top layers and the what can be done by training each layer independently of the other layers.



The top four layers in this diagram 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 of the top four layers in this diagram. POS tagging is typically accomplished with a Finite State Transducer (FST) like the methods in the `nltk.tag` package. If terms like FST or FSM (Finite State Machine) are confusing, just ignore them for now. Eventually your brain will learn what they mean from the context where we mention them throughout this book.[22]

Footnote 22 Word2Vec, the focus of Chapter 6, learns the meaning of words based on their neighboring words in a sentence, only your brain is better at it, and require fewer examples to generalize, form a model of the word

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 algortihms of the upper few layers. And these decisions can combine to produce surprisingly human-like behaviors.

Over the next few chapters we'll dive down through the top few layers of NLP. The top 3 layers are all that is required to perform meaningful sentiment analysis, 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 it's 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 of our chatbot pipeline, "Generate." 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.[23]

Footnote 23 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.[24]. `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"[25] 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 24 Will a chatbot for HipChat by Steven Skoczen and the HipChat community github.com/skoczen/will

Footnote 25 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 commandline application `grep`: github.com/laurikari/tre/

A lot of powerful things can be done with shallow NLP. And very 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).[26]

Footnote 26 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'll visit these and other unsupervised models in a chapter on natural language embedding.

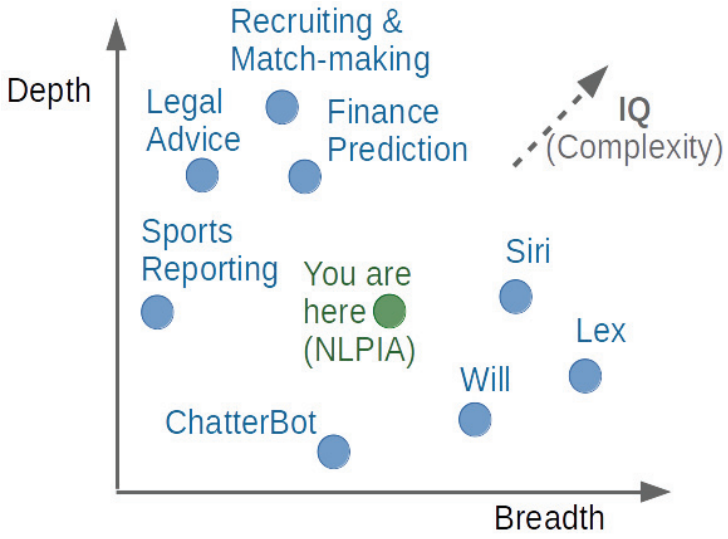
1.9 Natural Language IQ

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 very 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) essentially 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 very 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 the NLP Layers diagram. 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.

Examples of both approaches are used in this book to build a relatively narrow, but deep chatbot dialog engine. This will give you the tools you need to accomplish natural language processing in your "vertical" or application area and expand the breadth of capabilities for this natural language pipeline. The diagram below 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 gage their intellegence by probing them with difficult questions, 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.



As you progress through this book you will be building the elements of a chatbot. Chatbots require all of the tools of NLP to work well: feature extraction to produce a vector space model, information extraction to enable our chatbot to answer factual questions, semantic search to help our chatbot learn from previous dialogs between humans, and natural language generation to compose meaningful, relevant 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 they're 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. 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 this basic chatbot that we started with. In the next chapter we'll detect these same greetings and many many more using machine learning.

1.10 Summary

- Good NLP may save the world
- The meaning and intent of words can be decyphered by machines
- NLP requires some common sense knowledge and the ability to deal with ambiguity
- Chatbots can be thought of as sophisticated 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 undertanding of the imporant and useful NLP tools and techniques. And the chatbot depth and breadth of capabilities can be built up incrementally to help you build systems that approach human-like behaviors, at least for short periods of time. And you should be able to see in upcoming chapters how to direct this capability depth toward an area that interests you, whether it's finance, or sports, psychology or social sciences. As you are learning the tools of Natural Language Processing you'll be building a dialog engine that can help you accomplish your goals in life and in business.

2 Build Your Vocabulary

In this chapter you will learn how to:

- Tokenize your text into words and N-grams
- Build a vector representation of a statement
- Deal with text contractions and abbreviations
- Tokenzie social media texts, like tweets from Twitter
- Compress your token vocabulary with stemming and lemmatization
- Filter out words with negligible information content (stopwords)
- Handle capitalized words appropriately

So you're ready to save the world with the power of Natural Language Processing? 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 going to be 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, etc. Retrieving tokens from a document will require some string manipulation beyond just the `str.split()` method employed in Chapter 1. We'll need to separate punctuation from words and we'll need to split contractions like "we'll" into the words that were combined to form them. And once we've identified the tokens in a document that we'd like to include in our vocabulary, we'll return to our regular expression toolbox to try to combine words with similar meaning in a process called "stemming." Then we'll assemble a vector representation of our documents called a "bag of words" and we'll try to use this vector to see if it can help us improve upon the greeting recognizer we sketched out at the end of Chapter 1.

Think for a moment about what a word or token represents to you in your mind. 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 which have a precise definitions and set of gramatical 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? What about the contraction

"don't"? Should that string of characters be split into one or two "packets of meaning"? Can you think of additional words that are implied by the single-word command "Don't!"?

In this chapter, we'll show you straight-forward algorithms for separating a string into words. We'll also extract pairs and triplets of tokens so that compound words like "ice cream" and words that just belong together, like "Mr. Smith" will stay together in our vector representation. For now, all possible pairs of words will be included in our vocabulary, but in Chapter 3 we'll learn how to estimate the importance of words based on their document frequency, and 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 of the information content of the input data in any machine learning pipeline. But in natural language processing composing a numerical vector from text is a particularly "lossy" feature extraction process. Nonetheless the bag-of-words vectors retain enough of the information content of the text to produce useful and interesting machine learning models.

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" but not from "sing." Or imagine trying to discriminate between a pluralizing "s" at the end of words like "words" and normal "s"es 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'll show you how to make your NLP pipeline a bit smarter by dealing with these word spelling challenges using conventional stemming approaches for now. Later, in Chapter 5, we'll 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.

2.1 Building your vocabulary through tokenization

In Natural Language Processing (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 words and punctuation. In this chapter we will focus on segmenting text into tokens. This is called "tokenization."

Tokenization is a powerful tool. It breaks unstructured data, 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 structured, numerical data structure suitable for machine learning. These counts can be used directly by a computer to trigger useful actions and responses, or they may be used in a machine

learning pipeline as features driving more complex 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` builtin 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.']
```

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 our 90% tokenization solution.

With a bit more Python you can create a numerical vector representation for each word called one-hot word vectors. And a sequence of these one-hot word vectors fully captures the original document text in a numerical data structure!

```
>>> import numpy as np
>>> vocab = sorted(set(sentence.split()))
>>> onehotwords = np.zeros((len(sentence.split()), len(vocab)), int)
>>> for i, word in enumerate(sentence.split()):
...     onehotwords[i, vocab.index(word)] = 1
>>> ' '.join(vocab)
'26. Jefferson Monticello Thomas age at began building of the'
>>> onehotwords
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, 0, 1, 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]])
```

In this representation of a sentence, each row or line is a word. The "1" in a column indicates the word represented by that row or vector. Because these are "one-hot" vectors, all the other positions (for the other words in our vocabulary) are set to zero. A one (1) means "on" or "hot". A zero (0) mean "off" or absent.

One nice feature of this particular vector representation of words is that no information is lost.[27] As long as we keep track of which words are indicated by which column, we can reconstruct the original document from the sequence of one-hot word vectors generated this way. And this detokenization process is 100% accurate even

though our tokenizer was only 90% accurate at generating the tokens we thought would be useful! As a result, one-hot word vectors like this are typically used in neural nets, sequence-to-sequence langauge models, and generative language models, any model that needs to retain all of the meaning inherent in the order of words in the original text.

Footnote 27 Except for the distinction between various white spaces that were "split" on with our tokenizer. There's no way to tell whether a space or a newline or a tab should be inserted at each position between our words during detokenization. But the information content of whitespace is very low, negligible in most English documents.

If you squint hard enough you might be able to imagine that the matrix of ones and zeros above looks a bit like a player piano paper roll [28] or the bumps on the metal drum of a music box.[29] The vocabulary key at the top tells the machine which "note" (word) to play at each line. Our mechanical pianist is only allowed to use one "finger" at a time and can only play one note (word) melodies with a consistent beat. We can play back the original sentence to provide input for a a neural net or any other machine learning pipeline. Or we could just play a sequence of word vecs back to generate text for a chat bot, just like a player piano might play a song for a less artificial intelligence. Now all we need to do is figure out how to build a player piano that can "understand" and combine those word vectors in new ways to play us a song (say something) we haven't heard before.

Footnote 28 en.wikipedia.org/wiki/Player_piano en.wikipedia.org/wiki/Player_piano

Footnote 29 en.wikipedia.org/wiki/Music_box

This representation of a sentence in one-hot word vectors retains all of the detail, grammar, order of the original sentence and puts it into a form that a computer can understand. But if we created a matrix like this, maintaining a consistently ordered vocabulary key for all the sentences in a single book you can see that this would not be very practical. In most cases, the vocabulary of tokens that will be useful for an NLP pipeline is much more than ten thousand, and sometimes hundreds of thousands of tokens long. So if there are a million tokens in your vocabulary, and you want to process, say, 3000 books with 3500 sentences each (an average amount of sentences for a book), you're talking more than a million million bits, tens of gigabytes. So storing all those zeros, and trying to remember the order of the words doesn't make much sense.

What if our documents are much shorter than an entire book, like say the sentence typed into a chat application or the responses generated by a chatbot. And what if we assumed that most of the meaning of a sentence can be gleaned from just the words themselves, jumbled up in a "bag" where we no longer keep track of their order? That turns out to be a reasonable assumption. Even for documents several pages long, a bag-of-words representation is still useful at summarizing and compressing the information content into a data structure that isn't too unwieldy.

If we summed all these one-hot vectors together, rather than "replaying" them one at a time, we'd get a bag of words vector. Alternatively, if we're doing basic keyword search, we could *OR* the onehot word vectors binary bag of words vector (or a true bag of words

with counts) that we could use to represent the whole sentence in a single, reasonable-length vector (only as long as our vocabulary). But just like laying your arm on the piano, hitting all the notes (words) at once 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.

Fortunately, the words in our vocabulary are sparsely utilized in any given text. So rather than hitting all the notes on a piano at once, our bag-of-words vector is more like a broad and pleasant piano chord, a combination of notes (words) that work well together and contain structure, meaning. Our 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 information about a statement that can be used withing a machine learning pipeline.

Here's how we 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, we can keep track of the sentence (or the associated vector) where it occured. And we keep track of every single word (at least for now), whereas a textbook index generally only cares about important words relevant to the subject of the book, words that a reader may want to try to find later.

Here's what our 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 our vocabulary is unimportant. As long as we are consistent across all the documents we 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 our vector ensures that it only has to store a 1 when any one of the thousands, or even millions, of possible words in our 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 our "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 our 10-word sentence. And this dictionary could be made even more efficient if we represented each word as an integer pointer to each word's location within our lexicon—the list of words that makes up our vocabulary for a particular application.

So let's use an even more efficient form of a dictionary, a Pandas `Series`. And we'll wrap that up in a Pandas DataFrame so we can add more sentences to our binary vector "corpus" of texts about Thomas Jefferson. All this hand waving about gaps in the vectors and sparse vs. dense bags of words should become clear as we add more sentences and their corresponding bag-of-words vectors, to our DataFrame (table of vectors corresoding 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    1
```

Let's add a few more texts to our 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 we 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 very little overlap in word usage for these sentences. Among the first seven words in our vocabulary, only the word Monticello appears in more than one sentence. Now we just need to be able to compute this overlap within our

pipeline whenever we 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 we can tell that one word was used in both `sent0` and `sent2`. Likewise one of the words in our 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 Montecello 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 us that last dot product of 1:

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

This is our 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, etc. We can even compute things like 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 A Token Improvement

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

In the example above, the last token in the sentence was corrupted by a period at the end, "26." The trailing period can be misleading for the subsequent sections of an NLP pipeline, like stemming, where we 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',
 'twenty', 'six']
```

This regular expression splits the sentence on white space or punctuation which 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 will split 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 patterns [30]

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, since 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, we have a new problem. We need to filter the whitespace and punctuation characters that we do not want to include in our 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']
```

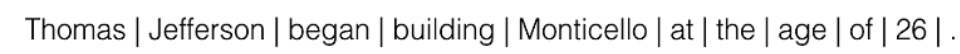


Figure 2.2 Thomas|Jefferson|began|building|Monticello|at|the|age|of|26

So the built-in Python `re` package seems to do just fine on this example sentence, as long as we 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. It's 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)

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 like pip:

```
$ pip install regex
```

More information about the *regex* module can be found on the PyPI website (pypi.python.org/pypi/regex)

As you can imagine, tokenizers can easily become very 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.

Various Python libraries offer tokenizers for different purposes. One of the popular NLP libraries in Python is the Natural Language Toolkit (NLTK). You can use the NLTK function `RegexpTokenizer` to replicate our simple word 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 we used originally, since 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 while retaining 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, e.g. the stemming. All rules for the Treebank tokenizer can be found 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 Monticello|wasn't|designated|…

2.2.1 Contractions

You might wonder why we would split the contraction `wasn't` into `was` and `n't`. For some applications, like grammar-based NLP models which utilize syntax trees, it is 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 everrrrrrr at Monticello.
    Awesommmmmmeeeeeeee day :*)"
>>> casual_tokenize(message)
['RT', '@TJMonticello', 'Best', 'day', 'everrrrrrr', 'at', 'Monticello', '.',
    'Awesommmmmmeeeeeeee', 'day', ':*)']
>>> casual_tokenize(message, reduce_len=True, strip_handles=True)
['RT', 'Best', 'day', 'everrr', 'at', 'Monticello', '.',
    'Awesommmeee', 'day', ':*)']
```

2.3 Extending your vocabulary with n-grams

2.3.1 What are n-grams?

An *n-gram* is a sequences containing up to *n* elements which 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.[31] For now, we're only interested in N-grams where our "grams" are words, not characters.

Footnote 31 Linguistic and NLP techniques are often used to glean information from DNA and RNA, here's 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_acid_abbreviations_and_properties

Why bother with N-grams? As we 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 in the original sentence. By extending our concept of a token to include multi-word tokens, N-grams, our NLP pipeline can retain much of the meaning inherent in the order of words in our statements. For example, the meaning inverting word "not" will remain attached to it's neighboring words, where it belongs, rather than "floating free" to be associated with the entire sentence or document. 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 we tie it to it's neighbor(s) in our pipeline.

And in the next chapter we'll show you how to recognize which of these N-grams contain the most information relative to the others, so our chatbot doesn't have to maintain a list of all the posssible word sequences. This will help it recognize "Thomas

Jefferson" and "ice cream", but not pay particular attention to "Thomas Smith" or "ice shattered." And in Chapter 4 we'll associate word pairs, and even longer sequences, with their actual meaning, independent of the meaning of their individual words. But for now, we just need our tokenizer to generate these sequences, these N-grams.

Let's use our original sentence about Thomas Jefferson to show what a 2-gram tokenizer should output, so we know what we'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 we'd just tokenized the sentence into words. The later stages of our NLP pipeline will only have access to whatever tokens our tokenizer generates. So we 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 our pipeline.

Here's our 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 ngram 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 which behave like iterators, yielding only one element at a time instead of returning the entire sequence at once. This is especially 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 we did in the example above. But keep in mind that you should only do this in an interactive session, not within a long-running tokenization of large texts.

The n-grams are provided above as tuples, but they can easily be joined together if we'd like all of the tokens in our 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 our 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 we can use to help identify topics or themes that connect documents or classes of documents. So they will not be helpful for classification problems. And you can imagine that most 2-grams are pretty rare. And this is even more true for 3 and 4-grams. And because word-combinations are rarer than individual words, our vocabulary size is exponentially approaching the number of N-grams in all of the documents in our corpus. If our feature vector dimensions exceeds the length of all our documents, our feature extraction step is being counter productive. It will be impossible to avoid overfitting if our dimensions is larger than the number of documents in our corpus. So we will have to use "document frequency" statistics in Chapter 3 to identify N-grams that are so rare that they are not useful for machine learning. Typically N-grams are filtered out which occur too infrequently (e.g. in 3 or fewer different documents). This is represented by the "rare token" filter in our coin-sorting machine diagram of Chapter 1.

Now consider the opposite problem. Consider the 2-gram "at the" in the phrase above. That's probably not a very rare combination of words. In fact it might be so common, spread among most of our documents, that it loses its utility for discriminating between the meanings of our documents. It has little "predictive power". Typically N-grams are filtered out if they ocurr too often (e.g. more than 25% of all the documents in our corpus). This is equivalent to the "stop words" filter in our coin-sorting machine diagram of Chapter 1. These filters are as useful for N-grams as they are for individual tokens. In fact they are even more useful.

2.3.2 Stopwords

Stopwords are common words in any language which occur with a high frequency, but carry much less substantive information about the meaning of a phrase. Examples of some common stopswords include:[32]

Footnote 32 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

- 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 very little information, the stop words can provide important relational information as part of an n-gram. Consider the two examples:

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

In our NLP pipeline, we might create 4-grams such as `reported to the CEO` and `reported as the CEO`. If we remove the stop words from the 4-grams, both examples would be reduced to `reported CEO` and we would lack the information about the professional hierachy. 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 stopwords within our 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 forces us to retain at least 4-grams if we want to avoid the ambiguity of the human resources example here.

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. And vocabulary size drives the required size of any training set you must aquire and process. If you have sufficient memory and processing bandwidth to run all the NLP steps in your pipeline on the larger vocabulary, and you have sufficient training data to avoid overfitting for this larger vocabular, consider keeping the stop words in your corpus. 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 we take a few stopwords and ignore them when we iterate through our 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 we 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 us, has low information content. The nltk package includes pronouns (not just first person ones) in it's list of stopwords. And these one-letter stopwords 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.

2.4 Normalizing your vocabulary

So we'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. This will reduce 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 reudcing your vocabulary can reduce the likelihood of overfitting.

2.4.1 Case normalization

Normalizing word capitilization 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, e.g. 'doctor' and 'Doctor' often have different meanings. Often, capitilization is used to indicate that a word is a proper noun, the name of a person, place, or thing. We'll want to be able to recognize proper nouns if "named entity recognition" is important to our pipeline. However, if tokens are not case normalized then your vocabulary 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 that your model must deal with. For 100000-D bag-of-words vectors, that means you usually must have of 100000 labeled examples, and often many times 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 simply `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 written in all caps for emphasis. Undoing this denormalization is called "case normalization."

With case normalization we 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 is to simply lowercase all the characters with a function like Python's builtin `str.lower()`. Unfortunately this approach will also "normalize" away a lot of meaningful capitalization as well as the lesss meaningful first-word-in-sentence capitalization that we want to normalize away. A better approach for case normalization is to only lowercase 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 our 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 simply do no normalize for case at all. The benefit of reducing ones 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 we searched for the word "Age" than if we searched for "age". "Age" would likely occurr 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 our sentence about Thomas Jefferson. By normalizing the vocabulary in our search index (as well as the query) we 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. Thus modern search engines allow normalization to be turned off by the user with each query, typically by quoting those words for which you want only exact matches returned. If you are building such a search engine pipeline, in order to accomidate 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.

2.4.2 Stemming

Another common vocabulary normalization technique is to eliminate the small meaning differnces of pluralization or posessive endings of words, or even various verb forms. This normalization, identifing 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, we 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 of 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 agressiveness of the stemmer you chose.

Stemming is especially 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 of 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`, then the stem is the word and the suffix is a blank string
- If a word ends with a single `s`, then the stem is the word without the `s` and the suffix is the `s`
- If a word does not end on an `s`, then 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.

Of course it also doesn't handing the "housing" example from our "Portland Housing" search example

Two of the most popular stemming algorithms are the Porter and Snowball stemmers. Named after the computer scientist Martin Porter, the algorithm[33] follows more complex rules to accomodate more complex cases of English grammar.

Footnote 33 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 our 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, so this can be important for applications where you want to treat names differently than other nouns.

2.4.3 Lemmatization

And if we have access to information about connections between the meanings of various words we 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 lemmatization can be used 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. This 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 very 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 is called "spoofing" when you try to elicit the wrong response from a machine learning pipeline by clevely constructing a difficult input.

Lemmatization is a more accurate way to normalize a word than stemming or case normalization because it takes into account the meaning of a word. 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 Part of Speech (POS) of that word 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 we 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'
```

2.4.4 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 thus its intended meaning. If your application involves search, then 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 meaning of the original word. 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.

2.5 Summary

In this chapter we learned:

- How to implement tokenization and configure a tokenizer for your application
- Several techniques for improving 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.
- Stopwords can contain useful information and it’s not always helpful to discard them.

3 Math with Words

In this chapter:

- Counting words and *Term Frequencies* to analyze meaning
- Word occurrence probabilities and *Zipf's Law*
- Vector representation of words and how to start using them
- Finding relevant documents from a corpus using *Inverse Document Frequencies*

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" weight to find relevant documents in a corpus based on keyword importance within each document. And 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. 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 NLP journey is fraught with peril though. The *uncanny valley* lies before us. The uncanny valley is the region where a machine starts to appear lifelike, but retains its machine awkwardness. Humans tend to respond less and less favorably to it, dramatically so, as the machine gets more and more human-like but is still recognizably a machine. It's getting creepier and creepier the better it gets at faking some aspects of human behavior. That trend downward in acceptance of a technology is only reversed once the machine becomes sufficiently lifelike so as to be indistinguishable from a live, conscious organism.

Imagine how uncanny it would be if a player piano started playing darker music as

the sun began to go down or rain started pattering on the windows. And what if it played with human-like emotion and variation. And if we improved the piano's "responsiveness" by making it imitate any song that you whistled or hummed, that might start to seem a bit creepy, like it was paying too much attention to you. And if it guessed your mood, somehow registering your facial expression and body language, that would be too much. But if it was able to speak back to you, carry on an intelligent conversation about the music it was about to play, and played spectacularly difficult and beautiful music, on request, that might help it start to climb out of the far side of the uncanny valley in your eyes.

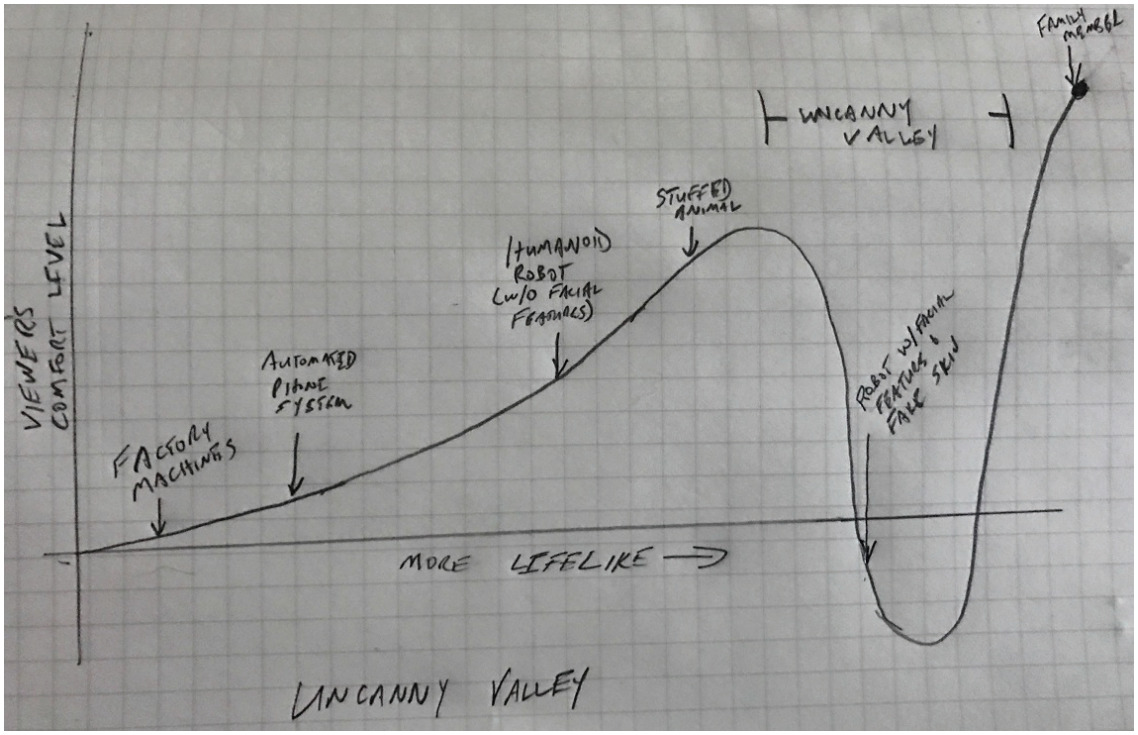


Figure 3.1 Uncanny Valley

Ultimately we want to venture into the dark, uncanny valley with our NLP machine, and ideally guide it to the far side. We'd like it to understand words so well that it can generate text on its own, text that sounds almost human. So the next step on our adventure is to turn the words of Chapter 2 into continuous numbers rather than just binary "bit vectors" that detect the presence or absence of particular words. Then our NLP machine can do what computers do best, math. Our goal is to find numbers for 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.

There are several paths into the uncanny valley. We'll take you part of the way down the first of 3 of these forks in the path. These branches come together in the end, so none of this exploration will be wasted. Plus, you can apply each of these techniques

separately as its own NLP pipeline. This often called "shallow learning" NLP. These shallow NLP machines are useful for practical applications like spam filtering and sentiment analysis.

These paths lead us to 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 (we'll get to this in a chapter or two)
- TF-IDF Vectors: word scores that better represent their importance

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.

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 can 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 Chapter 1 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. It's a bag of words, not a vector or sequence… yet. 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.

```
>>> # by default, `most_common()` lists all the objects from most frequent to least,
      but you can request any number
>>> bag_of_words.most_common(4)
[('the', 4), (',', 3), ('faster', 3), ('harry', 2)]
```

Specifically, 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. However this is an unnecessary computation that is undone by the 2-norm normalization at the end of the TF-IDF calculation.

So Our top four terms are 'the', ',', 'harry', and 'faster'. But the word 'the' and the punctuation ',' aren't very informative about the intent of these documents. And these uninformative terms (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 words in our TF vector (bag of words):

So our TF vector for this document first two elements in our T

```
>>> times_harry_appears = bag_of_words['harry']
>>> num_unique_words = len(bag_of_words)  # The number of unique tokens from our original source.
>>> tf = times_harry_appears / num_unique_words
>>> round(tf, 4)
0.1818
```

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

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

Figure 3.3 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 in the vectors we will use *Term Frequencies*.

Similarly we could calculate each word and get the relative "importance" to the document of that term. Our protagonist and his need for speed are clearly central to the story of this document, we've made some progress in turning text into numbers. 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 reac against the air to create lift and drag. A kite consists of wings, tethers, and anchors. Kite 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 sailplan for launch, the tether meets the wing at a single point. A kite may have fixed or movin anchors. Untraditionally in technical kiting, a kite consists of tether-set-coupled win 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' surface, producing low pressure above and high pressure below the wings. Th 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 th tension of one or more of the lines or tethers to which the kite is attached. The ancho point of the kite line may be static or moving (e.g., the towing of a kite by a runnin 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 kit lifting surface is called a kytoon.
>
> Kites have a long and varied history and many different types are flown individuall 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 kite

> are multi-line steerable kites designed to generate large forces which can be used t
> power activities such as kite surfing, kite landboarding, kite fishing, kite buggying and
> 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 import kite_text  # kite_text = "A kite is traditionally ..." as above
>>> tokens = tokenizer.tokenize(kite_text.lower())
>>> token_counts = Counter(tokens)
>>> token_counts
Counter({'the': 26, 'a': 20, 'kite': 16, ',': 15, ...})
```

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 there 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 "programmaticly" 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 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_count.most_common():
    document_vector.append(value / doc_length)

print(document_vector)

[0.07207207207207207, 0.06756756756756757, 0.036036036036036036, 0.02252252252252252,
    ..(+ others).., 0.0045045045045045045, 0.0045045045045045045]
```

This list, or *vector*, is something we can do math on directly. (TIP) There are many ways to speed up processing of these data structures[34], but for now as we are playing with the nuts and bolts, raw Python has all the tools we need.

Footnote 34 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 (check) and second, making all of the vectors of standard length.

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*. 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 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
>>> vector_template = OrderedDict((token, 0) for token in lexicon)
>>> vector_template
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 an array.

```
>>> import copy
>>> doc_vectors = []
>>> for doc in docs:
...     # Need multiple independent instances, not references to the same object:
...     vec = copy.copy(vector_template)
...     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)
```

And now we have 3 vectors, one for each document. So what? What can we do with them? Let's see what they are first.

Vectors, at least in the way we are going to use them are the primary building blocks of linear algebra. They describe a direction and magnitude in a given 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 space, one with 3 values in 3-D space, etc. But we aren't limited to normal 3-D 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 deal with those in Chapter 5.[35]

Footnote 35 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). See
en.wikipedia.org/wiki/Curse_of_dimensionality
docs.google.com/presentation/d/1SEU8VL0KWPDKKZnBSaMxUBDDwI8yqIxu9RQtq2bpnNg
github.com/spotify/annoy
scholar.google.com/scholar?q=high+dimensional+approximate+nearest+neighbor

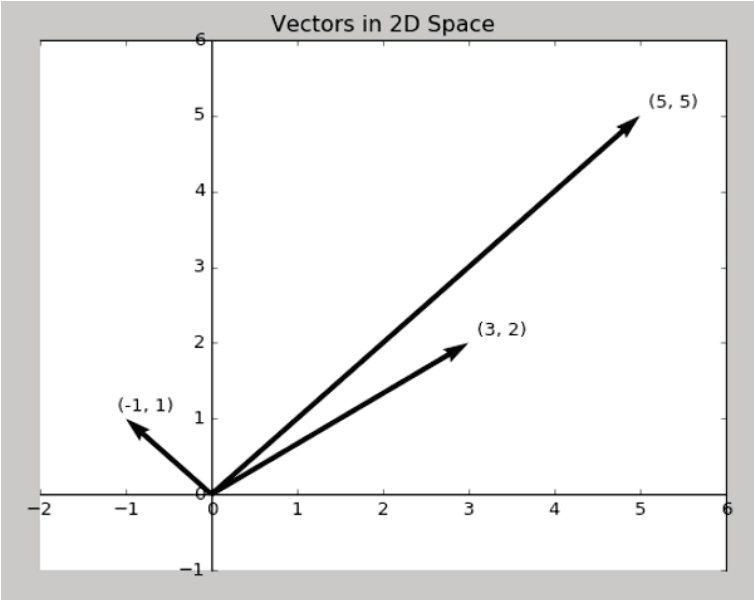


Figure 3.4 2-D Vectors

To decide how big our vector space is, we count the number of distinct words that appear in the entire corpus and call this capital K. 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 your reading.

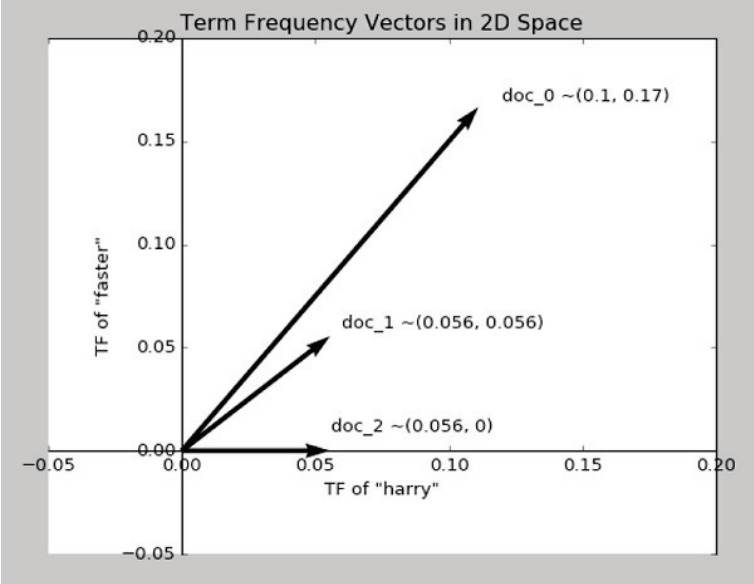


Figure 3.5 2-D Term Frequency Vectors

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 the Appendix 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 and similar magnitude (length). They are similar because they use the same words about the same amount of times. So they are probably talking about similar things.

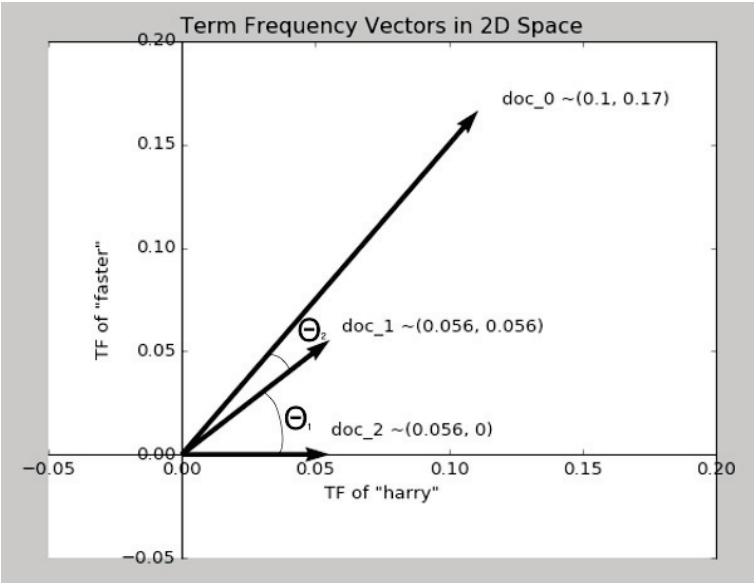


Figure 3.6 2-D Thetas

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

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

Figure 3.7 Dot Product: A dot B = norm(A) * norm(B) * cos(theta)

From which we can then derive:

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

Figure 3.8 Cosine Similarity: cos(theta) = A dot B / norm(A) * norm(B)

Or in Python:

```
>>> import math
>>> def cosine_sim(vec1, vec2):
...     """ Since our vectors are dictionaries, lets convert them to lists for easier mathing. """
...     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 each element pairwise and then sum up) and 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 origin, the square root of the sum of the squares of its elements. This normalized dot product, like cosine, will be a value between -1 and 1, because it is the cosine of the angle between these 2 vectors. A cosine similarity of 1 represents identical vectors that are exactly equal along all dimensions. A cosine similarity of -1 represents two vectors that are anti-similarly, completely opposite. So with a cosine similarity that is closer to 1 the vectors can be considered similar. And if two documents' vectors are similar, we can say, in some way, the documents themselves share the qualities that are modeled in the vector. Now, we will see later that we can get vectors that more accurately model a document, but this gives a clean introduction to the tools.

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. From the wikipedia entry, "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." 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:

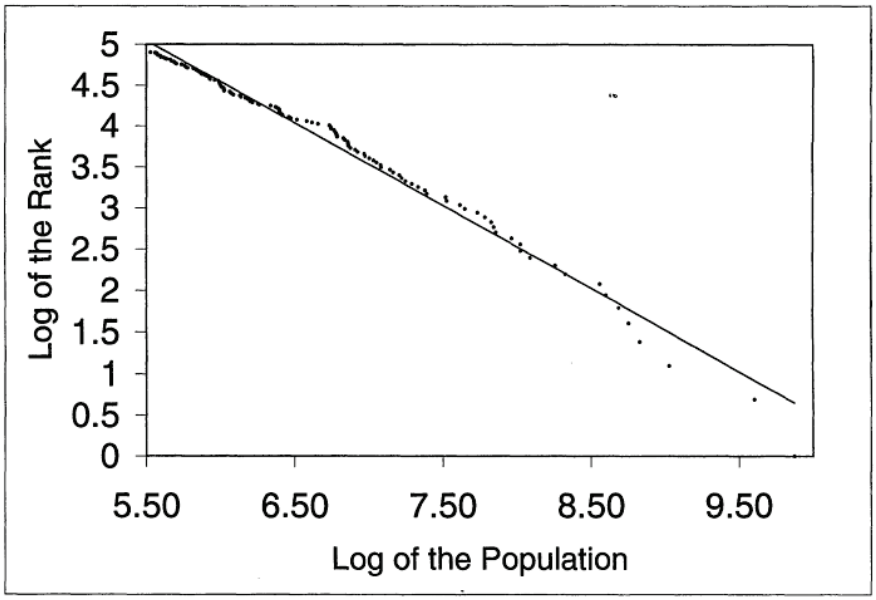


FIGURE I
Log Size versus Log Rank of the 135 largest U. S. Metropolitan Areas in 1991
Source: Statistical Abstract of the United States [1993].

Figure 3.9 City Population Distributions

> The usual complaint about economic theory is that our models are oversimplified — tha they offer excessively neat views of complex, messy reality. [With Zipf's law] th 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 source have been categorized by genre, such as news, editorial, and so on. 1.1 gives an example of each genre.[36]

Footnote 36 for a complete list, see icame.uib.no/brown/bcm-los.html

-- NLTK Documentation

```
>>> from nltk.corpus import brown
>>> len(brown.words())  # words is a builtin method of the nltk corpus object that gives a
    list of tokens
1161192
```

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 taken relative to 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 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 import kite_text, kite_history
>>> kite_intro = kite_text.lower()  # "A kite is traditionally ..."  # Step left to user, as above
>>> 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
```

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 a 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 seemd 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 total documents / 2 documents contain *and* = 2/2 = 1 2 total documents / 2 documents contain *kite* = 2/2 = 1

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

2 total documents / 1 document contains *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 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 the base of log function is not important, merely the smooth tempering effect as # of Documents / # of Documents containing our word grows large. For the ease of this example, we'll use base 10 and get:

search: cat

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

search: dog

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

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:

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

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

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

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(vector_template)  # So we are dealing with new objects, not multiple
                                            references to the same object
...     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)
```

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:

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

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(vector_template)
>>> query_vec = copy.copy(vector_template)  # So we are dealing with new 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:  # We didn't find that token in the lexicon go to next key
        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
```

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*.[37]

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 `Whoosh` [38] package and its source code.[39] Instead of showing you how to build this conventional keyword-based search engine, in Chapter 5 we'll show you the latest semantic indexing approaches that capture the meaning of text.

Footnote 37 en.wikipedia.org/wiki/Inverted_index

Footnote 38 pypi.python.org/pypi/Whoosh

Footnote 39 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).[40]will usually improve the search results for TF-IDF keyword-based searches.

Footnote 40 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, a.la "I'm feeling lucky" search engine buttons. 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 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.[41] 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 41 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
>>> corpus = docs
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> model = vectorizer.fit_transform(corpus)
>>> # The TF-IDF model is a sparse numpy matrix, because a large corpus TF-IDF would be
      mostly zeros
>>> # The .todense() method converts this matrix back into a regular numpy matrix for our
      viewing pleasure.
>>> 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.    0.29  0.22  0.    0.29  0.29  0.38  0.     0.    0.    0.    0.  ]]
```

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	$w_{ij} = \frac{f_{ij}}{0.5+1.5\times\frac{f_j}{\frac{f_j}{j}}+f_{ij}} \log\frac{N-n_j+0.5}{f_{ij}+0.5}$
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	$\max(0, \text{MI})$
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	$w_{ij} = \frac{\log f_{ij}+1}{\log n_j+1}$

If you're building a search engine that relies on keyword (term) matching between queries and a documents in a corpus, then you should spend some time investigating each of these. You can optimize your pipeline by chosing 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 TFIDF 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 Chapter 5 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.

3.5 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
- 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.

By finding a path from text to numbers we can begin to manipulate them. First to find meaning and similarity, from there it is a short hop to rudimentary Topic modeling. Numbers firmly in hand, let's delve a little deeper finding even more representative vectors for our text.

4 Finding Meaning in Word Counts

In this chapter:

- Creating topic vectors from text
- Understanding PCA, LSA, LSI, and SVD
- Creating topic vectors words in your lexicon
- Estimating the semantic similarity between documents or words
- Implementing semantic search
- Scaling semantic search for large corpora
- Using semantic topics as features in your NLP pipeline
- Navigating high-dimensional vector spaces

We've made a lot of progress in our climb up the ridge that surrounds the uncanny valley. Our next step will be to try to extract the meaning (semantics) from a bag of words. Using TF-IDF vectors from Chapter 3 we can estimate the importance of words in a text document. And we can tell how important each of those words are to the meanings expressed across a collection of documents or statements represented in a TF-IDF matrix. And this TF-IDF "importance" score works not only for words, but also for short sequences of words that occur together a lot, n-grams. This gives us a good way to identify the words and n-grams that best represent the subject (topic) of a statement, document, or corpus (collection of documents). And we can tell how "close" two statements or documents are to each other by how often two documents use these important words that they share.

Unfortunately this approach relies on counts of exact spellings of terms in a document, or relies on approximate "normalization" of words into bins for words with similar spellings (see *stemming* and *lemmatization* in Chapter 2). So two documents that talk about the same thing but use completely different words or spellings will not be very "close" to each other, no matter what similarity score we use (like TF-IDF cosine distance or BM25 from Chapter 3). For example, the cosine distance between the TF-IDF vector for this chapter in *NLPIA* that you're reading right now may not be as close to similar-meaning paragraphs in university textbooks or academic papers about "Latent

Semantic Indexing." Even though that's exactly what this chapter is about, we use colloquial terms familiar to a wider audience than the precise, formal terminology of a professor of statistical linguistics. Plus, the terminology that professors used a decade ago (like "latent semantic indexing") has evolved into the more common term "semantic analysis".[42]

Footnote 42 But our use of the formal terms in paragraphs like this may have brought these TF-IDF vectors closer together

And when we try to do math on these TF-IDF vectors, like vector addition and subtraction, these sums and differences only tell us about the frequency of word uses in the combinations of documents whose vectors we combined or differenced. That math doesn't tell us much about the "meaning" behind those words. "Vector oriented reasoning" doesn't work on Bag-of-Word or TF-IDF vectors.

So we need a way to extract some additional information, meaning, from a bag of words. We need a better estimate of what the words in a document "signify." We'd like to represent that meaning with a vector like a TF-IDF vector, only better, more compact, more meaningful. NLP experimenters before us have found a tool for revealing the meaning of word combinations. And when we use this tool, not only can we represent the meaning of words as vectors (word vectors), but we can represent the meaning of entire documents with those vectors (topic vectors).

The topic vectors and word vectors you'll create in this chapter can be added and subtracted with good effect. You can create new topic vectors with meaningful relationships to all the other topic vectors for documents in your corpus. This enables a basic form of "vector-oriented reasoning."[43]

Footnote 43 An even more powerful word vector representation is coming up in Chapter 6 that allows advanced vector-oriented reasoning like solutions to word analogy problems

4.1 From Word Counts to Topics

Coming up with a numerical representation of the semantics (meaning) of words and sentences can be tricky. This is especially true for dynamic, "fuzzy" languages like English with multiple dialects and many different interpretations of the same words. Even formal English text written by your English professor suffers from polysemy, a challenge for the new learner, including machine learners.

- *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.

- *Homonyms*
  words with the same spelling and pronounciation, but different meanings
- *Homographs*

words spelt the same, but with different pronounications and meanings

- *Zeugma*
  use of two meanings of a word simultaneously in the same sentence
- *Homophones*
  words with the same pronounication, but different spellings and meanings (an NLP challenge when generated text will be spoken)

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 (one for each word or term) down to a vector with 200 or so dimensions (one for each "topic" or component of meaning)? Might it be possible to have a dimension This is like identifying the right mix of primary colors to try to reproduce the color of a paint flake. 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. And we could have negative weights for words that reduce the likelihood that the text is about that topic. If we were doing this by hand for a small vocabulary of only six words and only three topics, we might do something like:

```
>>> topic = {}
>>> tfidf = dict(list(zip('cat dog apple lion NYC love'.split(), [1, 1, 1, 1, 1, 1])))  (1)

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

(1) topic and tfidf are arbitrary vectors to demonstrate how a human might combine term frequencies in a weighted sum to produce values for these three topics from any TF-IDF vector

We arbitrarily chose to decompose three demonstration topics ("pet", "animal", and "city"). And I decided that the terms "cat" and "dog" should have similar contributions to the "pet" topic (weight of .3). And I decided that the term "NYC" should have a negative weight for the "pet" topic, but the word "love" should have a positive weight. After all, we humans tend to love our pets.

And notice there's a small amount of "apple" in 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. If you read the rest of the weighted sums you might be able to see how we came up with these weights for these three topics and six words.

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 be hand, we've chosen to normalize our topic vectors by the easy-to-compute L1 -norm (Manhattan, taxicab, or city-block distance) so the absolute value of our word weights sum to 1.0 for each topic. Nonetheless, the Latent Semantic Analysis (LSA) algorithm below normalizes topic vectors by the more useful L2-norm (conventional Euclidean distance or length).

And you might have realized in reading these vectors that the relationships between words and topics is symmetric. Our 3x6 matrix of three topic vectors above can be transposed to produce topic weights for each word in our vocabulary (lexicon). These vectors of weights would be our word vectors for our vocabulary of six words:

```
>>> word_vector['cat']   =  .3 * topic['pet'] + .1 * topic['animal'] +  0 * topic['city']
>>> word_vector['dog']   =  .3 * topic['pet'] + .1 * topic['animal'] - .1 * topic['city']
>>> word_vector['apple'] =   0 * topic['pet'] - .1 * topic['animal'] + .2 * topic['city']
>>> word_vector['lion']  =   0 * topic['pet'] + .5 * topic['animal'] - .1 * topic['city']
>>> word_vector['NYC']   = -.2 * topic['pet'] + .1 * topic['animal'] + .5 * topic['city']
>>> word_vector['love']  =  .2 * topic['pet'] - .1 * topic['animal'] + .1 * topic['city']
```

Now we have a hand-crafted three-topic model for any natural language document that use some of these six words! Our corpus can use many more words, but this particular topic vector computation 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) to compute. In the thought experiment above we compressed six dimensions (TF-IDF normzlized frequencies) into three dimensions (topics).

This fuzzy manual approach relies on one human's intuition. So it isn't repeatable—you'd probably come up with different weights than I did. And obviously this isn't suitable for a machine learning pipeline (humans aren't machines). 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 or statements that we might want our machine to read and understand.

So let's see if we can automate this manual procedure. Let's use an algorithm to select topic weights for us automatically.[44]

Footnote 44 The wikipedia page for topic models has a video which shows how this might work for many more topics and 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 a pair of weighted sums is just a matrix multiplication or inner product of a 3 x *n* weight matrix with our 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 just a new 3x1 topic vector for that document. What we've done is "transform" a vector from one vector space

(TF-IDFs) to another (topic vectors). So we want an algorithm that can create a matrix of *n* terms by *m* topics that we could multiply (inner product) with a vector representing the words 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.

But 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 word [45] signifies. In 1957 he gave us a clue when he wrote:

Footnote 45 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 cooccurrences 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 cooccurrences" approach led to the development of several algorithms for creating vectors to represent the statistics of word usage within documents or sentences. Each of these techniques can be used to create vectors to represent the "meaning" of a word or collection of words (a document).

- Latent Dirichlet Allocation (abbreviated *LDiA* to distinguish it from LDA, below)
- Latent Semantic Analysis/Indexing (LSA or LSI) or Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)
- Quadratic Discriminant Analysis (QDA)
- Random Projection (RP)
- Nonnegative Matrix Factorization

The most commonly used approaches are *Latent Dirichlet Allocation* (*LDiA*)[46] and Latent Semantic Analysis (LSA).

Footnote 46 The nonstandard LDiA acronym disambiguates the polysemy of the acronym "LDA" which is more commonly used to refer to Linear Discriminant Analysis

Latent Dirichlet Allocation (*LDiA*) does a lot of the things we did to create our 3-topic model above using statistics to make decisions. But because it's more complicated and isn't as widely used (and useful) as *Latent Semantic Analysis* (LSA),

let's start there instead. LSA may be the oldest and most commonly-used technique for dimension reduction for machine learning. The underlying mathematics, Singular Value Decomposition (SVD), is useful for extracting features in a wide range of machine learning problems besides NLP, like image recognition and time-series modeling. For non-NLP problems, like image processing, it is usually called *Principal Component Analysis* (PCA).

LSA is a mathematical technique for finding the "best" way to linearly transform (rotate and stretch) any set of 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 data. 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. That's called *Truncated Singular Value Decomposition*. And we'll show you a trick that helps improve the accuracy of LSA vectors considerably. This trick may also be useful when you are doing PCA for machine learning and feature engineering problems in other domains.

Let's start with LSA, since it's based on some mathematics that you're likely familiar with.

4.2 Latent Semantic Analysis (LSA)

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. PCA on image pixel vectors is the same as LSA on the natural language TF-IDF vectors.

LSA uses 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 3 6-D topic vectors in your manually-computed examples at the beginning of this chapter. Each of our dimensions (axes) becomes a combination of word frequencies rather than a single word frequency. So you we think of them as the weighted combinations of words that make up various "topics" in our corpus.

What is even better, we can discard all those dimensions (topics, or combinations of words) that have the least amount of variance in our data. 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.

So let's review our Bag-of-Word (BOW) vectors and TF-IDF vectors from Chapter 2 to remind ourselves what they look like before we start rotating them around with SVD and LSA.

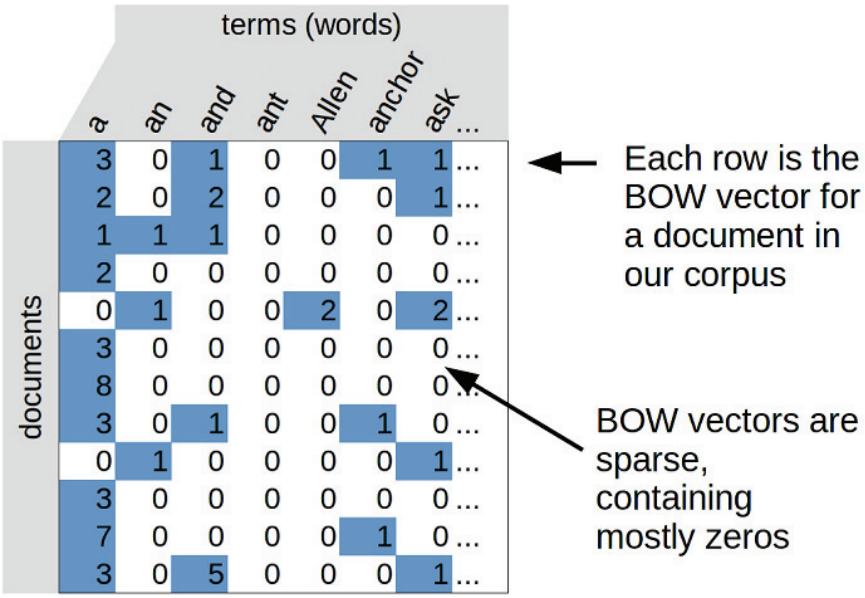


Figure 4.1 Term-Document Matrix for BOW Vectors

BOW vectors are sparse (mostly zeros) and contain only integer values for the counts of each word in a document. Do you remember what TF-IDF vectors look like? What makes them different from BOW vectors?

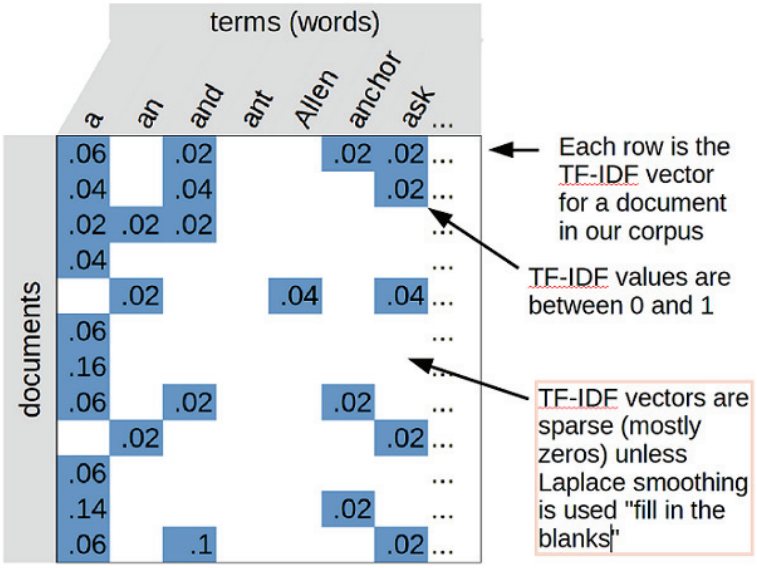


Figure 4.2 Term-Document Matrix for TF-IDF Vectors

Remember how we "normalized" the word counts in our BOW vectors to account for both the lengths of the documents (to get term frequencies) and also for the importance of words based on their document frequencies? And we normalized all our TF-IDF vectors by their `2-norm` (Euclidean length) to ensure that they all had the same length, 1.0. So the values in a TF-IDF matrix will always be between 0.0 and 1.0.

Both the BOW term-document matrix and the TF-IDF term-document matrix are often called the "context matrix." These matrices provide the context in which a word was found, "the company that a word keeps."

Let's keep our eye on the prize and take a look at our imaginary 3-topic vector space model from earlier and expand it to 20 dimensions, a more realistic number of dimensions (topics) for an LSA model of short documents like tweet, chat,and SMS messages. There's not likely to be much of a pattern here in the pattern of weight magnitudes or signs, but notice how they are typically small values close to zero because the L2-norm must be 1.0.[47]

Footnote 47 These are nominal, arbitrary values, not real values from an SVD computation. So examine your SVD matrices to see if you can discover a pattern that helps you understand the statistics of word cooccurrences in your documents.

documents \ topics	topic1	topic2	topic3	topic4	topic5	topic6	...	topic20
	.15	-.03	.02	-.01	.07	.02	...	-.03
	-.04	.01	.04	.01	.01	-.03	...	.12
	.12	.02	-.02	.01	.01	.01	...	-.02
	.04	-.05	-.04	-.03	.01	.01	...	-.03
	-.02	.02	-.05	-.01	.04	.05	...	.04
	.06	-.05	.01	.08	-.01	-.01	...	-.02
	.16	-.01	-.04	-.03	-.02	-.02	...	-.04
	.06	-.05	.02	.05	-.04	.02	...	-.03
	-.02	.02	-.02	-.01	-.03	.05	...	.02
	.06	.05	-.01	-.04	-.05	-.04	...	-.03
	.14	-.01	-.05	-.03	-.01	.02	...	-.04
	.06	-.05	.1	-.05	-.01	.05	...	.02

Figure 4.3 Topic-Document Matrix for LSA model with 20 dimensions

The rows in this topic-document matrix are called the the "document topic vectors" or just "topic vectors."

We should also keep track of all those weights we assign to each word for each topic:

topics \ terms	a	an	and	ant	Allen	anchor	...	zebra
	.15	-.03	.02	-.01	.07	.02	...	-.03
	-.04	.01	.04	.01	.01	-.03	...	.12
	.12	.02	-.02	.01	.01	.01	...	-.02
	.04	-.05	-.04	-.03	.01	.01	...	-.03
	-.02	.02	-.05	-.01	.04	.05	...	.04
	.06	-.05	.01	.08	-.01	-.01	...	-.02
	.16	-.01	-.04	-.03	-.02	-.02	...	-.04
	.06	-.05	.02	.05	-.04	.02	...	-.03
	-.02	.02	-.02	-.01	-.03	.05	...	.02
	.06	.05	-.01	-.04	-.05	-.04	...	-.03
	.14	-.01	-.05	-.03	-.01	.02	...	-.04
	.06	-.05	.1	-.05	-.01	.05	...	.02

Figure 4.4 Term-Topic Matrix for LSA model with 20 Dimensions

The weights we assign to term-topic pairs in this term-topic matrix will look like those for the topic-document matrix, only much smaller, because there are so many more of them. The columns of the term-topic matrix are sometimes called the "word vectors" of an LSA language model.

So how can we create these automatically? Singular Value Decomposition, from your linear algebra class, is what LSA uses to create these matrices.

4.3 Singular Value Decomposition (SVD)

SVD is an algorithm for decomposing a matrix into three "factors", three matrices that can be multiplied together to recreate the original matrix. This is analogous to finding the integer factors that can be multiplied to "recreate" the integer that you factored, only we need exactly three factors for our matrix of real numbers. But our factors aren't scalar integers, they are 2-D matrices. And there can be any number of integer factors and we are looking for only 3 very specific matrix factors. Our matrix factors contain real numbers (`floats`) rather than integers. But the three matrix factors we compute with SVD have some useful mathematical properties we can exploit for dimension reduction and LSA.

We're going to run this SVD algorithm on a term-document matrix. Or, even better, we can run it on the normalized TF-IDF term-document matrix. In both cases, SVD will help us find combinations of words that belong together, because they occur together a lot. SVD finds those co-occurring words by calculating the correlation between the columns (terms) of our term-document matrix, which is just the square root of the dot product of two columns (term-document occurrence vectors).

SVD will group those terms together that have high correlation (because they occur in the same documents together a lot). And we can then think of these collections of words as "topics" and this will help us turn our bags of words (or TFIDF 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, or at least what the Bag of Words is about.

It's unclear who came up with the idea to apply SVD to words. 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 (co-occurrence correlation). These researchers included Harris, Charles, Miller, Firth and Wittgenstein.

One last example to convince you of the usefulness of word cooccurrence. Can you figure out what "awas" means from its context in this statement?

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 and is very territorial, sometimes becoming dangerously aggressive. And your internal natural language processor may not have time to consciously figure out what "awas" meant until after you

are safely aboard the retreating boat. But when you thought about it, you'd probably guess that "awas" means "danger" in Indonesian. Ignoring all the real world context and just focusing on the language context, you can often "transfer" a lot of the significance or meaning of words you do know to words that you don't, in any statement or document. Try it sometime, with yourself or with a friend. Like a Mad Libs game, just replace a word in a sentence with a made up or foreign word and ask a friend to guess what that word means or fill in the blank with an English word. Often the guess won't be too far off from a valid translation of the foreign word.

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 words mean. But they can do it quite well, using SVD, 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 even entire 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 at the beginning and end of a longer document. So we are going to train a machine to recognize the meaning (semantics) of words and phrases by giving it 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.

Here's what that looks like in math notation.

Wm x n Um x p Sp x p Vp x 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 matrices look like for our example matrices above.

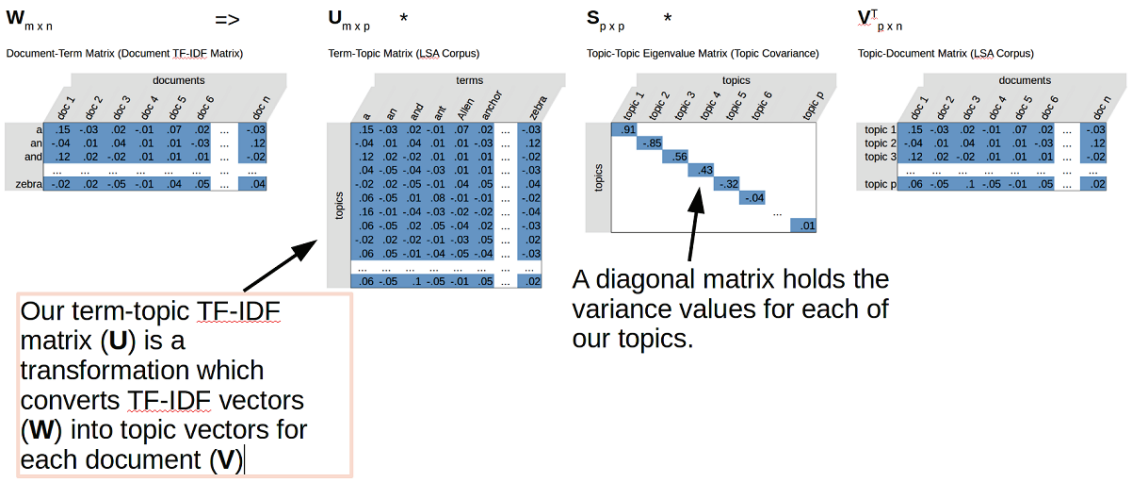


Figure 4.5 W_mxn U_mxp * S_pxp * V_pxnT

If you've used Scikit-Learn before, and were thinking about how this might fit into a training set or test set for a machine learning model, don't get confused by the orientation of our document-topic matrix. You'll want to transpose this matrix to create a training set for input into a SciKit Learn model. Each row in a training set matrix should be a feature vector for a particular example, a document in our case.[48] We'll show you that when we train a `TruncatedSVD` transformer and then train a SciKit Learn classifier on those document topic vectors.

Footnote 48 SciKit Learn documentation on LSA:
scikit-learn.org/stable/modules/decomposition.html#lsa

4.4 Truncated SVD

We've managed to create topic vectors for all our documents and words, but it has just as many dimensions as the original BOW vectors. We haven't reduced the number of dimensions… yet.

Let's make sure we understand what each of those matrices represent and see if there are some dimensions (topics) we can ignore.

The first matrix factor, *U*, contains all the topic vectors for each word in our corpus as columns. You can think of this as a transformation that can convert a TF-IDF vector into a topic vector.

SVD ensures that all the rows of this matrix have been normalized so they have a length of 1. This means our transformation from word frequencies to topics will only rotate the TFIDF vectors, it won't stretch or scale them in any way.

The diagonal *S* matrix in the middle contains the eigenvalues or singular values of our decomposed matrix. It can also be thought of as the variance of each topic. It tells us the "explained variance" that each of the topics contributes to the spread of the document TF-IDF vectors around in the vector space of term frequencies. So a larger value says that the topic associated with that column or row is really important to explaining the choice of words for a given document in our corpus. That topic is used in a lot of documents distributed throughout our vector space. So, if we want to reduce dimensions

without reducing information and "meaning" in our topic vectors we better hang on to the topics that have large eigenvalues.

That's what we'll do. We'll act like all the small eigenvalues (singular values) are actually zero and ignore them. We'll lose a little bit of subtlety in our word choices for some documents, but these aspects or dimensions of the documents aren't particularly interesting. They don't contain much information. We don't lose much if we ignore them. We'll show an example 3D object that makes this more visual and easier to understand in the next section.

The last matrix factor, *V*, contains all the topic vectors for each document as columns (before it's transposed to make the inner product work). This is the thing we want to be able to compute from a set of TF-IDF vectors, like the *W* matrix for our corpus on the far left. This is the "answer", the topic vector for each document in our corpus, but we also want to be able to compute it on a new bag of words or TF-IDF vector. So we don't need to record this matrix. Instead we can compute it anew whenever we need it. To create a row in this matrix all we need to do is multiply the inverse of the *V* matrix (or transpose, in our case) by any new TF-IDF vector to get the normalized topic vector for the document in that vector.

We can ignore the *S* matrix as well, once we've used it to determine the most important topics in our topic vector. The trick is to recognize what that middle diagonal matrix, *S*, represents. If you've ever done SVD on a set of 3D vectors, like for the point cloud of a physical 3D object below, you may have developed the intuition that this represents the "scaling" part of the linear transformation. It turns out, we don't need this "scaling" for documents. This just increases (or decreases) the length of our topic vectors, which we want to normalize to unit length (2-norm of 1) anyway. So we can safely ignore these values. In fact it actually improves the accuracy or our model if we just ignore them (assume they are all 1).[49]

Footnote 49 Levy, Goldberg and Dagan, Improving Distributional Similarity with Lesson Learn from Word Embeddings, 2015

Of course we need the singular values to find the largest ones which we used to select the topics that are most important to the variance ("spread") of our documents in our new topic vector space. This is the "Truncated" part of *Truncated SVD*, the algorithm used under the hood within `sklearn.decomposition.PCA`. We simply ignore all the columns (topics) of the *U* matrix that don't represent much of the "explained variance" in our corpus. This is how we end up with whatever number of topics we select when we configure a `TruncatedSVD` model in scikilearn or LsiModel in gensim. But after that selection has been made, we should set all our singular values (topic variances) to 1 or just ignore them and use the unscaled *U* matrix to multiply by TF-IDF vectors when we want to convert them to topic vectors.

4.5 SciKit-Learn vs gensim

We'll show you two different ways to implement LSA.

- `sklearn.decomposition.PCA`
- `gensim.models.LsiModel`

When considering which one to, keep in mind that gensim has the advantage that it doesn't require increasing amounts of RAM to deal with a growing vocabulary or set of documents. You can train your LSA model (`LsiModel`) and merge the models later. All of gensim's NLP models are designed to be "constant RAM" implementations.

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. This is called "out of core" computation. So packages like `gensim` are worth having in your toolbox. It can speed up your small-data experiments like in this book, and also power your hyperspace travel on Big Data in the future. But first we'll start with scikit learn, because of it's familiar interface. We'll use it on small datasets, so the parallelization and memory efficiency of gensim won't be necessary, just yet.

Here's the set of vectors we'll start with. We're only dealing with 3 dimensions at first, so it's straightforward to plot them using the `Axes3D` class in matplotlib. See the `nlpia` package for the code to create rotatable 3D plots like this.

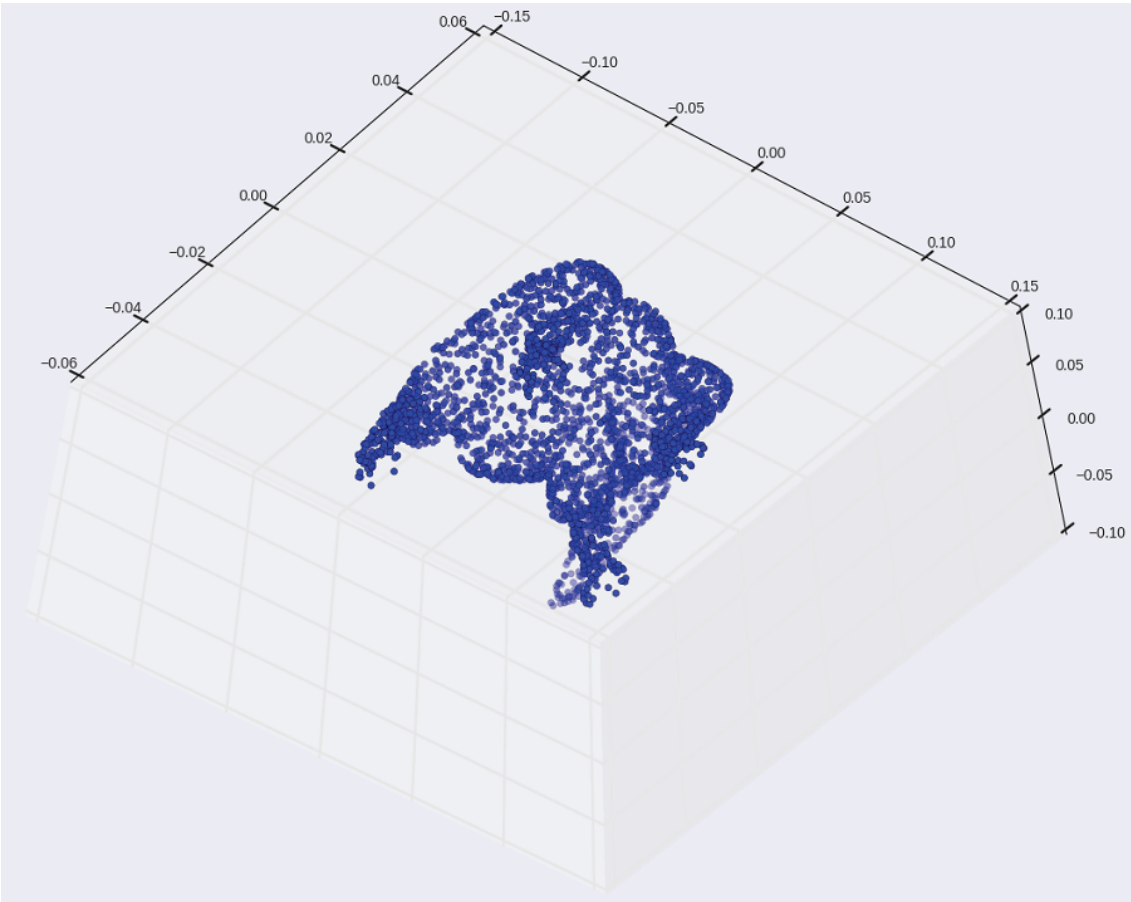


Figure 4.6 Looking up from below the "belly" at the point cloud for a real object

In fact, the point cloud shown above is from the 3D scan of the surface of a real-world object, not a set of BOW vectors. This will help you get a feel for how LSA

(PCA) works. One of the key things to notice is the rotation that maximizes the variance along the x axis and the second highes variance along the y axis. Can you guess what this object is, from this random projection from 3-D into 2-D for printing in this book?

4.5.1 PCA on a Point Cloud

Well I manually rotated the point cloud into this particular orientation to *minimize* the variance along the axes of the window for the plot, so that you'd have a hard time recognizing what it is. This will help you see how PCA preserves the structure, information content, of a point cloud (or set of vectors) more optimally for machine learning. It *maximizes* the variance along each axis.

```
>>> import pandas as pd
>>> pd.set_option('display.max_columns', 6)  # ❶
>>> from sklearn.decomposition import PCA
>>> import seaborn
>>> from matplotlib import pyplot as plt
>>> from nlpia.data import get_data

>>> df = get_data('pointcloud').sample(1000)
>>> pca = PCA(n_components=2)  # ❷
>>> df2d = pd.DataFrame(pca.fit_transform(df), columns=list('xy'))
>>> df2d.plot(kind='scatter', x='x', y='y')
>>> plt.show()
```

❶ Ensure that our `pd.DataFrame`s fit within the width of a page when printed

❷ We're reducing a 3-D point cloud to a 2-D "projection" for display in a 2-D scatter plot

If you run the script above, the orientation of your 2-D projection may "flip" left to right, but it never tips or twists to a new angle. The orientation of the 2-D projection is computed so that the maximum variance is always aligned with the x axis. The 2nd largest variance is always aligned with the y axis. 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 2-D projection

of the 3-D horse provide a good view of the point cloud data? Wouldn't a machine to be able to learn something from the statistics of these 2-D vectors computed from the 3-D vectoers of the surface of a horse?

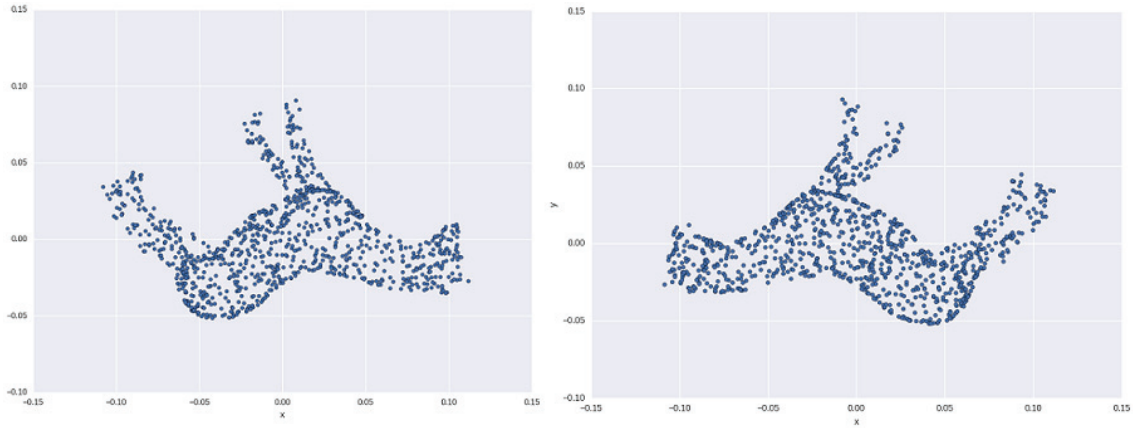


Figure 4.7 Head to Head Horse Point Clouds Upside Down

4.5.2 Let's Stop Horsing Around and Get Back to NLP

Let's see how SVD will do on some natural language documents. Let's find these same principle components using SVD for a set of about 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 SciKit-Learn's 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. The PCA model is design to provide a more exact solution using dense matrices. The `TfidfVectorizer` in scikit learn outputs sparse matrices so that's the only code change we'll have to make between the two models.

First, let's load the SMS messages from a `DataFrame` in the `nlpia` package and then calculate the TF-IDF topic-document matrix.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> from nltk.tokenize.casual import casual_tokenize
>>> from nlpia.data import get_data

>>> sms = get_data('sms-spam')
>>> 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)
>>> 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...
```

❶

❶ We've flagged SPAM messages by appending an exclamation point, "!", to their

label.

Now we can calculate the TF-IDF vectors for each of these messages.

```
>>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray()
>>> tfidf_docs = pd.DataFrame(tfidf_docs, index=index)
>>> tfidf_docs = tfidf_docs - tfidf_docs.mean()
>>> tfidf_docs.shape
(4837, 9232)
>>> sms.spam.sum()
638
```

So we have nearly 5000 SMS messages with about 9000 different 1-gram tokens (using the `casual_tokenize`r). And only about 600 of these 5000 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).

However, the vocabulary size, |V|, with its 9232 tokens is even more problematic. We have more unique words in our lexicon (vocabulary) than we have SMS messages, and many more words than messages labeled as SPAM. That's a recipe for overfitting.

So some dimension reduction/consolidation is definitely in order. That's exactly what LSA is for. LSA will reduce our dimensions and thus reduce overfitting. Plus it will help generalize our NLP pipeline so we can apply to more general SMS messages instead of just this particular set of messages.

4.5.3 PCA on SMS Messages

Let's try the `PCA` model from SciKit-Learn first. You've already seen it in action wrangling 3-D horses into a 2-D 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()
       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   -0.020
```

4.5.4 Truncated SVD

Now we can try the SVD model in scikit-learn. It can handle sparse matrices, so if you're working with large datasets you'll want to use `TruncatedSVD` instead of `PCA`. into 3 matrices that we can use to transform any TF-IDF vector into a topic vector. And we can discard the dimensions (truncate these matrices) so that we only retain 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_transpose 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 frequecies 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? No cheating by looking at the "!" label. 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.

4.5.5 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 will 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   ❶
>>> svd_topic_vectors.iloc[:10].dot(svd_topic_vectors.iloc[:10].T).round(1)
       sms0  sms1  sms2!  sms3  sms4  sms5!
sms0    1.0   0.6   -0.1   0.6  -0.0   -0.3
sms1    0.6   1.0   -0.2   0.8  -0.2    0.0
sms2!  -0.1  -0.2    1.0  -0.2   0.1    0.4
sms3    0.6   0.8   -0.2   1.0  -0.2   -0.3
sms4   -0.0  -0.2    0.1  -0.2   1.0    0.2
sms5!  -0.3   0.0    0.4  -0.3   0.2    1.0
```

❶ Normalizing each topic vector by it's length (L2-norm) simplifies the cosine similarity computation into the dot product.

Looking at the cosine similarity valuesReading down the "sms0" column, the cosine similarity between this non-SPAM message and the other non-SPAM messages is significantly positive (except for "sms4" which is only slightly negative). And the the cosine similarity between "sms0" and the SPAM messages ("sms2!" and "sms5!") is significantly negative.

And this is how semantic search works as well. You can use the distances between a query vector and all the topic vectors for your database of documents to approximate the sematica similarity. 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 that 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)

Normalizing 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 in it's exactly implementation of Truncated SVD.

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 LSA, PCA, or SVD implementation you use, be sure that the approach discards the eigenvalues (the diagonal `Sigma` matrix in the middle of the SVD product of 3 matrices). Alternatively, just normalize whatever BOW or TF-IDF vectors you pass into your semantic analysis algorithm. Otherwise the eigenvalues can create large scale differences between your topics. Scale differences between topics usually reduce the ability of your model to differentiate between subtle, infrequent topics.

4.5.6 A Simple Classifier

Linear Discriminant Analysis (LDA) is one of the most straightforward and fastest multidimensional classifiers you'll find. It just computes the the average position of all the vectors in each class (like our SMS messages labeled spam and nonspam). These two locations are called the centroids of those point clouds for each class. Any vector that is closer to the spam centroid than it is to the nonspam centroid is classified as spam. All the others are classified as "ham" (nonspam).

Before using LDA to discriminate between SPAM and non-SPAM, let's fit a 16-D PCA topic vector model to our SMS messages first.

```
>>> tfidf = TfidfVectorizer(tokenizer=casual_tokenize)
>>> tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray()
>>> pca = PCA(n_components=16)
>>> pca16_topic_vectors = pca.fit_transform(tfidf_docs)
```

Now we can train our new LDA model to classify SPAM based on the centroids of our 16-D topic vectors for each SMS message (document).

```
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

>>> X_train, X_test, y_train, y_test = train_test_split(pca16_topic_vectors, sms.spam,
    test_size=0.5, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(X_train, y_train)
>>> sms['pca16_spam'] = lda.predict(pca16_topic_vectors)
>>> round(float(lda.score(X_test, y_test)), 3)
0.963    (1)
```

(1) That's really incredible accuracy! 96% of the messages were classified correctly.

We only had to train our supervised LDA model on half of the SMS messages and it could figure out the right classification for the other half 96% of the time! That's the power of LSA (PCA). You can compute topic vectors without any labels on your data. And then they help you extrapolate from a smaller number of labeled examples to all the others in your data set. That's a really awesome thing!

And, as a stress test, why not try a few of your own SMS SPAM or non-SPAM messages in this spartan classifier to see if it can generalize across an ocean of cultural differences. Our small training set is from a single European university research lab with very few students. So decent performance on your SMS messages in the US would be quite an impressive "generalization" from this small, focused training set.

And it's only going to get better from here on out. In later chapters we'll show you how only a few dozen labeled examples are necessary to classify things like job titles. But you have to train your language model on millions of example documents so your unsupervised model can learn what words "mean."

4.6 Latent Dirichlet Allocation (LDiA)

Latent Dirichlet Allocation creates a semantic vector space model (like our topic vectors) using an approach similar the way we discussed at the beginning of the chapter in our thought experiment. 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, where all words are part of all topics, to some degree.

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 it name.

4.6.1 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.[50] Stanford Researchers (including Andrew Ng) popularized the approach for NLP in 2003 and .[51] But don't be intimidated by the big names that came up with this approach. I'll explain the keep points of it in a few lines of Python below. But 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 pipline.

Footnote 50 "Jonathan K. Pritchard, Matthew Stephens, Peter Donnelly, Inference of Population Structure Using Multilocus Genotype Data" www.genetics.org/content/155/2/945

Footnote 51 www.jmlr.org/papers/volume3/blei03a/blei03a.pdf

Blei and Ng came up with the idea by flipping our thought experiment on it's 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 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 to match the distribution of these values in real documents analyzed by humans for their topics and words.

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 would just iterate over those topics and randomly chose words appropriate to that topic until it hit 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, the imaginary 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 cartoon example earlier in this chapter.

So all this machine needs is a single parameter for that Poisson distribution (mentioned above) 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 a term-topic matrix of all the words and topics it likes to use, or its vocabulary,. And it need 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 just analyzing the statistics of the documents in a corpus. For example, for step 1, they could just 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)
18.12
```

Or even more concisely:

```
>>> sum([len(casual_tokenize(t)) for t in sms.text]) * 1. / len(sms.text)
18.1...
```

Keep in mind, you should calculate this statistic directly from your BOWs, after you have tokenized and vectorized (`Counter()`-ed) your documents and done any stopword filtering, etc. That way your count includes all the words in your BOW vector vocabulary (any n-grams), but only those words that your BOWs use (no stopwords). This LDiA algorithm relies on the BOW vector space model (VSM) of natural language text, like all the other algorithms in this chapter.

The second parameter is a bit trickier. The number of topics in a particular set of documents can't be measured until after you've assigned words to those topics. So, like *K-Means* and *KNN* and other clustering algorithms you must must tell it how many topics to compute. 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 and whatever NLP problem you want to solve. 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 in the corpus)[52] by adjusting it until it works for your application. You can automate this optimization if you can measure something about the quality of your LiDA language model for representing the meaning of your documents in your domain. One "cost function" you could use for this optimization is how well (or poorly) that LiDA model performs in some classification or regression problem (like sentiment analysis, document keyword tagging, or topic analysis) for which you have labeled documents.

Footnote 52 The symbol used by Blei and Ng for this parameter was actually *Theta* rather than *K*

4.6.2 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 together that started out close together.

Let's see how that works for a data set of a few thousand SMS messages, labeled for spaminess. Now let's compute the TF-IDF vectors and then some topics vectors for each SMS message (document). And let's 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.[53]

Footnote 53 See Appendix D if you want to learn more about why overfitting is a bad thing

LDiA works best with raw BOW count vectors rather than normalized TF-IDF vectors. Here's how to compute those in SciKit-Learn

```
>>> 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 amore wat...'
>>> bow_docs.loc['sms0'][bow_docs.loc['sms0'] > 0].head()
,            1
..           1
...          2
amore        1
available    1
Name: sms0, dtype: int64
```

And here's the Dirichlet Allocation (LDiA) topics allocated 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 9232 unique 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

```
>>> components = pd.DataFrame(ldia.components_.T, index=terms, columns=columns)
>>> components.round(2).head()
       topic0  topic1  topic2   ...     topic13  topic14  topic15
!       51.62   36.56    5.22   ...       16.65   450.05   100.99
"       18.17    9.60    0.06   ...        7.10     0.10     0.06
#        0.06    0.06    0.06   ...        0.06     0.06     0.06
#150     0.06    0.06    0.06   ...        1.06     0.06     0.06
#5000    0.06    0.06    0.06   ...        0.06     0.06     0.06
```

So an exclamation point was allocated to most of the topics, but is a particularly strong part of topic14 where the quote symbol is hardly playing a role at all. Perhaps "topic14" might be about emotional intensity or emphasis and doesn't care much about numbers or quotes. Let's see.

```
>>> components.topic14.sort_values(ascending=False)[:10]
you     980.696757
?       815.891903
```

```
i       679.981857
!       450.051780
to      432.918701
,       399.005200
.       278.308090
...     262.068378
are     252.820433
your    214.362627
```

So the top ten tokens for this topic seem to be the type of words that would be used in emphatic directives requesting someone do something or answer a question. It will be interesting to find out if this topic is used more in SPAM messages rather than non-SPAM messages. You can already see that the allocation of words to topics can be rationalized or reasoned about, even with this cursory look.

But 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.

```
>>> ldia16_topic_vectors = ldia.transform(bow_docs)
>>> ldia16_topic_vectors = pd.DataFrame(ldia16_topic_vectors, index=index, columns=columns)
>>> ldia16_topic_vectors.round(2).head()
       topic0  topic1  topic2   ...     topic13  topic14  topic15
sms0     0.00    0.00    0.00   ...        0.00     0.00     0.00
sms1     0.13    0.01    0.01   ...        0.01     0.13     0.01
sms2!    0.00    0.00    0.00   ...        0.00     0.00     0.68
sms3     0.00    0.00    0.00   ...        0.00     0.00     0.00
sms4     0.00    0.00    0.00   ...        0.00     0.20     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 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?

4.6.3 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).

```
>>> X_train, X_test, y_train, y_test = train_test_split(ldia16_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['ldia16_spam'] = lda.predict(ldia16_topic_vectors)
>>> round(float(lda.score(X_test, y_test)), 3)
0.901                                                  (2)
```

(1) 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.

❷ 90.1% accuracy on the test set is pretty good, but not quite as good as LSA (PCA) from before.

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 LDA 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.

We're 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 "underdetermined" 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.

```
>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs, sms.spam,
    test_size=0.5, random_state=271828)
>>> lda = LDA(n_components=1)
>>> lda = lda.fit(X_train, y_train)     ❶
>>> round(float(lda.score(X_train, y_train)), 3)
1.0
>>> round(float(lda.score(X_test, y_test)), 3)
0.829
```

❶ 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 for our LDiA-based topic model.

And test set accuracy is the only accuracy that really counts. This is exactly what topic modeling (LSA and LDiA) 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).

4.6.4 A Fairer Comparison: 32 LDiA Topics

Let's try one more time. Perhaps LDiA isn't as efficient as LSA (PCA). 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)
>>> X_train.shape                                          (1)
(2418, 32)
>>> round(float(lda.score(X_train, y_train)), 3)
0.924
>>> round(float(lda.score(X_test, y_test)), 3)
0.927                                                      (2)
```

(1) .shape is another way to check the number of dimensions in our topic vectors

(2) 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.

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 our 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 `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??`

4.7 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 similarities and distances before in the higher-dimensional space. 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 our algorithm produces 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 (have similar subject matter) are close to each other in meaning are also close to each other in our new topic space.

We should see that 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 for that matter. So what are our options for measuring distance in high-dimensional space, and which one should we chose for NLP? 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, Cartesian, root mean square error (RMSE): 2-norm or L2
- Squared Euclidean, sum of squared distance (SSD): L22
- Cosine (angular, projection): normalized dot product
- Minkowski: p-norm or Lp
- Fractional, fractional norm: p-norm or Lp for `0 < p < 1`
- City block, Manhattan, taxicab, sum of absolute distance (SAD): 1-norm or L1
- Jaccard, inverse set similarity,
- Mahalanobis
- Levenshtein (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.[54] For reference, here are all the distance metrics you can find in the `sklearn.metrics.pairwise` module:[55]

Footnote 54 other distance metrics: numerics.mathdotnet.com/Distance.html

Footnote 55 docs for sklearn.metrics.pairwise: scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html

'cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan', 'braycurtis', 'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', '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 that range between 0 and 1, like similarity scores or probabilities, it's more common to use a formula like this:

```
>>> similarity = 1. - distance
>>> distance = 1. - similarity
```

And cosine distances have their own convention. The angular distance between two vectors is computed as a fraction of the maximum possible distance (180 deg or pi radians).[56] As a result angular distances are actual radians of angle between two vectors.

Footnote 56 en.wikipedia.org/wiki/Cosine_similarity

```
>>> import math
>>> angular_distance = math.acos(cosine_similarity) / math.pi
>>> distance = 1. / similarity - 1.
>>> similarity = 1. - distance
```

IMPORTANT

Definition

The terms *distance* and *length* are often confused the term *metric* because many *distances* and *lengths* are valid and useful *metrics*. But unfortunately not all distances qualify to be called *metrics*, in the mathematics, and set theory. To make matters worse, *metrics* are also sometimes called *distance functions* or *distance_metrics*. A true *metric* must obey four properties that distances or scores do not.[57]

Footnote 57 en.wikipedia.org/wiki/Metric_(mathematics)

1. nonnegativity: *metrics* can never be negative
2. indiscerniblity: if a *metric* between two objects is zero then they must be identical
3. symmetry: the *metric* from A to B equals the metric from B to A
4. triangle inequality: the *metric* A to C is no larger than the *metrics* A to B plus B to C

IMPORTANT

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.[58]

Footnote 58 en.wikipedia.org/wiki/Measure_(mathematics)

4.8 Steering

"Learned distance metrics"[59] 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 59 users.cecs.anu.edu.au/~sgould/papers/eccv14-spgraph.pdf

At Talentpair we were 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.

4.8.1 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 that 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)
>>> df['lda_spaminess'] = lda.predict(tfidf_docs)
>>> ((df.spam - df.lda_spaminess) ** 2.).sum() ** .5
0.0
>>> (df.spam == df.lda_spaminess).sum()
4837
>>> len(df)
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, df.spam, cv=5)
>>> "Accuracy: {:.2f} (+/-{:.2f}".format(scores.mean(), scores.std() * 2)
'Accuracy: .96 (+/-.01)''
```

Or what about retaining 1/3 of the data for validation.

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_docs, df.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)
0.965
```

But 97% accuracy still sounds fishy. Remember that imbalance we had in the dataset? Only 13% of the SMS messages were spam. So we need to penalize our score more for mislabeling spam messages (false negatives) than for getting normal messages wrong (false positives).

But before we do that, let's see if LSA combined with LDA will help us create an accurate model that is also generalized well (so that new words and new SMS messages don't trip it up).

```
>>> X_train, X_test, y_train, y_test = train_test_split(pca_topicvectors.values, df.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, df.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.

4.9 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 limitted 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."[60] 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. And topic vectors provide a way for computers to tell the difference between pythons in a Florida pet shop aquarium, and a `Python` package in "The Cheese Shop" and yet still recodnize these Python packages similarity to a `ruby gem`.

Footnote 60 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.

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. The table below shows what happens when you try a "divide and conquer" approach to search among a set of high dimensional vectors. As Geoffry Hinton says, "To deal with hyper-planes in a 14-dimensional space, visualize a 3-D 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 3-D world into hyperspace, enough to catch a glimpse of that 3-D world from the outside. Like in *Flatland*, we'll use a lot of 2-D visualizations in this chapter to help you explore the shadows that words in hyperspace leave in our 3-D 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 3-D 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 2-D to 3-D and exponentially greater than the growth in complexity from a 1-D (scalar) world of numbers to a 2-D world of triangles, squares, and circles.

NOTE That explosive grown in possibilities from 1-D lines to 2-D ellipses and rectangles and on to a 3-D world of spheres and cubes 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 2-D plane while having less than two dimensions![61] But fortunately these aren't "real" dimensions. [62] So we don't have to worry about them in NLP, unless they somehow help you understand fractional distances (`p-norm`) which have noninteger exponents in their formula.

Footnote 61 fractional dimensions,
www.math.cornell.edu/~erin/docs/research-talk.pdf

Footnote 62 "fractal dimensions",
q-what-are-fractional-dimensions-can-space-have-a-fractional-dimension/

4.9.1 Semantic Search

One of the promises of "Latent Semantic Indexing" is that you can find a document which matches your query semantically. Unfortunately, the "indexing" part of LSI is not realistic. It's kind-of a misnomer. Perhaps that is why LSA has become the more popular term to describe this technique.

To find semantic matches, we'd need to search through our entire database of topic vectors for the best match. Unlike 2 and 3-D 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 (LSH) 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 we add to the Zip Code. But lets see where an LSH breaks down. Below we've 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 we 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 our search results cease to make much sense at all?

D	N	100th Cosine Distance	Top 1 Correct	Top 2 Correct	Top 10 Correct	Top 100 Correct
2	4254	0	TRUE	TRUE	TRUE	TRUE
3	7727	0.0003	TRUE	TRUE	TRUE	TRUE
4	12198	0.0028	TRUE	TRUE	TRUE	TRUE
5	9920	0.0143	TRUE	TRUE	TRUE	TRUE
6	11310	0.0166	TRUE	TRUE	TRUE	TRUE
7	12002	0.0246	TRUE	TRUE	TRUE	FALSE
8	11859	0.0334	TRUE	TRUE	TRUE	FALSE
9	6958	0.0378	TRUE	TRUE	TRUE	FALSE
10	5196	0.0513	TRUE	TRUE	FALSE	FALSE
11	3019	0.0695	TRUE	TRUE	TRUE	FALSE
12	12263	0.0606	TRUE	TRUE	FALSE	FALSE
13	1562	0.0871	TRUE	TRUE	FALSE	FALSE
14	733	0.1379	TRUE	FALSE	FALSE	FALSE
15	6350	0.1375	TRUE	TRUE	FALSE	FALSE
16	10980	0.0942	TRUE	TRUE	FALSE	FALSE

Figure 4.8 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 algorith, 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've chosen to use for our chatbot. In addition to it being the workhorse for finding matches among vectors representing song metadata for music fans, Dark Horse Comics is also using `annoy` to suggest comic books efficiently. We'll talk more about these tools in the chapter on scaling towards the end of this book.

4.10 "Like" Prediction

So this is what a collection of tweets looks like in hyperspace. Or, more accurately, these are the 2-D shadows of 100-D tweet topic vectors (points). The green marks represent tweets that were favorited at least once, while the red marks are for tweets that received zero "likes."

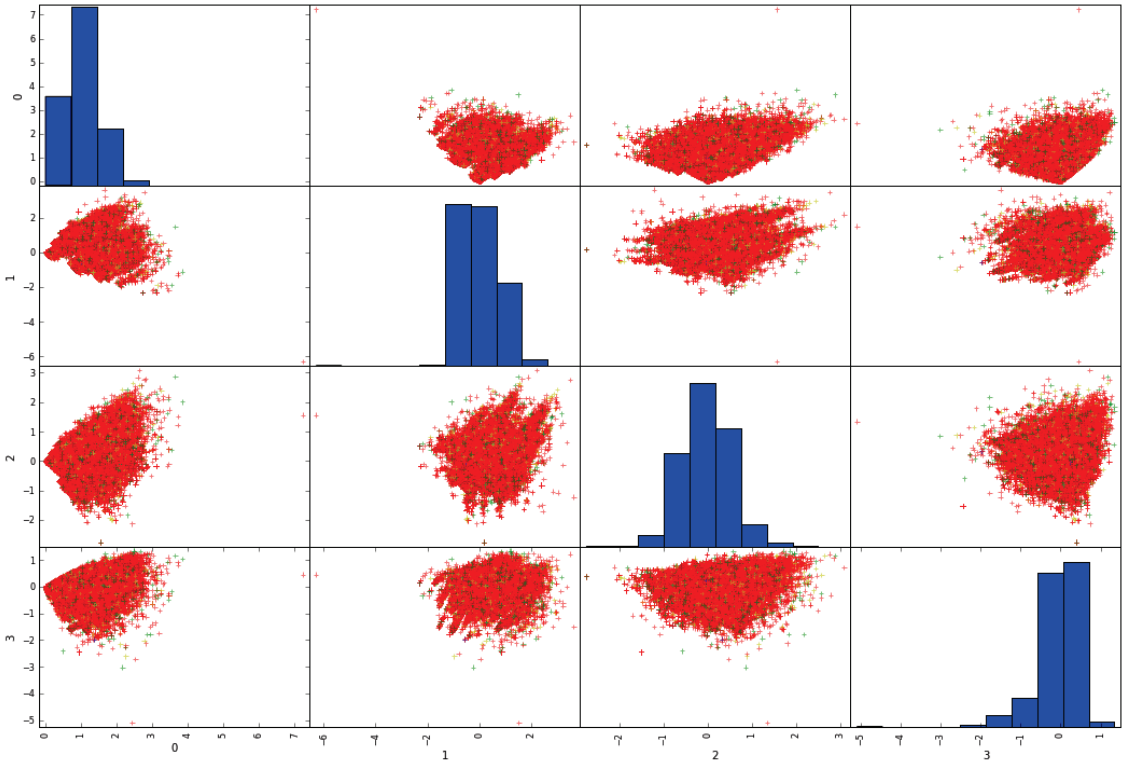


Figure 4.9 Scatter Matrix of 4 topics for Tweets

An LDA model fit to these topic vectors will succeed 80% of the time. However, like our SMS dataset, our 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 our main language modeling tools, we should probably stick to semantic matching and more straight-forward classification problems, like spam detection.

4.11 Summary

In this chapter you've learned how to

- Compress TF-IDF vectors into topic vectors to make them them more compact and more meaningful
- Search for text semantically—find documents or statements based on their meaning
- Classify and cluster texts based on their meaning
- Predict things like whether a text will be "liked" by readers or if it is likely to be spam
- How to use semantic search to as part of a dialog engine (chatbot)

In the next chapter, 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 have to start learning about neural nets. This will improve your pipeline's ability to work with short texts or even single words.

5 Baby Steps with Neural Networks

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, *backpropagaion*, 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.

5.1 A Regression Digression

The fundamental unit of a neural network is the neuron, and the simplest kind of neuron is nothing more than a linear "gain" or slope multiplied by the input. And, like a linear regression, most neural networks add a bias or intercept. And a neuron can learn the right slope and intercept by processing a set of examples in the same way that you computed the slope and intercept of a set of points for a linear regression in Calculus or Algebra class.

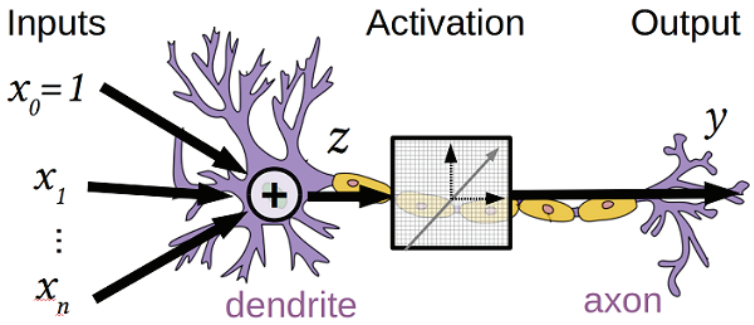


Figure 5.1 An artificial neuron is an abstraction of a biological neuron

All alone a single neuron is no more powerful than the *closed form* linear regression you learned in High School. But when working in concert with other neurons in a single "layer" or connected to additional neurons in a "deep learning" network, these neurons can solve some machine learning problems that were unsolvable before. But first we'll show you how this single neuron works by performing a linear regression the traditional way and then using a single neuron to accomplish the same thing.

TIP A *closed form* solution provides the coefficients of a model by evaluating algebraic expressions. It doesn't require guessing and incrementally improving an approximate numerical solution. A closed form solution is always preferable because it will give you the most accurate answer with the least computation possible. And it will always "converge" to the correct solution. Remember the "solve for x" problems in algebra? A computer can compute these expressions in an instant. However, if we don't know the equation is that contains the "x" we're interested in, if we don't know the form of the "model" of our problem, then there isn't a closed form solution to be found. And that is where neural networks shine, when you have no idea what the relationship is between your inputs and outputs.

Let's revisit the SMS spam example from Chapter 4 and model the relationship between one of the topics in our LSA topic vector and the "sentiment" (positivity of emotion) in those SMS messages. Perhaps the "spamminess" topic is linearly proportional to the positivity or intensity of emotion in the SMS messages. We'll use the VADER sentiment analysis algorithm from Chapter 2 to score the positivity of SMS messages and compare it to one of the spaminess topics from the LSA topic vectors for

those messages. The positive sentiment score will be our target or output variable and our input will be the spaminess topic.

We'll show you how to fit a linear model too this data the "old-fashioned way", using the closed-form equations you learned in Algebra or Calculus class. Then we'll show you how to get the same answer using a incremental numerical solver in scikit learn, the SGDRegressor. Finally we'll build a simple single-neuron neural net from scratch in python. That should help show how it get's the same answer as well. Once you've mastered single-neuron regression, the next section will show you how to do single-neuron classification. And then we'll build up increasingly complicated neural networks from these neurons to solve more complicated problems in the following chapters.

5.1.1 The Spaminess Topic

First lets get the SMS data and compute the TFIDF vector for each messagesome 256-dimension topic vectors so we can build a model based on one of those topics as our input.

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from nltk.tokenize.casual import casual_tokenize
from nlpia.data import get_data

sms = get_data('sms-spam')
tfidf = TfidfVectorizer(tokenizer=casual_tokenize)
tfidf_docs = tfidf.fit_transform(raw_documents=sms.text).toarray()
tfidf_docs = pd.DataFrame(tfidf_docs, columns=list(zip(*sorted([(v, k)
    for (k, v) in tfidf.vocabulary_.items()])))[1])
```

We can fit an LDA model to the TFIDF vectors to find the spamiest words in our SMS messages.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

# TFIDF->LDA
tfidf_lda = LDA(n_components=1)
tfidf_lda.fit(tfidf_docs, sms.spam)
# Most spammy terms (words) in the TFIDF:
tfidf_lda_coef = pd.DataFrame(list(zip(tfidf_lda.coef_[0, :], tfidf_docs.columns)),
    columns='coef term'.split())
print(tfidf_lda_coef.sort_values('coef', ascending=False).head())
#               coef                term
# 2666  7.606693e+06             darling
# 7168  5.393772e+06             sexychat
# 895   5.202198e+06                80488
# 6085  4.865422e+06              parties
# 9025  4.852177e+06  www.07781482378.com
```

This is interesting and may give us some URLs and words to be wary of, but we'd like to a regression on something a little less discrete. Specific "red flag" words occur rarely, and it would be hard to tell if a linear model based on the TFIDF for one of these words as our input feature (predictor variable) was a good fit or not. We need some

combination of words to help us gage the spaminess as a continuous score. That's what LSA topics are great for. And it's a lot faster to do this math on 256 topics rather than all 10,000 or so word frequencies.

```
from sklearn.decomposition import PCA

# TFIDF->PCA->LDA
pca = PCA(n_components=256)
pca = pca.fit(tfidf_docs)
pca_topic_vectors = pca.transform(tfidf_docs)
pca_topic_vectors = pd.DataFrame(pca_topic_vectors, columns=['topic{}'.format(i)
                                 for i in range(pca.n_components_)])
pca_components = pd.DataFrame(pca.components_,
                                 columns=tfidf_docs.columns,
                                 index=['topic{}'.format(i)
                                 for i in range(pca.n_components_)])
```

Now let's build that LDA model again using these more continuous topics instead of discrete words. We'll use it to find find the topic among these 256 components that is most correlated with the spam label for our SMS messages.

```
pca_lda = LDA(n_components=1)
pca_lda.fit(pca_topic_vectors, sms.spam)
sms['pca_lda_spam_prob'] = pca_lda.predict_proba(pca_topic_vectors)[:, 1]
```

The code above are available in the `ch05_sms_spam_linear_regression.py` script in the nlpia repository within the scripts directory. If any of this doesn't look familiar, you may want to review Chapter 4.

So let's find the most spammy topic among the topic vectors we produced for these SMS messages and use that. This topic will combine the frequencies of many words to give us a continuous spaminess score. The topic with the highest correlation with the spam/ham label should be a good one for us to use for our 1-D linear model to predict.

```
pca_topic_vectors['spam_label_'] = sms.spam
print(pca_topic_vectors.corr().spam_label_.sort_values(ascending=False).head())
# spam_label_    1.000000
# topic4         0.564850
# topic2         0.275897
# topic10        0.186002
# topic9         0.167077
```

So `topic4` is 56% correlated with the spaminess of our text messages. That should be strong enough so that if sentiment is also proportional to this topic we can say that sentiment is partly responsible for the spam probability or spaminess score that we're computing with our topic model. Let's add that to our SMS `DataFrame` so we can use it later in our model.

```
sms['topic4'] = pca_topic_vectors.topic4
```

THE SENTIMENT OF SPAM

Let's compute the sentiment (positive emotion) of the SMS Messages using VADER and make sure it's correlated with this spaminess topic.

```
>>> from nltk.sentiment import SentimentIntensityAnalyzer

>>> vader = SentimentIntensityAnalyzer()
>>> scores = pd.DataFrame([vader.polarity_scores(text) for text in sms.text])
>>> sms['vader'] = scores['compound']
>>> sms.describe().tail()
     spam  pca_lda_spam_prob   vader
min   0.0       5.845537e-15 -0.9042
25%   0.0       2.155922e-09  0.0000
50%   0.0       2.822494e-08  0.0000
75%   0.0       1.172246e-06  0.5267
max   1.0       1.000000e+00  1.0000
```

The VADER compound score is +1 for strong positive emotion and -1 for strong negative emotion. The more negative or positive this score is, the more intense the emotion being expressed in that text. But it looks like most of our messages have a zero VADER sentiment score. Those SMS messages with a zero VADER score don't contain any words that VADER knows about. We haven't done any stemming or lemmatization, so it's not a surprise since our SMS messages are from a graduate student lab in Europe and contain Japanese characters and foreign words. VADER is only designed to work with properly spell English words. Let's remove those messages that VADER can't score, as well as any without strong positive or negative emotion. This will help simplify our plots or and highlight any relaitonshipo between sentiment and spamminess.

```
mask = (sms.vader > 0.1) | (sms.vader < -0.1)
sms = sms[mask].copy()
```

And let's make sure all of our spaminess and sentiment data is normalized to have zero mean and unit standard deviation (`df.std() == 1.0`) so that each model start on the same footing.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
for col in ['pca_lda_spam_prob', 'vader', 'topic4']:
    sms.loc[:, col] = scaler.fit_transform(sms[[col]])
```

THE ALGEBRA APPROACH

Our first model of the positivity of spam is a conventional linear regression The spaminess score is our input feature and our VADER sentiment score as the output we want to predict. This will tell us if spam has positive or negative sentiment on average. If our model has a positive slope that tells us that spam is generally upbeat and happy and positive and uses words found in the VADER lexicon of happy words.

```
from nlpia.models import LinearRegressor
```

```
line = LinearRegressor()
line = line.fit(sms['topic4'], sms['vader'])
print('{:.4f}'.format(line.slope))
# 0.2934

sms['line'] = line.predict(sms['topic4'])
```

So there is a small but significant positive slope in our regression. This indicates that sales people use positive happy language when they want to get us to read and respond to their solicitations… at least in this limited data set.

If you'd like to brush up on the equations required to get a closed form solution to a linear regression problem, check out the source code in the `nlpia.models.LinearRegressor` class. Also it might help to review the Machine Learning introduction in Appendix D where we discuss linear regression in greater detail. We provide simpler examples there that don't rely on familiarity with NLP concepts like LSA and TFIDF.

TIP In an iPython console or Jupyter notebook, you can check out the source code of any function or class that you have imported by appending two question marks to the end of its name: `LinearRegressor??`

THE MACHINE LEARNING APPROACH

Let's fit an SGDRegressor to this data to see if it gives us the same answer that our algebra teacher taught us. Rather than simply computing sums and squares to get our answer, we use the machine learning approach of guessing the answer and then correcting it a bit at a time, iteratively. Lets have an SGDRegressor run through our training examples for 2000 iterations. That should get us pretty close to the exact answer we got above.

```
from sklearn.linear_model import SGDRegressor

sgd = SGDRegressor(n_iter=20000)
sgd = sgd.fit(sms[['topic4']], sms['vader'])
print('{:.4f}'.format(sgd.coef_[0]))
# 0.2936

sms['sgd'] = sgd.predict(sms[['topic4']])
```

The classical closed form algebra solution and the incremental numerical machine learning solution agree to better than 1%! Your answer may be slightly different from ours, due the S (stochastic) in the SGD algorithm. SGD uses a random number generator to perturb the gradient descient direction and magnitude to improve the robustness of the algorithm (ensure that it finds and answer close to the global optimum more often) We could run it for many more `n_iter` and with a smaller `learning_rate` or `alpha` if we wanted to get the exact same answer.

But what about a single neuron regressor. The SciKit-Learn Perceptron (simplest artificial neuron) implementation uses an SGDClassifier under the hood. And

Scikit-Learn even provides you with the initialization arguments to ensure that they both give you the same answer. And the SGDRegressor that we just used to fit a continuous linear model is what the SGDClassifier uses under the hood, before thresholding to determine the class label. In case that's not enough to convince you that a Neuron is merely a linear regressor, let's implement a single linear neuron from scratch and see what slope and intercept it produces.

THE NEURAL NET APPROACH

The `nlpia.models` module contains a single-neuron perceptron class called `OneNeuronRegress` with a linear "pass-through" activation function, to keep things simple. It fit a single weight (also called *gain* or *slope*) and a single bias (*intercept* or *offset* or W0) to our data to produce a sentiment score from our spaminess topic just like the SGDRegressor and LinearRegressor did above.

```
from nlpia.models import OneNeuronRegressor

nn = OneNeuronRegressor(alpha=100, n_iter=200)
nn = nn.fit(sms[['topic4']], sms['vader'])
print(nn.W[0, 1])
# 0.29386408

sms['neuron'] = nn.predict(sms[['topic4']])
```

So the only difference between our single neuron regressor and an SGDRegressor or closed form lienar regression is that we've computed the slope and intercept as part of the same `W1` (weight) matrix for the first "layer" of our single-layer neural net.

Here's a plot of all these lines and the underlying data. It's not pretty, but spam rarely is ;)

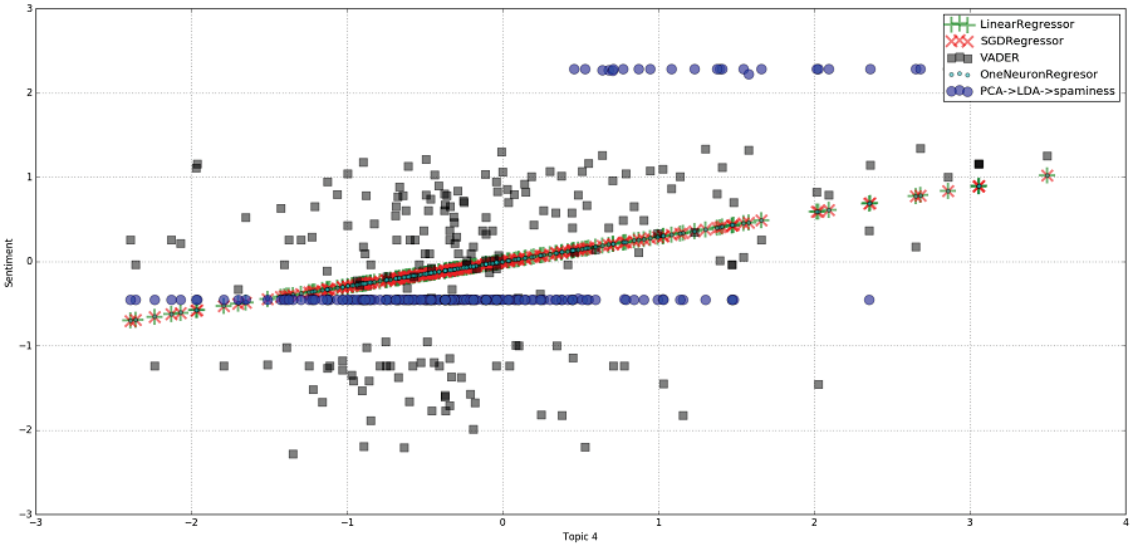


Figure 5.2 Comparison of 3 linear models of the sentiment in spam

5.2 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 Perceptron [63] offered a novel algorithm for finding patterns in data. 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 Papert [64] who showed the Perceptron was severly 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 63 Rosenblatt, Frank (1957), The Perceptron—a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

Footnote 64 Perceptrons by Minsky and Papert, 1969

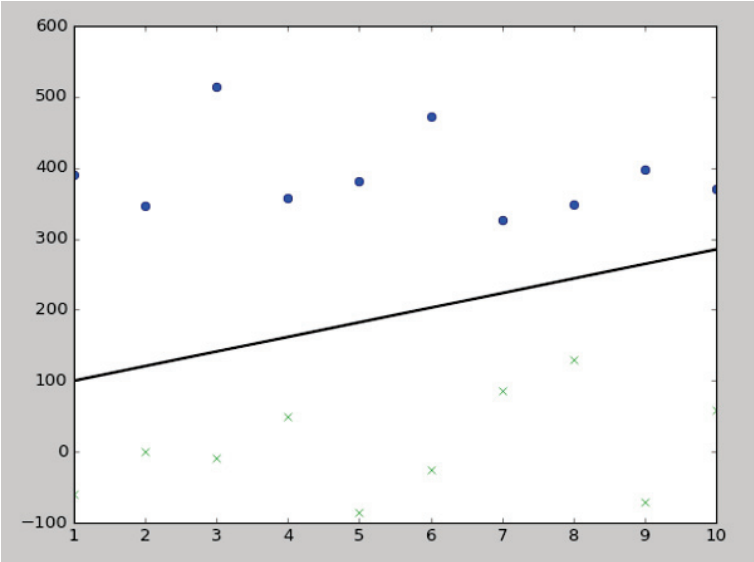


Figure 5.3 Linearly Separable Data

Linear Separable data points, no problem for a Perceptron

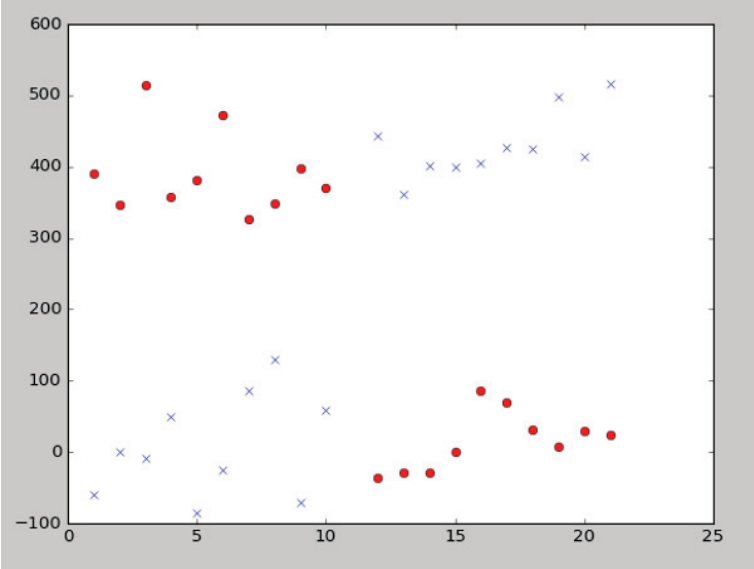


Figure 5.4 Non-Linearly Separable Data

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. But can you think of a way it might be possible to "box in" our classes to separate blue from red? What if we used 2 neurons? That might be enough to "box out" one of the 2 clusters of red data points. And if we had 4 neurons that would be enough to train a perfect classifier.

As most data in the world is not cleanly separable with lines and planes, Minsky and Paperts' "proof" relegated the Perceptron to the storage shelves. But the Perceptron idea didn't die easily. It resurfaced again when Geoffrey Hinton and team footnoteref[3, Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323, 533–536] showed you could use it to solve the XOR problem by stacking Perceptrons and also used an earlier concept called *backpropogation* to adjust the network to correct for errors from each each sample. With this idea for *backpropogation* across layers, the first modern Neural Network was born.

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.

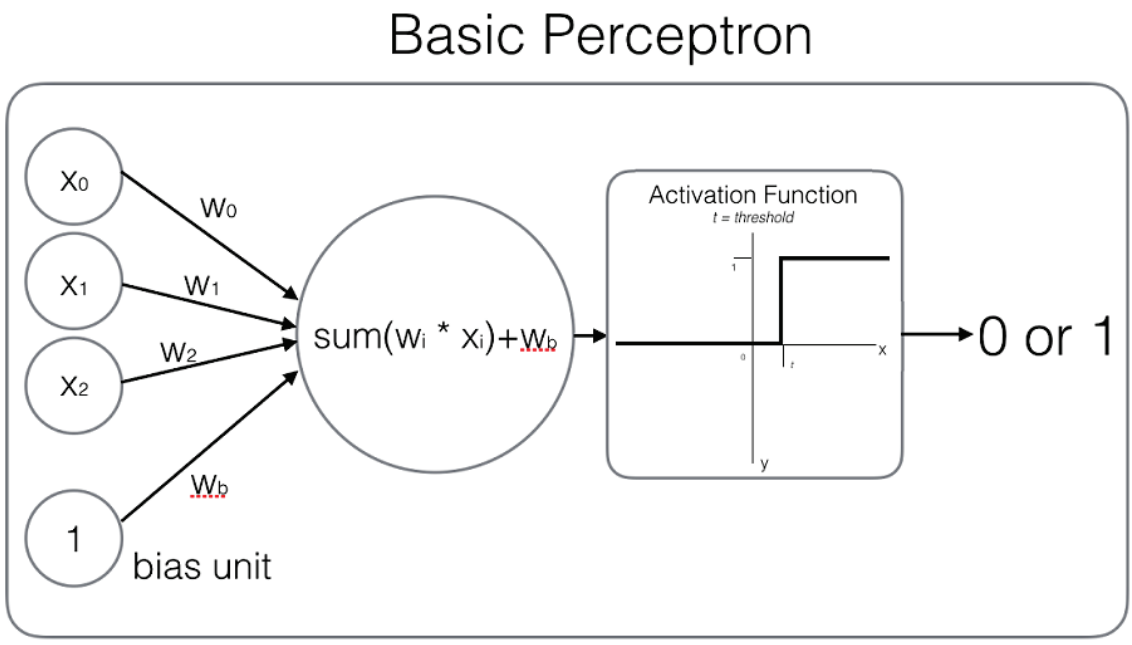


Figure 5.5 Basic Perceptron

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 the Perceptron as a neuron for now, and come back to the terminology when it no longer applies.

The input to the neuron as should be no surprise needs to be numerical values (floats or integers) usually represented as vector. Luckily, we now know to transform text into vectors so we are in the game here. TF-IDF, word frequencies, one-hot encoding all are vector representations of text so we can use them as input for our neuron, and more broadly for a neural network. And as we'll see in the coming chapters we'll see how neural networks themselves can be used to create highly effective vector representations of words as well. But for now we'll just think in terms of numbers as we head into a neural network to look around.

The neuron itself is made up of a collection of "weights" (one weight for each element or dimension of the input vector). When a sample is input into the neuron, each element of the input vector is multiplied by its associated weight. These products are then summed up and passed into the neuron's *activation function*. The original Perceptron used a simple threshold activation function. When the sum of products (inputs * weights) was over a certain amount the neuron would *fire* and output 1, otherwise it would output 0. For those readers with a linear algebra background, this can be represented as the activation function applied to the dot product of the input vector and a vector whose elements are the weights of the neuron. There will be one weight for each input to the neuron, so those vectors will be of the same dimension.

$$f(\vec{x}) = 1 \; if \; \sum_{i=0}^{n} x_i w_i > threshold \; else \; 0$$

Figure 5.6 Threshold Activation Function

Calculating the output of a neuron 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
```

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 * 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.

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.

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 inputs are shown to the system and each time the weights are readjusted a small amount based on whether the neuron output what we wanted it to or not. With enough examples (and under the right conditions) the error *should* tend toward zero, and the system *learns*. The trick is 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 actually result 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(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. As it turns out there needs to be a little more to it than that, as the basic Perceptron has the inherent flaw [65] that if the data is not linearly separable the model will not converge and therefore provide no useful predictive ability.

Footnote 65 en.wikipedia.org/wiki/Perceptrons_(book)#The_XOR_affair

Some of you may have noticed a similarity to another data science algorithm here, *linear regression*. For a set of variables (our input vector) we are trying to learn the appropriate coefficients to solve a linear equation.

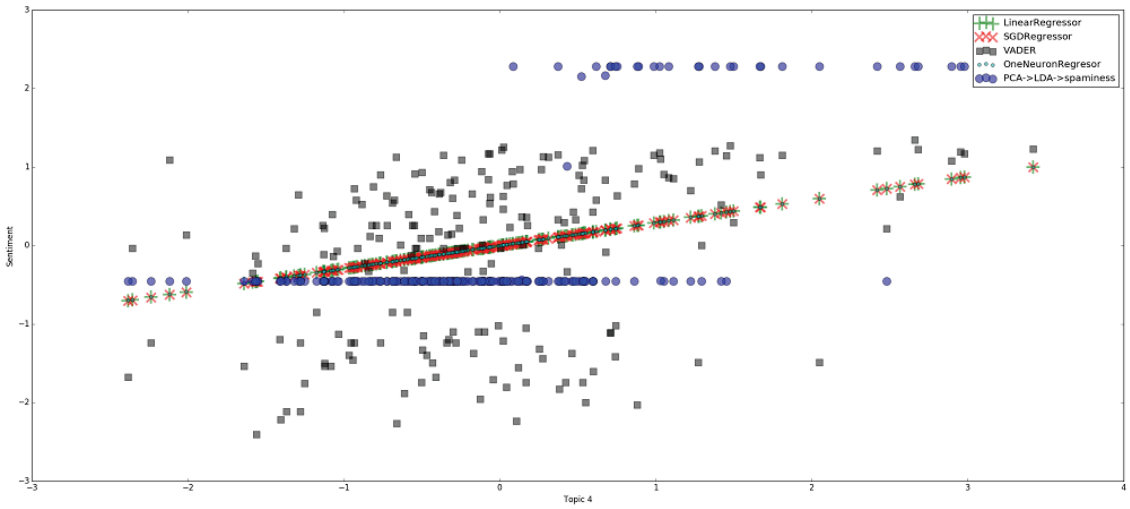


Figure 5.7 Linear Regression

We would like to find the coefficients that minimize a loss function, in this case:

$$J(x) = f(x) - expected$$

Figure 5.8 Cost Function

Instead of solving for the coefficients through direct calculations, we are solving for them iteratively! We inch our way toward the right answer example by example. So why do it this way at all? The short answer is to get to the next step. As we begin to combine neurons this analogy begins to fall apart and the computational steps no longer align. But it is interesting to note the relationship in the base models. And this relationship also highlights why a single Perceptron cannot learn to classify a dataset that is not linear separable, as Minsky and Papert proved.

5.2.1 Backpropagation

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?

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.

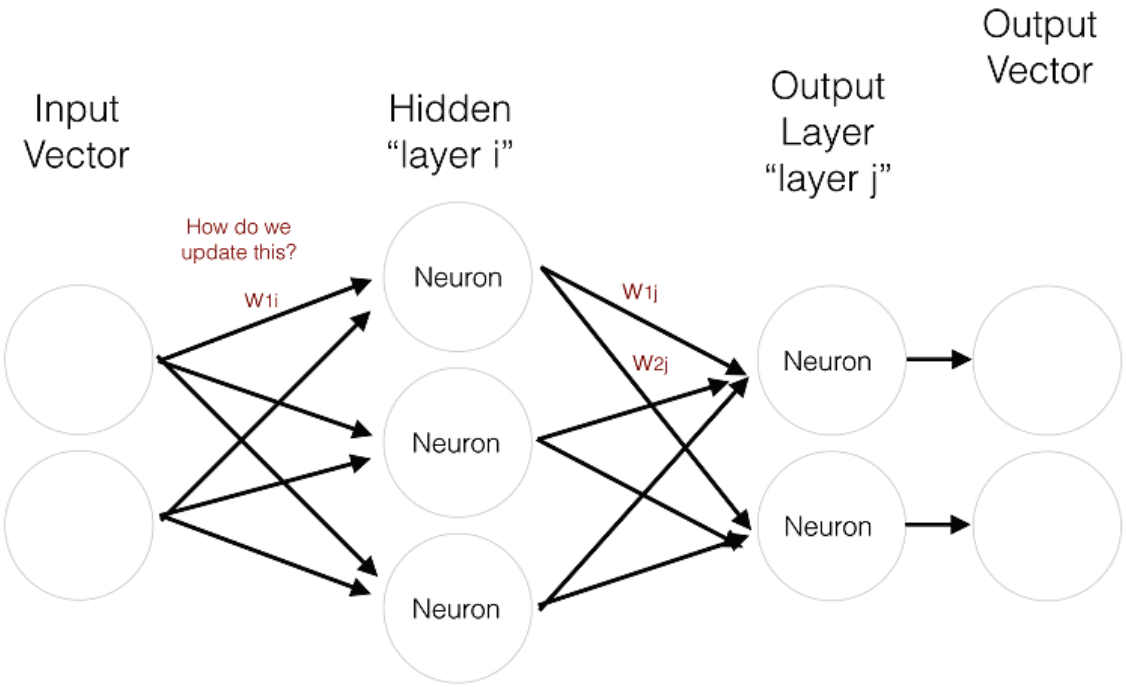


Figure 5.9 Neural Net with Hidden Weights

We need a way to calculate the the amount a particular weight (w 1 i in the image above) contributed to the error given that it contributed to the error via other weights (w 1 j and w 2 j) in the next layer. And the way to do that is with *backpropagation*. The backpropagation algorithm was what pulled the Perceptron off the shelf, dusted it off, and introduced the *neural network*.

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 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 were the output of a neuron becomes the input of another neuron, we begin to talk about *hidden* layers vs an *input* or *output* layer.

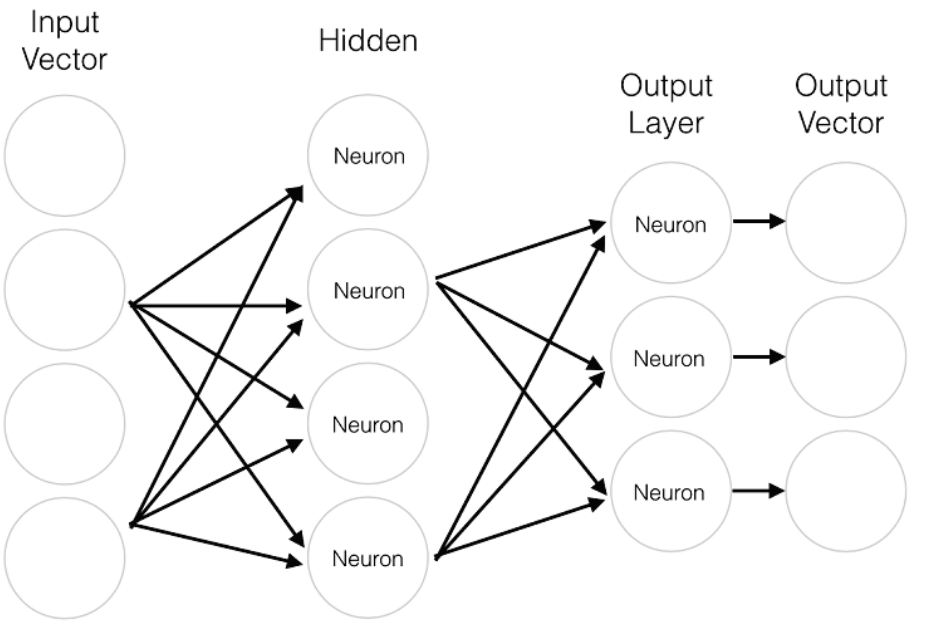


Figure 5.10 Fully Connected Neural Net

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. Specifically we need a 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:

$$S(x) = \frac{1}{1 + e^{-x}}$$

Figure 5.11 Sigmoid Function

Or -1 and 1 in Hyperbolic Tangent (tanh):

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Figure 5.12 Hyperbolic Tangent Function

Where x is the dot product of the input vector and the weight vector.

So why differentiable? If we have a function which is differentiable we can calculate the derivative of the function. If we can make it this far, we can also do partial derivatives, with respect to various variables in the function itself.

We start with the error of the network and apply a cost function, say *squared error*.

$$squarederror = (y_{expected} - y_{actual})^2$$

Figure 5.13 Squared Error Cost Function

We can then lean on the *chain rule* of calculus to calculate the derivative of compositions of functions. And the network itself is nothing but a composition of functions.

$$(f \circ g)' = (f' \circ g) \cdot g'$$

Figure 5.14 Chain Rule

Or more traditionally:

$$(f \circ g)' = F'(x) = f'(g(x))g'(x)$$

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:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = -\alpha o_i (o_j - t_j) o_j (1 - o_j)$$

Figure 5.15 Output Layer Derivative

If we are updating the weights of a hidden layer, things are a little more complex:

$$\Delta w_{ij} = -\alpha \frac{\partial E}{\partial w_{ij}} = -\alpha o_i (\sum_{l \epsilon L} \delta_l w_{jl}) o_j (1 - o_j)$$

Figure 5.16 Inner Layer Derivative

The o in theses 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*.

5.2.2 Let's Go Skiing - The Error Surface

The goal of training in neural networks is to minimize a cost function by finding the best parameters (weights). 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:

$$squarederror = (y_{expected} - y_{actual})^2$$

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).

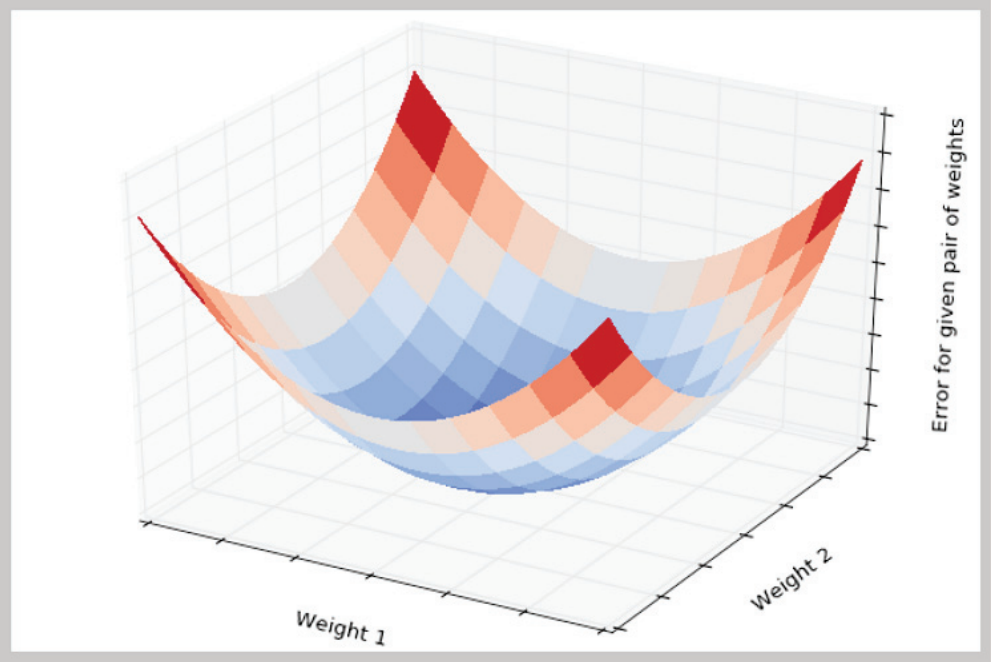


Figure 5.18 Convex Error Curve

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.

5.2.3 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*.

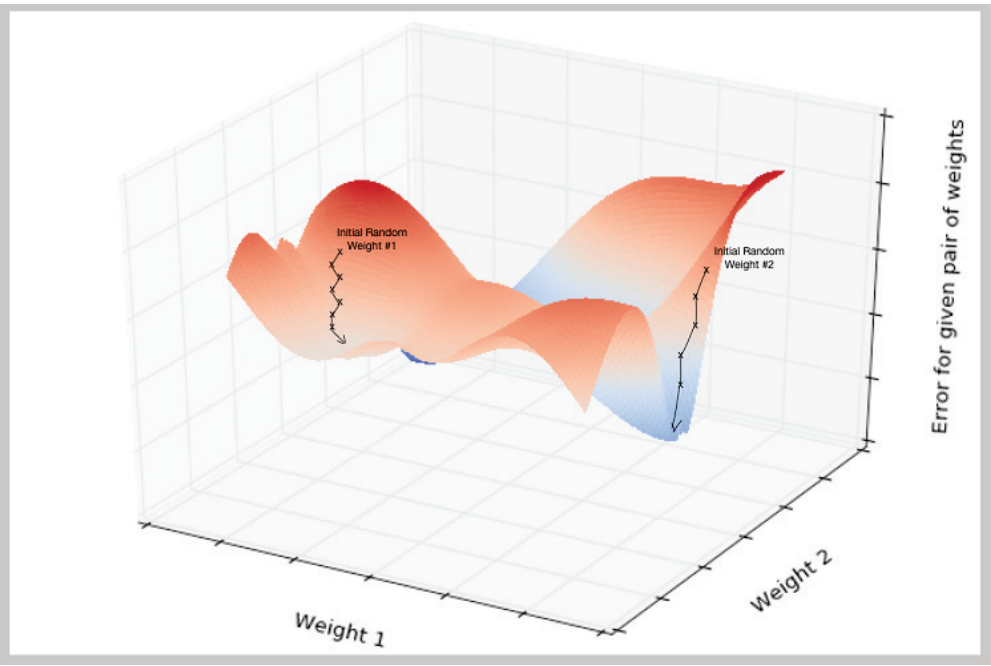


Figure 5.20 Non-Convex Error Curve

And as we get into even higher-dimensional space the local minima will follow us there as well.

5.2.4 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 minumum 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 inconsitent 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 benifits of both *batch* (speedy) and *stochastic* (resilient) training methods.

While the details of how *backpropagation* works are fascinating [66], 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 66 en.wikipedia.org/wiki/Backpropagation

5.2.5 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 soon CNTK from Microsoft as well. 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 shoot 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 an train this.

```
# Here is where we train the model
model.fit(x_train, y_train,
          batch_size=batch_size,
```

```
          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
```

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 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.

```
# And then we save the structure and learned weights for later use
model_structure = model.to_json()
with open("basic_model.json", "w") as json_file:
    json_file.write(model_structure)

model.save_weights("basic_weights.h5")
```

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!

5.2.6 Onward and Deepward

As neural networks have spread an spawned the entire field of *deep learning*, much research has been done and continues to be into the details of theses 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.

5.2.7 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.

5.3 Summary

- Basic Perceptron is a novel way of looking at linear regression
- Neural networks are constructed from these basic blocks
- Backpropagation algorithm is the means by which a networks LEARNS
- Neural Networks are at their heart optimization engines
- There are pitfalls to avoid during training
- But Keras has means to make this accessible

6 Reasoning with Word Vectors

In this chapter:

- Understanding what word vector models can do
- Training Word2vec on new words
- Using pre-trained models for your applications
- Reasoning with word vectors to solve real problems
- Some surprising uses for word embeddings

Have you ever had trouble remembering a word for something or the name of something? For example, maybe you are trying to recall the name of a musician or composer but you only have a general impression of him, like maybe he's the "Einstein of music." Or maybe you've tried to remember the name of a sports team, or their mascot, but can only remember the city they're from. You might even have an analogy question in your head like "Who is the Einstein of music?" or "What was the equivalent soccer team in Seattle for the Timbers in Portland?" Those would be a hard questions for a Google Search if you couldn't remember some keywords associated with the thing you were trying to remember.

Well now you can do semantic reasoning to answer questions like these, thanks to Tomas Mikolov. In 2012, Mikolov, then an intern at Microsoft, found a way to encode the meaning of words in a relatively small number of vector dimensions (200-300) using an algorithm called Word2vec. Word2vec can handle everyday words like "woman" and "king." And it can even handle proper nouns like place names or the names of people, like the musicians and scientists in our example. The vectors capture geography, sentiment, part of speech, meaning, all in a numerical vector. And Mikolov's Word2vec approach is purely statistical, an unsupervised machine learning algorithm.[67] It only requires unlabeled data. It could learn vector representations from a large corpora of unlabeled example sentences or phrases broken into N-grams, or small, overlapping bags of words.

Footnote 67 Word2vec is very similar to an autoencoder with a single hidden layer. That gives it two matrices of weights, one for the input and one for the output, the input weights for each word become the word vectors

Mikolov designed and tuned the Word2vec algorithm with the intent to numerically represent words in a way that would allow reasoning "algebra." As a result his Word2vec representations are a much better at capturing the essence of a word than the word vectors that came out of Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDiA) in Chapter 4. They were a byproduct, rather than the focus of the topic models we built there. Hannes and Hobson used Word2vec vectors to identify synonyms for job titles, programming language names, and even cities. The resulting thesauruses were accurate enough to use directly in a production app, with minimal curation, within Talentpair's candidate sourcing, skill suggestion, and opportunity search engines. This is a valuable feature of word vectors. Word-vector-based pipelines can adapt to new terminology and phrases quickly, as long as the new compound words or phrases are combinations of words already modeled.

Using Mikolov's `Word2vec` (W2V) model we can now do "vector-oriented reasoning," or math with words. We can add and subtract word vectors to reason about the words they represent and answer questions similar to our examples above, like

```
Portland Timbers - Portland + Seattle = ?
```

Ideally we'd like this math (word vector reasoning) to give us

```
Seattle Sounders
```

Similarly, our analogy question "'Einstein' is to 'physics' as __ is to 'classical music'?" can be thought about as a math expression like this

```
Einstein - physics + classical music = ?
```

In this chapter we want to improve on the LSA 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 word vectors produced 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 supplanted LSA topic and word vectors for many applications.

For example, the word vectors we'll create in this chapter can help you generalize a model from only a few labeled examples, producing results you couldn't achieve any other way. You will be able to label a few job titles, song titles, sports teams, or any other set of compound words, according to some feature you are interested in, like say the

seniority or industry or "style" associated with those words. And your model can then "compute" those features from compound words or phrases that the model has never seen before!

6.1 Vector-Oriented Reasoning

In 2013, Tomas Mikolov and his team published their discoveries around applications of word vector representations [68]. The paper with the dry-sounding title "Linguistic Regularities in Continuous Space Word Representations" described a surprisingly accurate language model (an order of magnitude greater leap in accuracy than the incremental improvements demonstrated by other researchers with much more complex models).[69] It was so surprisingly good, in fact, that Mikolov's initial paper was rejected by ICLR (International Conference on Learning Representations)! Established NLP experts assumed that word-order was critical to creating accurate language models.[70] They were sure that Mikolov's performance results were too good to be true.

Footnote 68 msr-waypoint.com/en-us/um/people/gzweig/Pubs/NAACL2013Regularities.pdf

Footnote 69 Radim ehek's interview of Tomas Mikolov:
rare-technologies.com/rrp#episode_1_tomas_mikolov_on_ai

Footnote 70 ICRL2013 open review
openreview.net/forum?id=idpCdOWtqXd60&noteId=C8Vn84fqSG8qa

It took nearly a year for Mikolov and his new team at Google to prove otherwise by releasing the source and resubmitting their paper to a more receptive journal (NAACL). The team discovered that the vector representations 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 computed by like this:

```
(0.0168,  0.007,  0.247, ...) + (0.093, -0.028 , -0.214, ...) -
- (0.104,  0.0883, -0.318, ...) = (0.006, -0.109,  0.352 ...)
```

Then, you can find the Word2vec vector closest to that sum, `(0.006, -.109,…)` and that's your "answer"! 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, do our math, and then convert back to a natural language space. You can probably imagine how important this is to a chatbot, search engine, question answering system, or information extraction algorithm.

Here's another look at how that math works in Word2vec vector space:

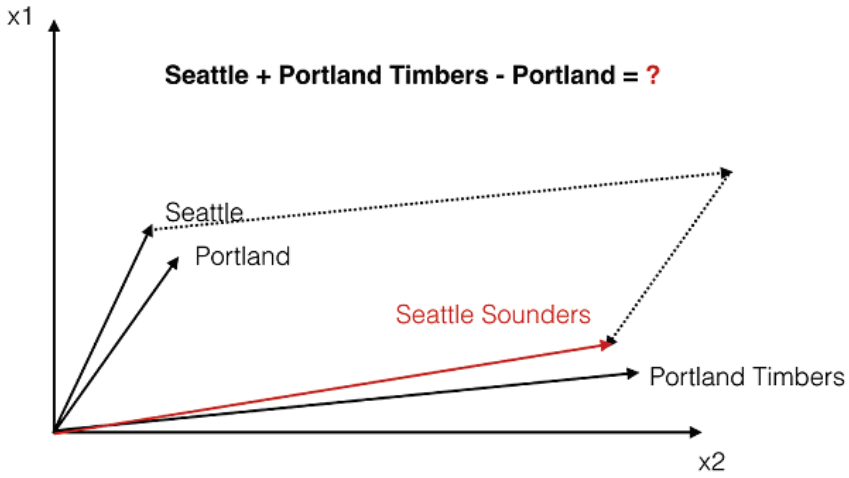


Figure 6.1 Geometry of Word2vec Math

The word closest to the resulting vector will often be the answer to our NLP question.

NOTE The initial paper 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 the Word2Vec model on larger corpora, like the reference implementation trained on 100 billion words from the Google News Corpus to create 300 dimensional vectors.

The research team also discovered that the difference between a singular and a plural word is often roughly the same. In other words

$$\vec{X}_{coffee} - \vec{X}_{coffees} \approx \vec{X}_{cup} - \vec{X}_{cups} \approx \vec{X}_{cookie} - \vec{X}_{cookies}$$

Figure 6.2 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

```
"What is Colorado's equivalent of San Francisco in California?"
```

```
Colorado + San Francisco - California = Denver
```

Let's look at some example word vectors where we add the vectors for a city to the vector for a state name as well as the vector for the state abbreviation. The idea is to disambiguate cities like Portland and incorporate into our city vector the essence of the state where that city is located. Geographic proximity is only a weak driver for these word vectors. These word vectors for US cities seem to capture the cultural, economic, and climate similarities between cities, in addition to the their geographic proximity.

California cities are so culturally different from their geographic neighbors that PCA tends to push them to the side, away from all the other cities in the west. Other 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.

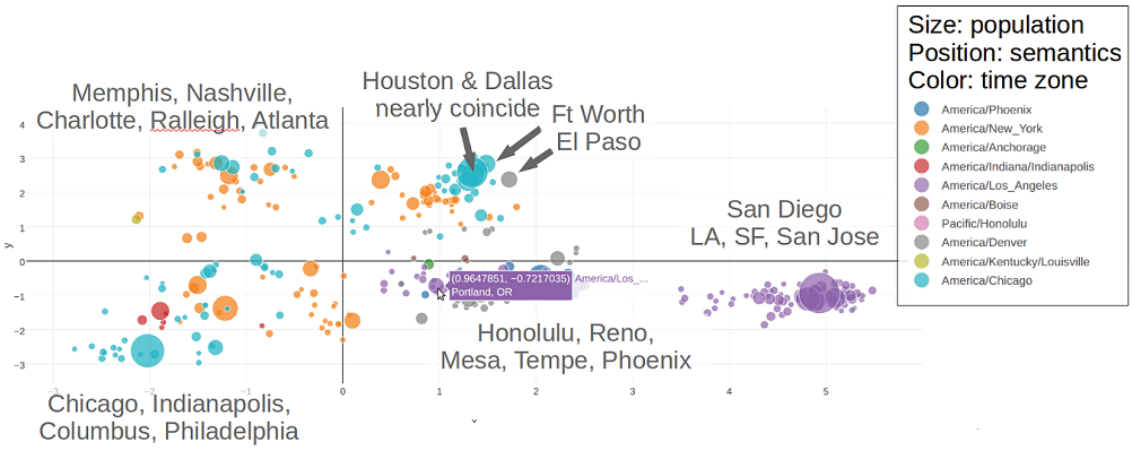
Fortunately we can use conventional algebra to add the vectors for cities to the vectors for states and state abbreviations. Then we can pass these summed vectors directly to PCA to get our 2-D representation so we can plot them in Flatland.

```
>>> import os
>>> import pandas as pd
>>> from nlpia.data import DATA_PATH, get_data
>>> from gensim.models.word2vec import Word2Vec
>>> path = os.path.join(DATA_PATH, 'GoogleNews-vectors-negative300.bin')  # not yet available
                                                                            in the nlpia package
>>> wv = Word2Vec.load_word2vec_format(path, binary=True)
>>> cities = get_data('cities')
>>> us = cities[(cities.country_code == 'US') & (cities.admin1_code.notnull())]
>>> vocab = pd.np.concatenate([us.city_, us.state_, us.state_abbreviation])
>>> vocab = np.array([word for word in vocab if word in wv.wv])
>>> us_300D = pd.DataFrame([[i] + list(wv[c] + (wv[state] if state in vocab else wv[s]) + wv[s])
    for i, c, state, s
...                         in zip(us.index, us.city_, us.state_, us.state_abbreviation)
                            if c in vocab])
```

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 nlpia.data import get_data
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)
>>> us_300D = get_data('cities_us_wordvectors')
>>> us_2D = pca.fit_transform(us_300D)
```



Note: Word2Vec from "GoogleNews-vectors-negative300" with PCA to project to 2D

Figure 6.3 GoogleNews Word2vec 300D vectors projected onto a 2D map using PCA

NOTE Low semantic distance (distance values close to zero) represent high similarity betwen 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 likelyhood 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 "denmoralized" your data. The plotly wrapper expects a DataFrame with a row for each sample and column for things 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.

```
>>> from nlpia.data import get_data
>>> 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})
```

To produce the 2-D representations of your 300D word vectors, we found PCA to be adequate, at least when you limit your vectors to a well-defined class of objects, like cities. 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) to get the most out of your data. 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 Word Vector Representations based on Word2Vec

By turning dictionaries and corpora into a vector space of N dimensions, the representation opens up endless opportunites. As we will discover in later chapters, word vectors are powerful representations for NLP applications, including text generation and text classification. For example, you can train a neural network on a small set of training examples, and due to the properties of the word vectors, the neural network will generalize well and show good classification performance for unseen, but similar words (words not in your labeled training set). We will show you how to do that in Chapter 9. In the following sections, we will focus on the details of the word vector model Word2Vec, which will give you a good basis for other word vector implementations, e.g. GloVe or FastText.

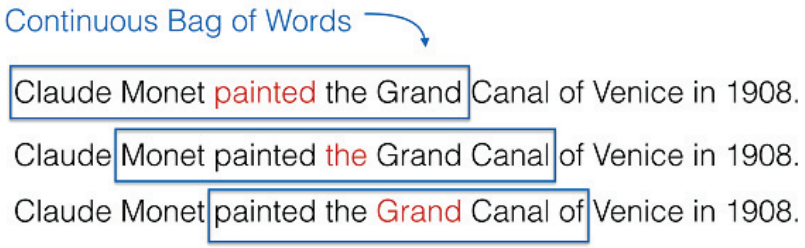
6.2.1 How to compute the Word2Vec Representations?

As you have seen from the previous examples, word vector representations are powerful. They represent the meaning of words as vectors. This allows us to answer analogy questions and 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 in the coming sections.

NOTE

Continous 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 continous 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 continous bag of words for the specific target word, therefore the continous bag of words is changing while sliding over the document.



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 useful for a broad range of documents. Companies that deal with large corpora and can afford the computation have open sourced their pretrained word vector models.

TIP Several companies provide their pre-trained word vector representations free of charge. The pretrained models are great starting points for your word vector applications.

- Facebook word vectors for 50+ languages: github.com/facebookresearch/fastText
- Google Word2vec based on Google News: drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/

Nonetheless, if your domain has specialized vocabuluary or uses words in different ways from the copora that were used to train the available models, training your own word vector models based on your corpus will improve your accuracy. We will show you how in a later part of this chapter.

6.2.2 Skip-gram Approach

In the skip-gram training approach, we are trying to predict the surronding 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 skipgrams are the surrounding words "*Claude*", "*Monet*", "*the*" and "*Grand*".

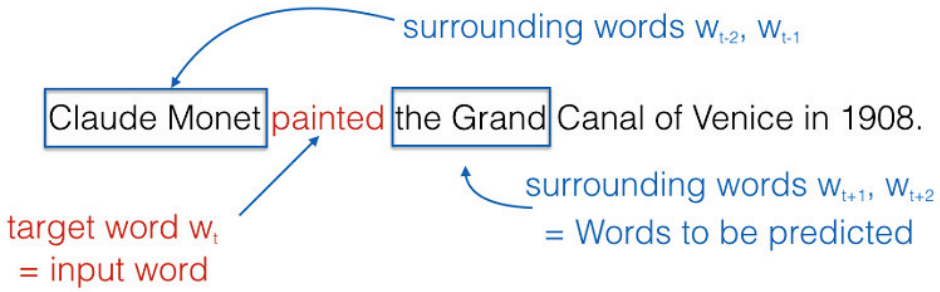


Figure 6.4 Training Input and Output Example for the Skipgram Approach

The structure of the neural network which is used to predict the surronding words is

fairly simple. As you can see in the figure below 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[71], which is commonly used for classification problems.

Footnote 71 Training speedup was achieved by approximating the softmax, O(M) with a heirarchical softmax by Hinton that groups words with similar frequency together during computation of the softmax: papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf

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.

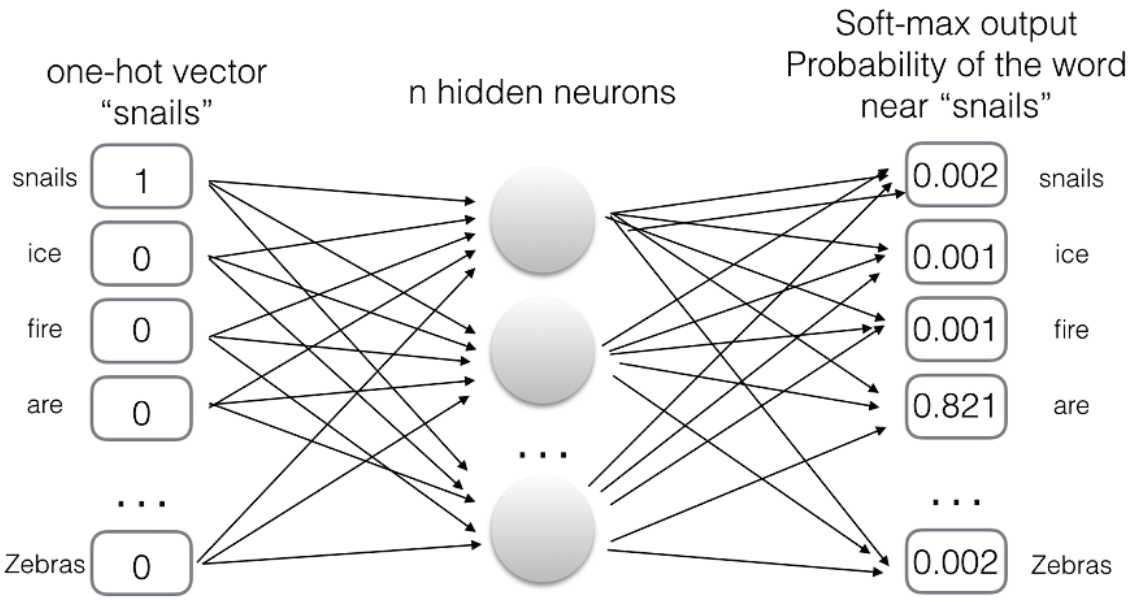


Figure 6.5 Network Example for the Skipgram Training

NOTE When you look at the structure of the training network for the word embedding, you'll notice that the implementation looks similar to what some people call a "multilayer perceptron." However, it would be more accurately described a as a fully-connected feedfoward network with a single hidden layer. Many experts, including Geoffrey Hinton, consider that term to be an oxymoron. If you would like to learn more about the details of neural networks, check out Appendix C.

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 four 5-grams with the target word as the center:

```
...
input: painted (wt) -> output: Claude (wt-2),  Monet (wt-1),   the (wt+1),   Grand (wt+2)
input: the (wt)     -> output: Monet (wt-2),   painted (wt-1), Grand (wt+1), Canal (wt+2)
input: Grand (wt)   -> output: painted (wt-2), the (wt-1),     Canal (wt+1), of (wt+2)
input: Canal (wt)   -> output: the (wt-2),     Grand (wt-1),   of (wt+1),    Venice (wt+2)
...
```

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 wordh is being predicted based on the input word.

The words are represented as one-hot vectors to the network (check Chapter 2 for more details). The idea of a one-hot vector also becomes useful to understand the output vector. The softmax activation of the output layer nodes (one for each token in the skip-gram) calculates the probablity of a 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 probablity will be converted to 1 and all remainind terms will be set to 0. This simplifies the error calculation and speeds up the training. After the training is complete, the output layer of the network can be ignored. Only the weights of the inputs to the hidden layer are used as the embeddings. The dot product of the one-hot vector representing the input term and the weights then represents the *word vector embedding*.

6.2.3 Continous Bag of Words Approach

In the Continous Bag of Words (CBOW) approach, we are trying to predict the center word based on the surrounding words. As you can see in the following illustration, this is basically a flipped Skip-gram approach. In the *CBOW* approach, we are creating training sets as follows:

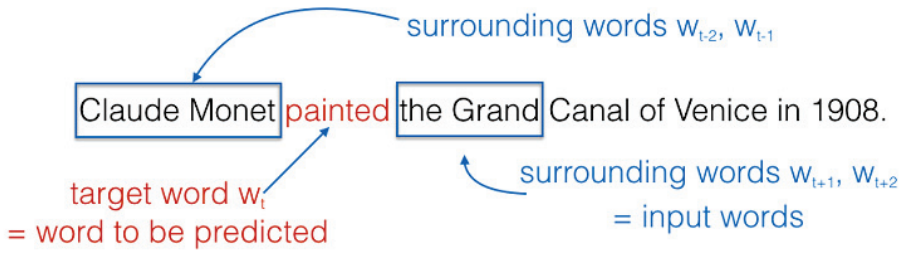
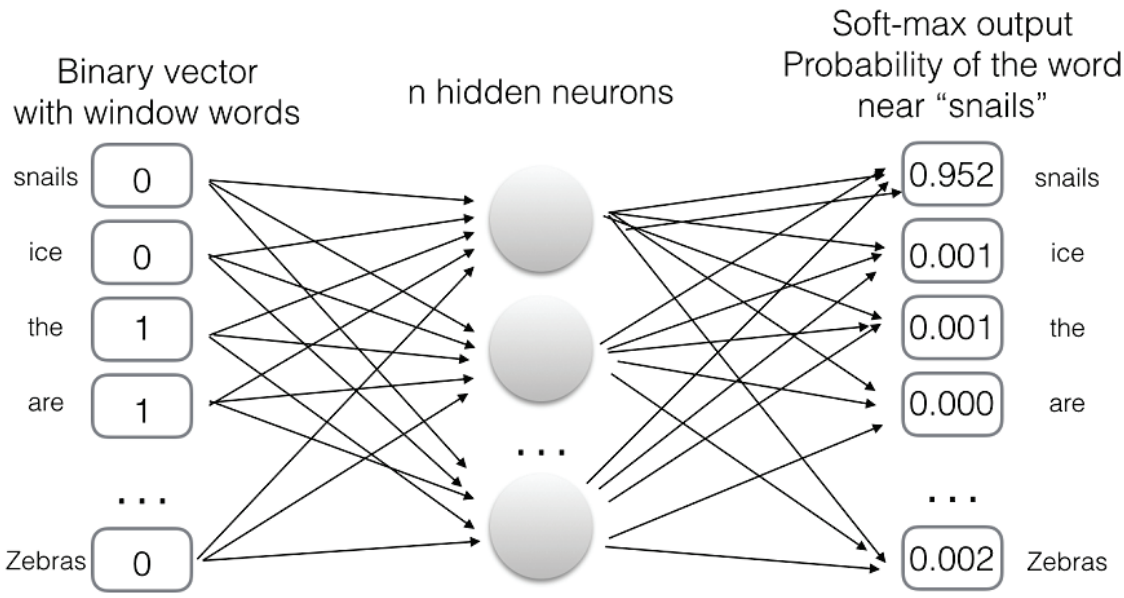


Figure 6.6 Training Input and Output Example for the CBOW Approach

```
...
input: Claude (wt-2),  Monet (wt-1),   the (wt+1),   Grand (wt+2)  -> output: painted (wt)
input: Monet (wt-2),   painted (wt-1), Grand (wt+1), Canal (wt+2)  -> output: the (wt)
input: painted (wt-2), the (wt-1),     Canal (wt+1), of (wt+2)     -> output: Grand (wt)
input: the (wt-2),     Grand (wt-1),   of (wt+1),    Venice (wt+2) -> output: Canal (wt)
...
```

Based on the training sets, we can create training pairs wt-2 wt, wt-1 wt, wt+1 wt, etc. The input vectors are represented again as one-hot vectors and the output is derived from the softmax of the output node with the highest probability



6.2.4 Skip-gram vs. Continous Bag of Words: When to use which approach

Mikolov hightlighted 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 continous bag of words approach shows higher accurracies for frequent words and it is much faster to train.

6.2.5 Computational Tricks of Word2vec

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 team [72] used cooccurrence frequency to identify bigrams and trigrams that should be considered single terms using the following scoring function:

Footnote 72 The publication by the team around Tomas Mikolov is an interesting read: arxiv.org/pdf/1310.4546.pdf

$$\text{score}(w_i, w_j) = \frac{\text{count}(w_i w_j) - \delta}{\text{count}(w_i) \times \text{count}(w_j)}$$

If the words w_iw_i and w_jw_j result in a high score and the score is above the threshold \delta\delta then they will be included in the Word2vec vocabulary as a pair

term. When you browse the included words in the vocabulary, you'll notice various terms that include "_". These are the pair terms like "New_York" or "San_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 occuring bigrams like team names to the Word2vec model, they can easily be included in the one-hot vector for model training.

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 coorcurrence 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):

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

In the earlier equation, f(w_i)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}10^{-5} and 10^{-6}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}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.

One last trick the Mikolov came up with was the idea of negative sampling. If our vocabulary contains thousands or millions of words, then updating the weights for the large one-hot vector can be daunting and time consuming. That's where the idea of negative sampling becomes interesting. Instead of updating all weights of the words which weren't included in the word window, the Google team suggested to sample a few negative samples to update their weights. That way, the computation can be reduced dramatically and the performance of the trained network doesn't decrease significantly.

6.3 How to use Gensim's Word2vec module?

If the previous section sounded too complicated, don't worry. Various companies provide their pretrained word vector models. Popular NLP libraries for different programming languages allow you to use the pretrained models efficiently. In the following section, we would like to show you 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 5.

With the few lines of code below, you can load a Word2vec model [73].[74]

Footnote 73 You can download the latest Google Word2vec model from drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/

Footnote 74 If you would prefer a programatic downloader to retrieve the Word2vec model as well as other models and datasets, you can `pip install nlpia` and use the `nlpia.data.download()` method.

```
>>> from gensim.models.keyedvectors import KeyedVectors
>>> word_vectors = KeyedVectors.load_word2vec_format
    ('/path/to/the/model/GoogleNews-vectors-negative300.bin.gz', binary=True)
```

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* attribute. 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.bin.gz', binary=True, limit=200000)
```

However, keep in mind that a reduced word vector model will lead to a lower performance in 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 as much training data is available for your problem.

If you want to determine synonyms, 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 synonyms as you like and also know how synonymous Word2vec thinks they are! 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 find antonyms. The argument *topn* determines how many synonyms 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 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'
```

If you want to preform 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 as a dictionary where your word of interest is the dictionary key.

```
>>> 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, ...
```

6.4 How to generate your own Word vector representations?

Sometimes it can be useful to generate domain-specific word vector models. In this case, the text you are converting into vectors is very focused on a domain (e.g., processing medical transcripts) and the focus of your application will not change. In the following section, we will show you how to train your own Word2vec model without going through the pain of implementing the actual Word2vec algorithm.

For the purpose of training a domain-specific Word2vec model, we will use the popular NLP library *gensim*, but before we can start training the model, we will need to preprocess our corpus a bit.

6.4.1 Preprocess Steps

Before we can train our model, we need to break our documents into sentences and the sentences into tokens. *gensim's* Word2vec model is aspecting 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 segmeneter availablein NLTK and gensim for some applications.[75]. Once you have converted your documents into lists of token lists (one for each sentence), you are aready for your Word2vec training.

Footnote 75 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

6.4.2 Train your domain-specific Word2vec model

gensim provides a wonderful tool to train your own Word2vec models efficiently. You can get started by loading the *word2vec* module.

```
>>> from gensim.models import word2Vec
```

The training requires a few setup details

```
num_features = 300    # Number of vector elements to represent the word vector
min_word_count = 3    # Min number of word count to be considered in the Word2vec model
num_workers = 2       # Number of threads to run in parallel
window_size = 6       # Context window size
subsampling = 1e-3    # Subsampling rate for frequent terms
```

With this, you are ready to start our training

```
>>> model = Word2vec.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. If you start processing larger corpora, e.g. the Wikipedia corpus, expect a much longer training time and a much larger memory consumption. However, using gensim, your memroy consumption will only increase if you increase the size of your vocabulary, not your training corpus.

Word2vec models are not very memory efficient. But *gensim* has you covered. If you run

```
>>> model.init_sims(replace=True)
```

gensim will "freeze" the training model and only store the weights of for the hidden layer and discard the output weights which predict word coocurrences. This reduces the memory consumption by roughly 50%. The disadvantage is that the model cannot be trained further on additional documents once the weights of the output layer have been discarded.

You can save the trained model with the following command and preserve it for a 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 import Word2vec
>>> model_name = "my_domain_specific_word2vec_model"
>>> model = Word2vec.Word2vec.load(model_name)
>>> model.most_similar('radiology')
```

6.5 Word2vec vs GloVe (Global Vector)

Word2vec was a breakthrough, but it relies on a neural network model which must be trained using backpropagation, which is usually less efficient than direct optimization of a cost function using gradient descent. Stanford NLP researchers[76] 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 found that counting word co-occurrences and recording them in a square matrix they could then compute the singular value decomposition (SVD)[77] of this co-occurrence matrix they could split this co-occurrence matrix into the same two weight matrices that Word2vec produces (if they are normalized the same way and the Word2vec model also converges to the global minimum).[78] It's this direct optimization of the global vectors of word co-occurrences (co-occurrences across the entire corpus) that gives GloVe it's name.

Footnote 76 Stanford GloVe Project: nlp.stanford.edu/projects/glove/

Footnote 77 See Chapter 5 and Appendix C for more details on SVD

Footnote 78 *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 for a language model with the same accuracy as Word2vec, but in much less time. And it uses the data more efficiently so it can be trained on smaller corpora and still converge.[79] And the research and experience on SVD procedures is much more extensive than for neural nets, giving you more confidence that your result is indeed the best possible representation of the word co-occurrence count matrix that word vector weight matrices represent.[Richard Socher explains GloVe: youtu.be/oGk1v1jQITw?t=19m50s]

Footnote 79 Gensim's comparison of Word2vec and GloVe performance: rare-technologies.com/making-sense-of-Word2vec/#glove_vs_word2vec

Fortunately the gensim package provides an implementation of GloVe and compares its performance to Word2vec so you can chose which one you prefer for your application. But when implemented properly and trained to convergence, they should both produce very similar results, regardless of your pipeline. The `gensim` team found cases where GloVe did indeed converge more quickly to a higher accuracy solution than Word2vec. However, for the fully trained, Word2vec model trained on a large corpus (GoogleNews300negative) GloVe vectors failed to exceed Word2vec's accuracy that Google achieved withfor the pretrained model. Vectors trained to convergence on large corpora with Word2vec will match GloVe's performance.

Advantages of GloVe:

- Faster training
- Better RAM/CPU efficiency (large corpora)
- More efficient use of data (small corpora)

6.6 Word2vec vs LSA

You might now be wondering how Word2vec word vectors compare to the LSA topic vectors of Chapter 5. LSA topic vectors correspond to the sum of the Word2vec word vectors for all the words in those documents. And even though we didn't use them in Chapter 5, LSA gives us word vectors too. If your LSA matrix of topic vectors is of size [N_words x 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 word vectors are just as useful as Word2vec vectors for finding both synonyms and antonyms. As you learned in the GloVe discussion above, Word2vec vectors can be created using the exact same SVD algorithm used for LSA. The only difference is the size of the documents used for training. In LSA, documents are typically many sentences. In Word2vec the "documents" are typically only 5 words long, and in Word2vec these skip-gram or CBOW documents never extend beyond sentence boundaries. And Word2vec gets more use out of the same number of words in it's documents by creating a sliding window that overlaps from one document to the next, reusing the same words 5 times before sliding on.

What about incremental or online training? In both LSA and Word2vec we can add new documents to our corpus and adjust our word vectors to account for the co-occurrences in the new documents, as long as we already have "bins" in our lexicon for all the words in the new documents. In the case of LSA, we cannot expand our vocabulary (lexicon) to include new words discovered in any new documents added to our corpus after the initial pass to compile the vocabulary. However, it is possible for Word2vec to add additional rows as new words are discovered. As long as the learning rate is adjusted so that words found early on in the corpus are not weighted more heavily than later words, then Word2vec can handle these additional words (rows) seamlessly.

Like GloVe, the other SVD-based word vector algorithm, LSA trains faster compared to Word2vec. This is its onlyadvantage. And the "killer app" for Word2vec is the semantic reasoning possible with Word2vec. With Word2vec you can determine the answer to questions like "Harry Potter" + "University" = "Hogwarts"[80]

Footnote 80 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:

- Scales better to larger vocabularies and corpora
- Supports online training
- Supports semantic reasoning including analogies

6.7 Doc2Vec

The concept of Word2vec can also be applied to larger sections of documents, like sentences, paragraphs or entire documents. As it turns out *Doc2Vec* is a recent approach used to predict a vector representation of a document. The concept allows the same advantages as Word2vec does, such as continuing the training to add new documents as your corpus grows. *Doc2Vec* brings the semantic reasoning that allowed us to do math on words (semantic reasoning) to do similar math on documents.[81]

Footnote 81 arxiv.org/pdf/1405.4053v2.pdf

6.8 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 heirogliphics 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* [82] like RO13 or a *substitution cipher* [83] 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 82 en.wikipedia.org/wiki/Caesar_cipher

Footnote 83 en.wikipedia.org/wiki/Substitution_cipher

Figure 6.7 Decoder Rings

And Word2vec has even been used to glean information and relationships from unatural 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.

[84] 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. Sen And if an ID number only apprears in one "sentence"

Footnote 84 medium.com/towards-data-science/a-non-nlp-application-of-word2vec-c637e35d3668

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). a sequence of symbols with structure and semantics and delimitters for meaningful groups of characters, like the parentheses and hyphens in phone numbers. These symbol sequences and numbers just need to be tokenized by defining those delimitters and then embedded with something like `model = Word2Vec(sentences, min_count=3)`

6.9 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 pre-trained 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 especialy 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.

7

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 between the words, in the order and combination of words. And 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.[85] 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 word, 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 85 *International Association of Facilitators Handbook*, books.google.com/books?id=TgWsY7oSgtsC&lpg=PT35&dq=%22beneath%20the%20words%22%20empathy%20list

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 what machines tend to produce sounds so 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. And this intution 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.

7.1.2 Word Proximity

The ship's hull, despite years at sea, millions of tons of cargo, and two mid-sea collisions, shone lik

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 temporarily. 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. You could just as easily scramble the position of each of the example's elements before feeding it into the net (as long as you scrambled all inputs in the exact same way) and get the same output. 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.

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 and 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 Theano for the examples. We also need the h5py package for saving the internal state of our trained model.

By default, Keras will use Theano 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 hyperparameters 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 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. They 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 of each data point, as in a traditional Feed-Forward net, but defines a set of *filters* (also known as *kernels*) that move across the image. In image recognition the elements of each data point is usually the intensity of each pixel in a black and white image, or the intensity in each of the color channels of each pixel in a color image. Each filter slides across the input in turn and generates output at each location it pauses at.

The distance of each convolution 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 word relating its neighbors.

The filters are themselves are small sets of weights, realized as a matrix as the positions of the weights are important, with an activation function on the back side. And they are much smaller than the image as a whole, usually just 3x3 or 5x5 pixels in size. 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.

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). 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.

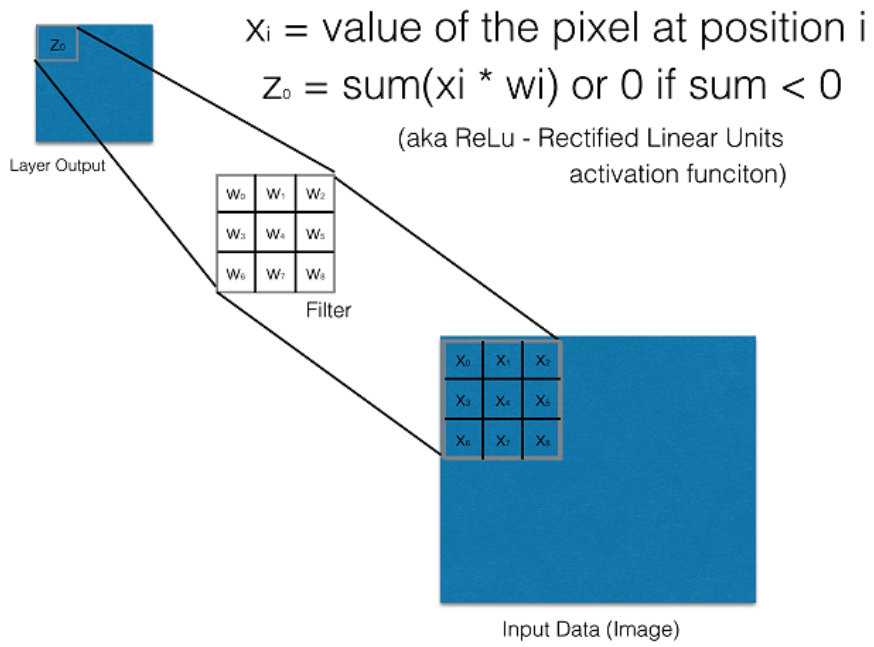


Figure 7.1 Convolutional Neural Net Step

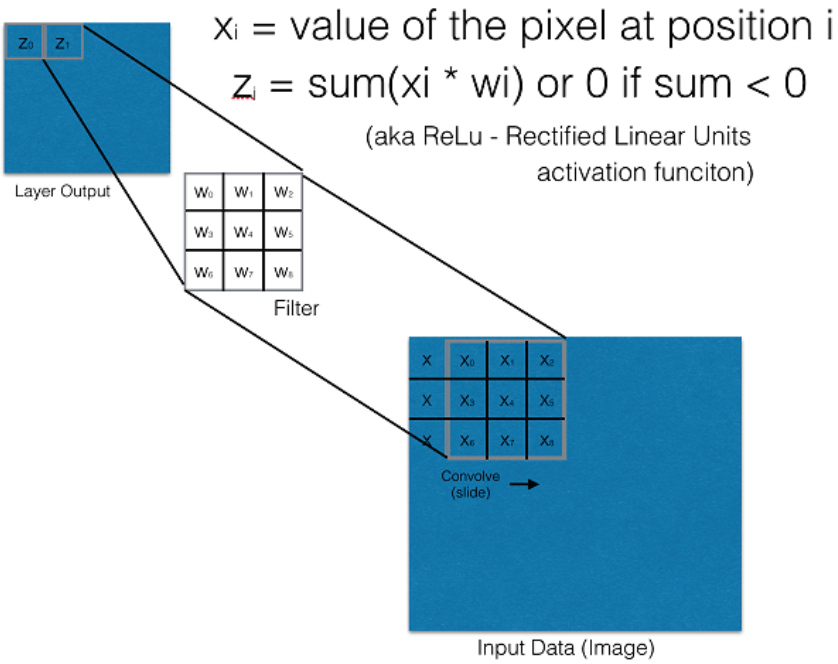


Figure 7.2 Convolution

There will be several of these filters in a layer and as they each convolve over the entire image, they create a value (via the activation function) that was created by the input from the original data at that position. At each such position the weights of a particular filter are held constant for each input. The output of a Convolutional layer could then be thought of as versions of the input, in this case an image as seen through each filter. The output "image" from each filter is then stacked. And we have new "image" that is *multi-channel*, one channel per filter.

7.3.1 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 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 an minimize the ill effects though. You can pad with 0's:

```
from keras.layers.convolutional import ZeroPadding2D

model.add(ZeroPadding2D(padding=(amt)))  # Where amt == amount of padding added to each borders
```

or, in the layer itself:

```
model.add(Conv1D(filters,
                 kernel_size,
                 padding='same',   # <-- 'same' or 'valid'
                 activation='relu',
                 strides=1,
                 input_shape=(maxlen, embedding_dims)))
```

More on the implementation details in a moment.

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.

The output from these filters can then be fed into a Feed-Forward network or another layer of convolutional filters. But once we have a final output (and hence an error) we can backpropogate back through 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. In a color image the filters become 3-dimensional and the 3

layers of the 3rd dimension each handle a color channel of the image, red-green-blue. Or even more dimensions if the data warrants it. But we can just easily scale back down to 1 dimensional filters, which is what we shall do.

7.3.2 Learning

The filters themselves, as in any good Neural Network, start out with weights that are initialized to small random values. So how is the output "image" going to be anything more than noise? At first, at the beginning 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.

$$\frac{\partial E}{\partial w_{ab}} = \sum_{i=0}^{m} \sum_{j=0}^{n} \frac{\partial E}{\partial x_{ij}} \frac{\partial x_{ij}}{\partial w_{ab}}$$

Figure 7.3 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 a weight contributed to the overall error of the system. And then deciding 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. Well it turns out we can use Convlutional Neural Networks for NLP by using word embeddings instead of pixel values as the input to our network and then pass the filters over the embedding vectors just as we did with pictures and pixels. 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 relavent information is in the relative "horizontal" positions. 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". Either way, we only want to focus on the relationships of tokens in one spatial dimension. So instead of 2 dimensional filter that we would convolve over a 2 dimensional input (a picture) or 3 dimensional input (a color picture), we will convolve 1 dimensional filters over a 1 dimensional input, such as a sentence.

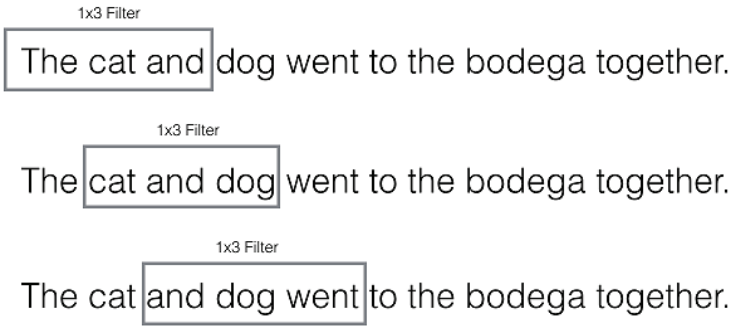


Figure 7.4 1 Dimensional Convolution

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" 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.

The term convolution is actually a bit of shorthand. The actual sliding has no effect on the model. It 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 order of the output is respected (with respect to the order of the input). The filters themselves are unchanged for a given input, 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 is why research keep coming back to this tool.

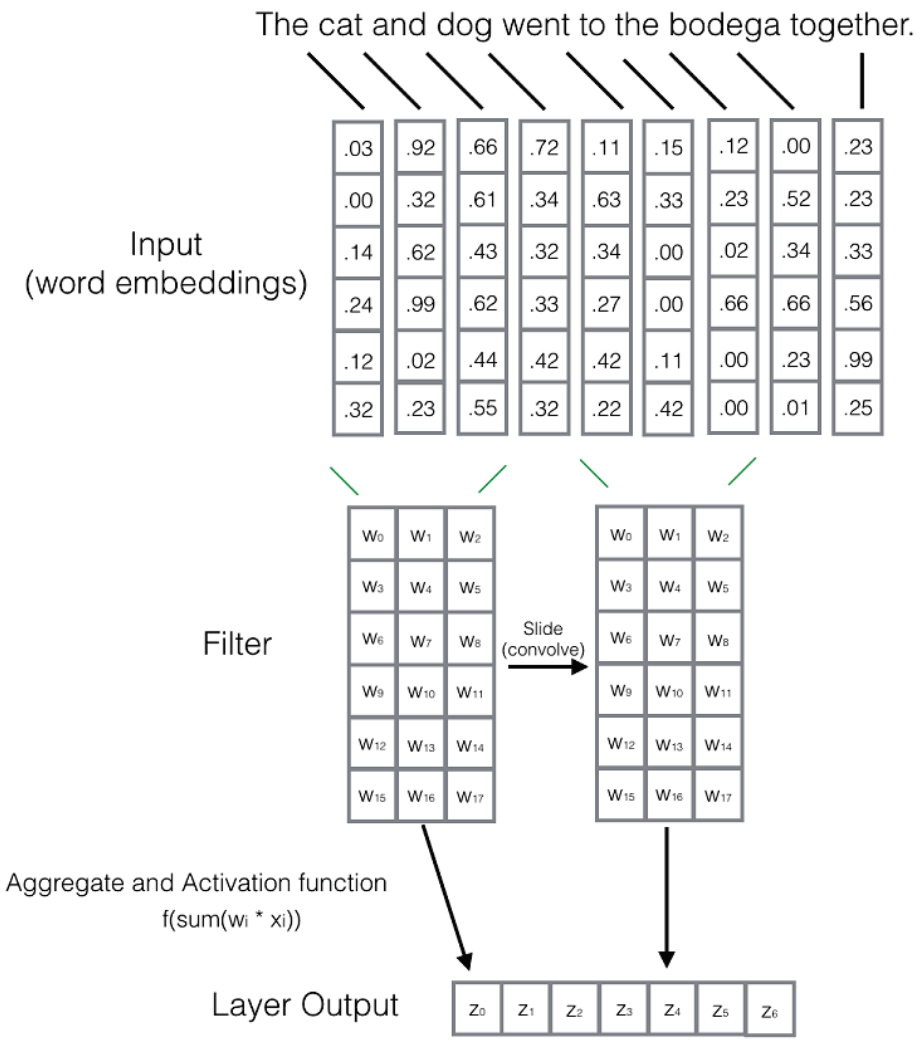


Figure 7.5 1 Dimensional Convolution with Embeddings

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 which is pre-labeled with 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  # Keras takes care of most of this but it likes to see Numpy arrays
from keras.preprocessing import sequence    # A helper module to handle padding input
from keras.models import Sequential         # The base keras Neural Network model
from keras.layers import Dense, Dropout, Activation   # The layer objects we will pile into the model
from keras.layers import Conv1D, GlobalMaxPooling1D   # Our convolution layer, and pooling
```

First download the dataset from ai.stanford.edu/~amaas/data/sentiment. A datset compiled for the 2011 paper Learning Word Vectors for Sentiment Analysis [86]. 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. 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 hyperpameters like *momentum* are used.

Footnote 86 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

```
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 pretrained word2vec vectors, so download those from here: (pending the repository to host the googlenews-vectors word2vec model, google for it there are several repos hosting it on Github). Then we will use gensim to unpack them. You can experiment with

the limit argument, 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 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 <br\>. Those are things we do want to exclude as they aren't usually relevant to the sentiment of text.

And then we 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 of the target values from the dataset """
    expected = []
    for sample in dataset:
        expected.append(sample[0])
    return expected
```

And then pass our data into those functions:

```
vectorized_data = tokenize_and_vectorize(dataset)
expected = collect_expected(dataset)
```

We then need 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 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 hyperparameters 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         # How many samples to show the net before backpropagating the error and
                          updating the weights
embedding_dims = 300    # Length of the token vectors we will create for passing into the Convnet
filters = 250           # Number of filters we will train
kernel_size = 3         # 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
hidden_dims = 250       # Number of neurons in the plain feed forward net at the end of the chain
epochs = 2              # Number of times we will pass the entire training dataset through the
                          network
```

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 int 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 the base class for all Neural Networks in Keras. From here we can start to layer on the magic.

The piece we add is a Convolutional Layer. In this case we are assuming it is okay 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 set to 3 above. And we are using the 'relu' activation function. So at each step, the filter 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)))
```

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. The idea is for given subsection of the *filtered* output, the network takes a single value representation of that section and uses that to represent that section or block. 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.

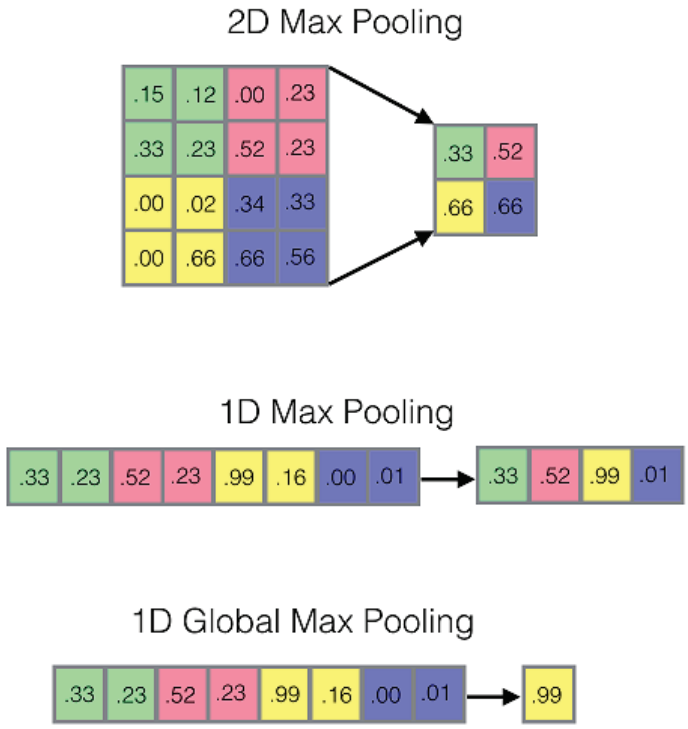


Figure 7.6 Pooling Layers

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. 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 NLP.

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())
```

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 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! 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 NLP 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 received 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).

The parameter passed in to 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. The rest will go in as 0's. 20% dropout is common, but dropout up to 50% can have satifying results, so another hyperparameter to play with.

And then we use 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.

```
# 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 build it and train it. The loss function is what the network will try to minimize and the optimizer is one of a list of strategies to optimize training during runtime; such as a *learning rate* that decays overtime, so you can make large leaps at the beginning then make more gentle steps as you close in on the global minimum. Or momentum, to move more quickly toward the minimum if the network makes repeated progress in a given "direction". Each comes with its own trade-offs and benefits. The settings are described in the documentation, but the details of each are beyond the scope of this book.

Where *compile* builds the model, *fit* is where the magic happens. All of the inputs times their weights, all of the activation functions, all the backpropogation 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 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.

```
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size,
          epochs=epochs,
          validation_data=(x_test, y_test))
```

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()
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")
```

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 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 Theano 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...
/Users/uglyboxer/.virtualenvs/nltk/lib/python3.5/site-packages/keras/backend/theano_backend.py:1703:
  filter_dilation=dilation_rate)
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 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.

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. If only one of 100 examples 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. The 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.

```
# 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.
vec_list = tokenize_and_vectorize([(1, sample_1)])

# Tokenize returns a list of the data (length 1 here)
test_vec_list = pad_trunc(vec_list, maxlen)

test_vec = np.reshape(test_vec, (len(test_vec_list), maxlen, embedding_dims))
model.predict(test_vec)
```

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 greyscale 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 NLP. 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 CNN, 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 choose a CNNs 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 preform 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.

7.5 Summary

In this chapter,

- 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 in the sequence by convolving shared weights with those words. 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 a fragment or sentence. Can we give the net a concept of what went on earlier? 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, but the effects on the learning stage of the next example are largely independent, in the aggregate, of the order of input data. Convnets make an attempt to capture that ordering relationship, 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 (word vector length x number of words in sample).

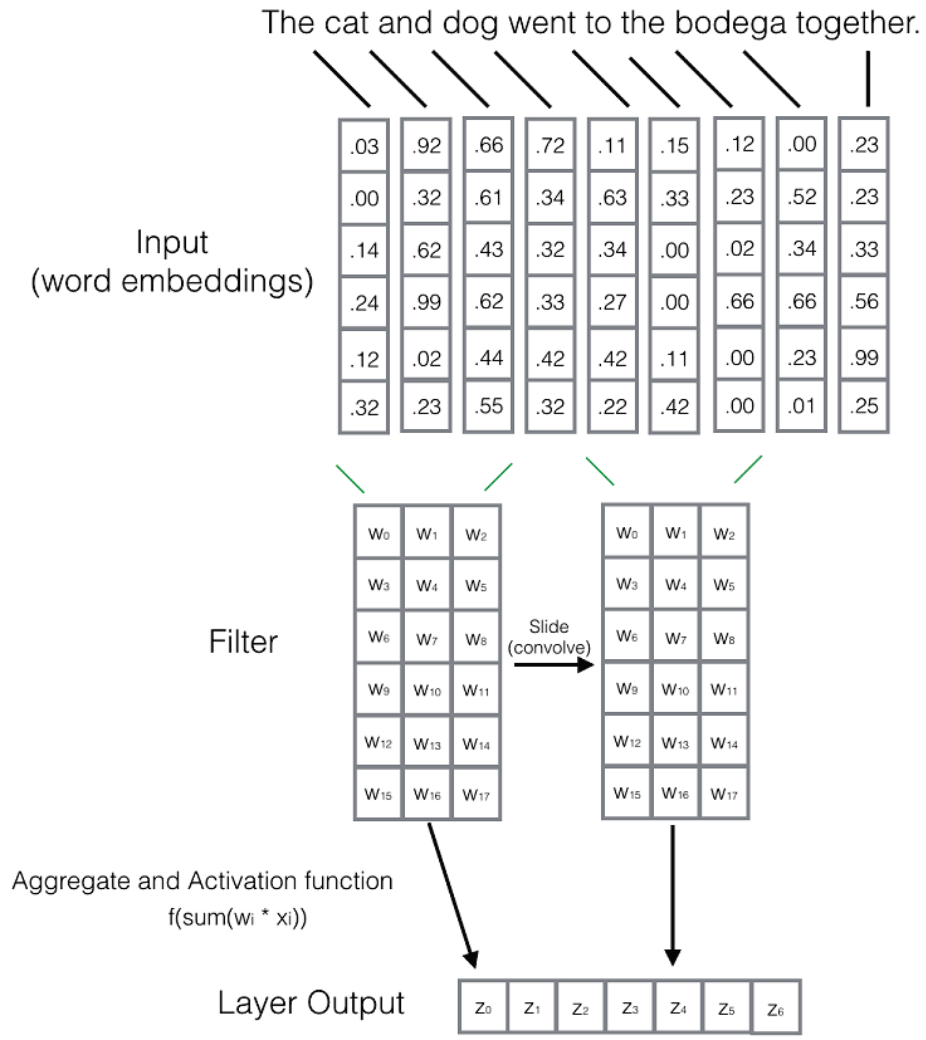


Figure 8.1 1 Dimensional Convolution with Embeddings

But that sample could just as easily be passed into a standard feed-forward network from chapter 5 right?

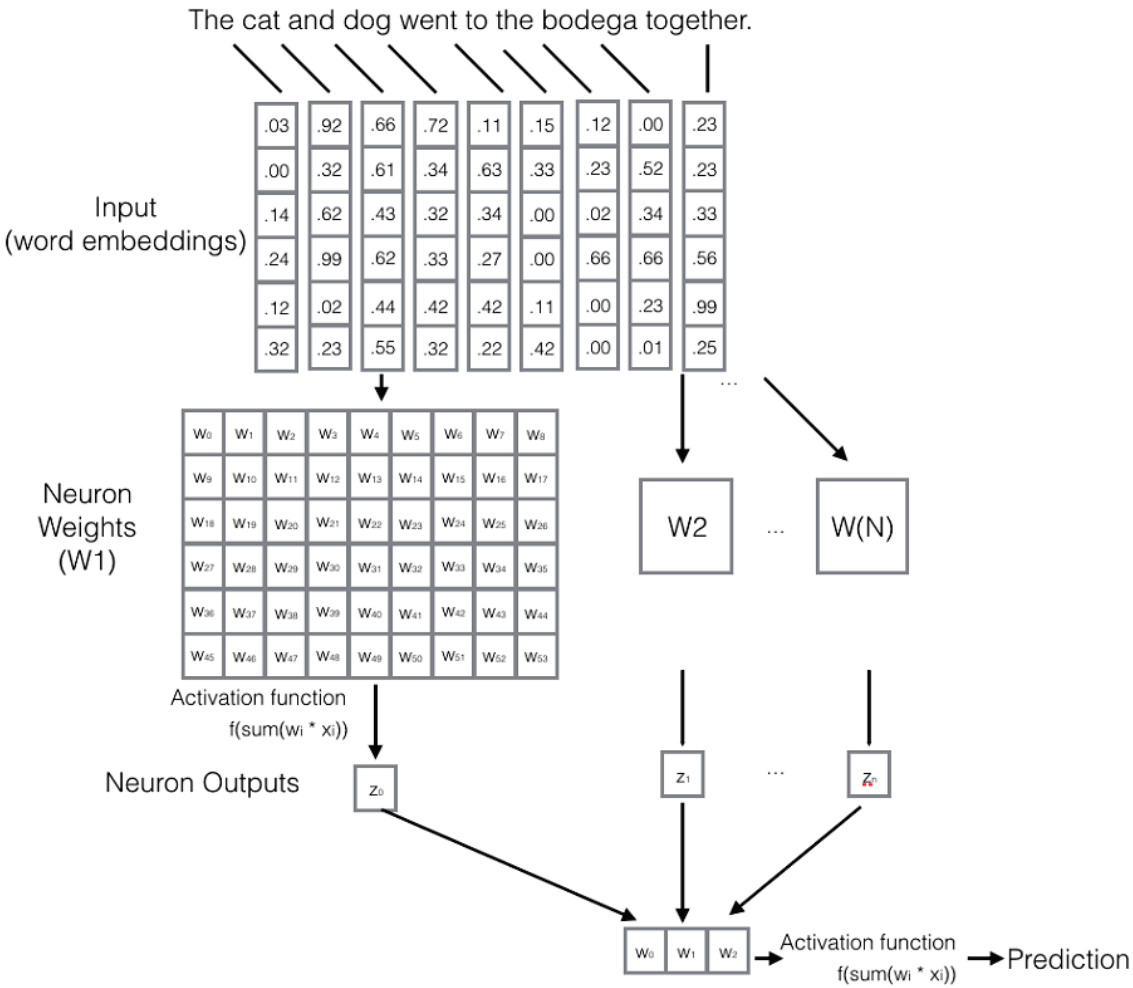


Figure 8.2 Text into a Feed-Foward Network

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 relationship of a 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. In this chapter, we will look at a different way. And 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. If we can find a way to *remember* what just happened in the moment before (time step n, for time step n+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 way we remember.

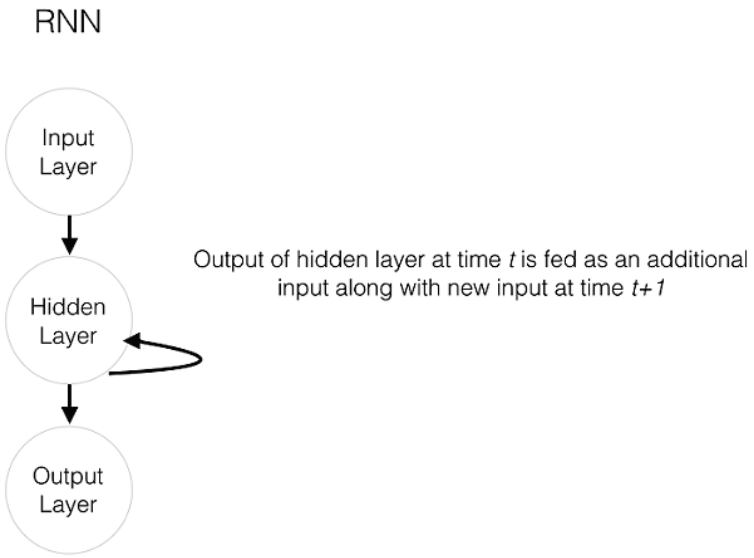


Figure 8.3 Recurrent Neural Net

The basic concept is simple. For each input we tell the feed-forward network what happened before along with what is happening "now".

Sometimes this is represented with an arc from the output of a layer back into it own inputs, as above, but more commonly this is shown, by *unrolling* the net.

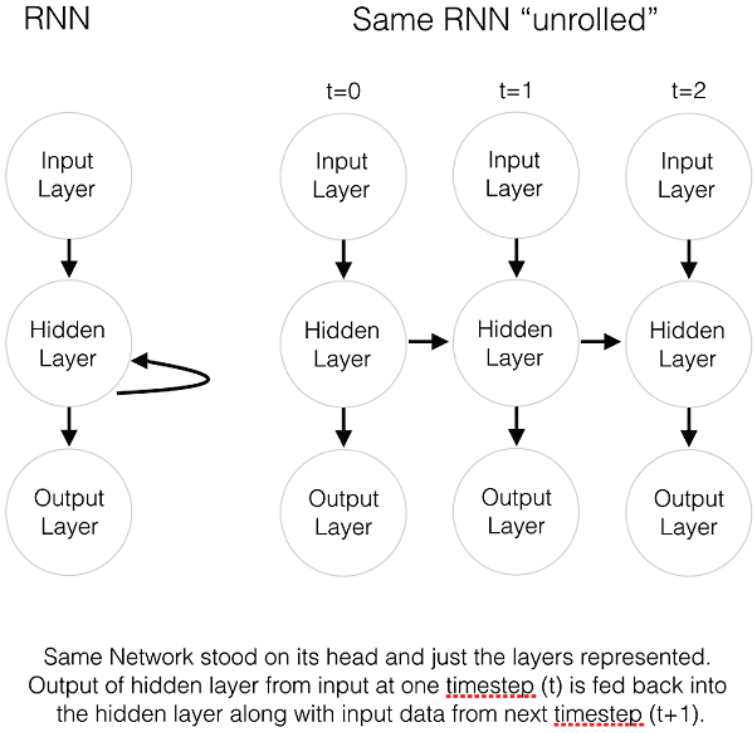
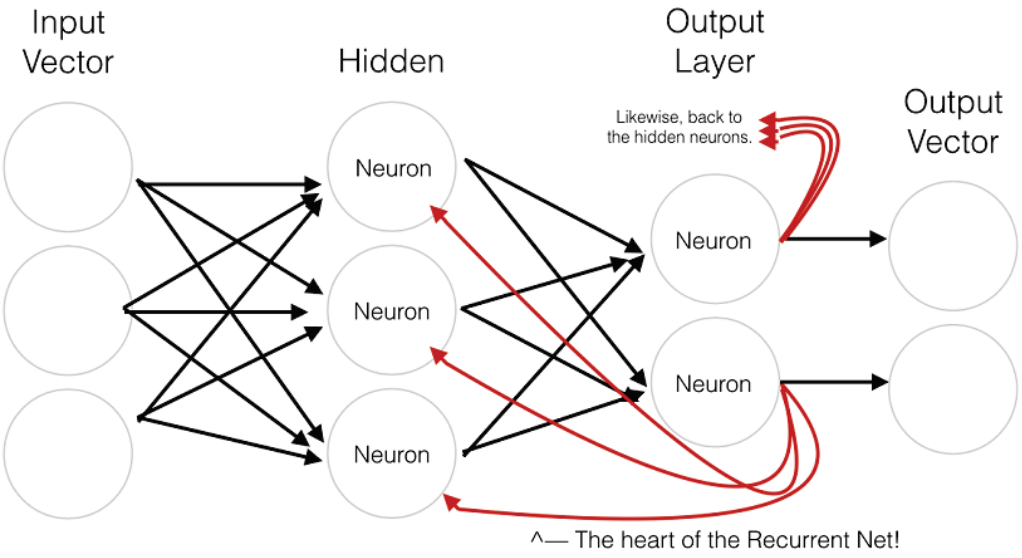


Figure 8.4 Unrolled Recurrent Neural Net

Each time step represented in the unrolled version of the net is the *same* network. This visualization is indeed helpful, especially when we talk about how information flows through the network forward and during backpropagation, but it is easy to lose sight of the fact that this is just one network with one set of weights. If we explode the

layer back out to the original representation of a neural network, we get something that look like this:



In a Fully-Connected Network, all nodes feed into each node in the neuron in the next layer, but with a trick.

Figure 8.5 Detailed Recurrent Neural Net

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. 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.

So now we are remembering! Sort of. Well. There are a couple of things we need to figure out. How does backpropagation even work in a structure like this. We are outputting information in a time step and basing the output of the next time step on information from the first. No problem, that's why we invented time machines. Life got much easier after that … before that … uh, well, anyway.

8.2 Backpropagation Through Time

So we know 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. Pause. What is the output of this unrolled contraption? The answer is, it depends. For a given sequence of tokens we may care about only the final output of the last time step from an input sequence.

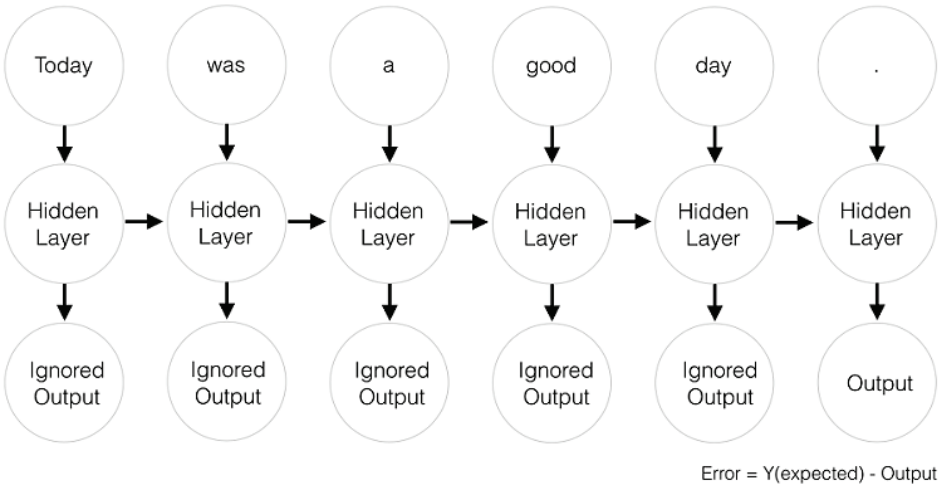
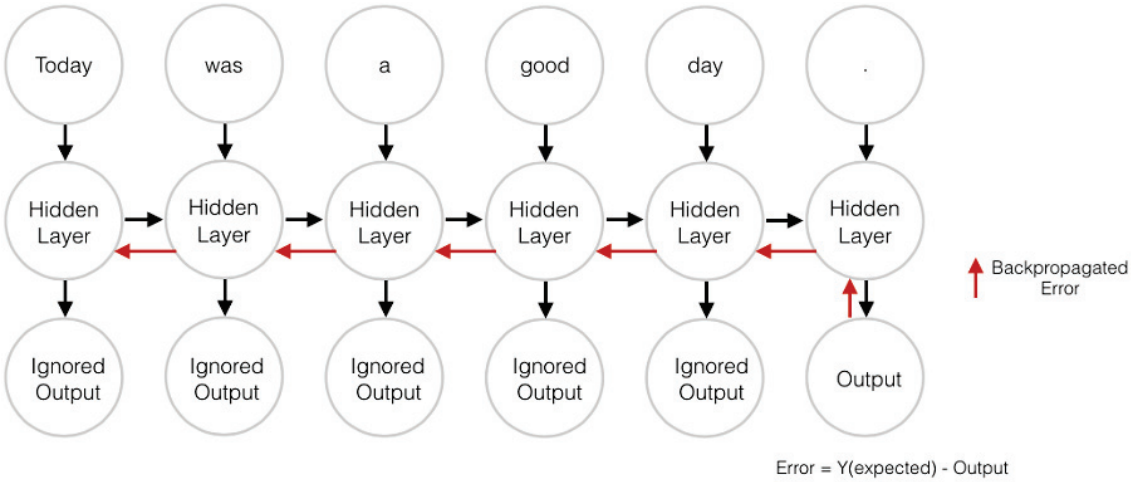


Figure 8.6 Only Last Output Matters Here

If this is the case we pass the entire sequence into the unrolled RNN, passing the hidden state's output into the next time step's version of the net at each step. At the final step in the sequence (t) we take the error and backpropagate it to the hidden state of the last time step as normal. And then we jump in our handy time machine and as in a stacked feed-forward net, 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. The math is the same.

8.2.1 When Do We Update What?

The tricky part of the update process is the weights we are updating are the SAME for each time step. This is just an *unrolled* version of the net, so we calculate the update for the weights of the hidden state at time *t* based on the error. We then backpropagate with how much the weights in the hidden state at time *t-1* and update the same weights again. The same weights we just updated. We do this all the way back down the chain to time *t=0*.

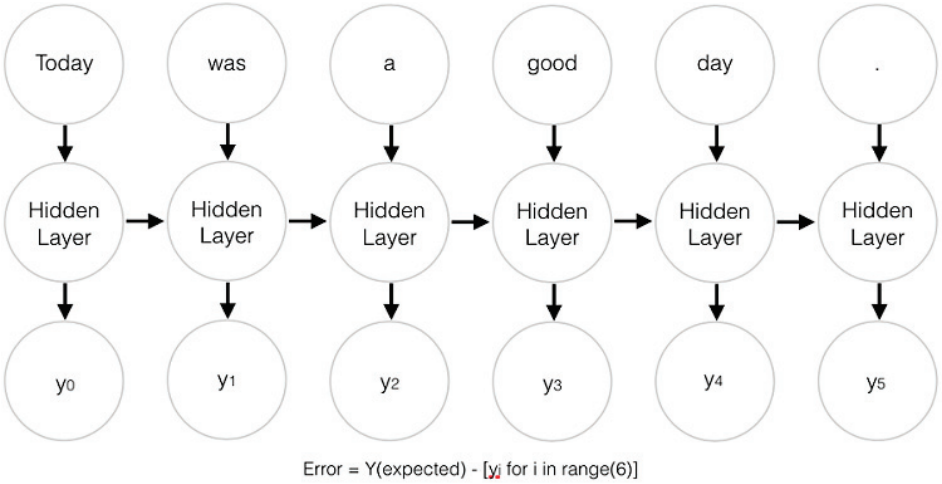


The error from the last step is backpropagated back in time. Each "older" step the gradient with respect to the newer step is taken. The changes are aggregated and applied to the single set of weights after all have been calculated back to t=0

Figure 8.7 Backpropagation Through Time

That last statement is not precisely true, but we've found it handy to help create the mental image of what goes on in an RNN. In actuality, the weight corrections are calculated at each time step, but not immediately updated. The gradient calculations need to be based on the values that the weights had when they contributed that much to the error. So we figure out the various changes to the weights at each time step and then sum up the changes and apply the aggregated changes to the weights of the hidden layer as the last step of the learning phase. As with any Neural Network this can happen after each training sample or in batches as well.

But we may care about the entire sequence that is generated by each of the intermediary time steps as well.

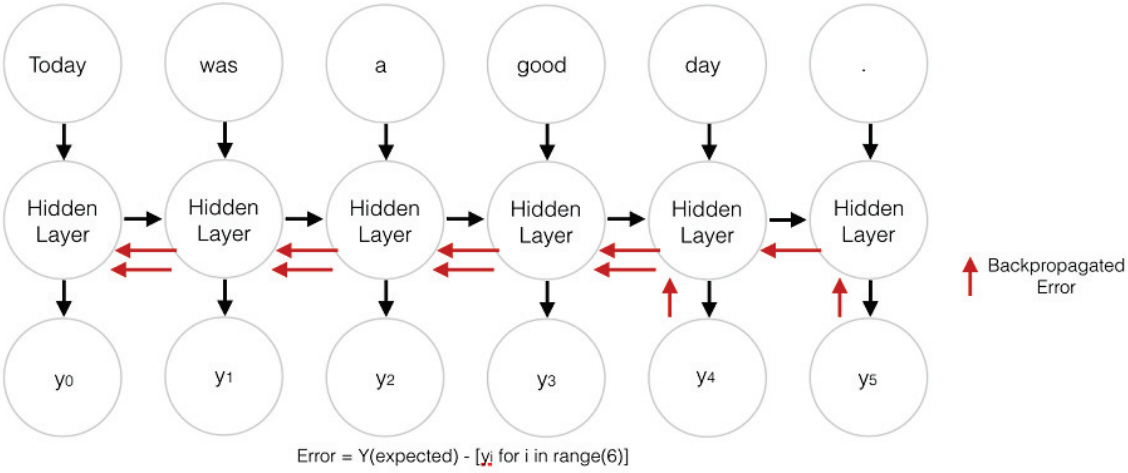


Or we can collect the sequence and calculate the error from an expected sequence. The error at each timestep is separately backpropagated through time.

Figure 8.8 All Outputs Matter Here

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 *t-1* 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 to the weights of the hidden layer.



The error is backpropagated from each output all the way back to *t=0* and aggregated before finally applying changes to the weights.

Figure 8.9 Multiple Outputs and Backpropagation Through Time

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). Updating the weights earlier would "pollute" the calculations of the backpropagations earlier in time.

8.2.2 There's Always a Catch

While an RNN 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, that many more derivatives to calculate. This doesn't mean they 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.

8.2.3 RNN 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 off in an ordered set. And finally we split it 80/20 into the training and test sets.

```
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 as always 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 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 RNN (not-simple is in the next chapter) and for each token gather 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. 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. 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, as we will see in a few pages, 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 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 I'm excited for the next chapter? Hang in there, all of this critical to understanding the next part).

8.3 Putting Things Together

We compile just as we did with the Convolutional 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. 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.

```
model.compile('rmsprop', 'binary_crossentropy',  metrics=['accuracy'])
print(model.summary())
```

```
Using Theano backend.
Build model...
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 400, 50)           17550
_________________________________________________________________
dropout_1 (Dropout)          (None, 400, 50)           0
```

```
_________________________________________________________________
flatten_1 (Flatten)          (None, 20000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 20001
=================================================================
Total params: 37,551.0
Trainable params: 37,551.0
Non-trainable params: 0.0
_________________________________________________________________
None
```

In the SimpleRNN layer, we requested 50 neurons. Each of those neurons will receive input (and a 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 vectors 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 (bias weights) = 15050

15,050 weights in the first time step of the first layer. Now each of those 50 neurons will feed it's 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*). 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 of vanishing gradients, but even this is still 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 input elements + 1 bias unit) * 1 neuron = 20,001 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.

8.4 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_model1.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 …

8.5 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 # Arbitrary sequence length based on perusing the data.
embedding_dims = 300 # This came from the pre-trained word2vec model
batch_size = 32 # Number of sample sequences to pass through (and aggregate the error)
                  before backpropagating
epochs = 2
num_nuerons = 50 # 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 in the next chapter 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 approve 2 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 Theano 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 adjust two at a time, but of course this sends the complexity of the task through the roof. 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.

8.6 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')

# 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.
vec_list = tokenize_and_vectorize([(1, sample_1)])

# Tokenize returns a list of the data (length 1 here)
test_vec_list = pad_trunc(vec_list, maxlen)
```

```
test_vec = np.reshape(test_vec_list, (len(test_vec_list),
    maxlen, embedding_dims))model.predict_classes(test_vec)
```

```
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 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 an RNN 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". [87]

Footnote 87 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.

The are times when you want to remember from one input sample to the next, not just time step to time step 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 the nature of the text 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.

8.7 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 an 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 rnns* come in. Keras has recently added a layer wrapper that will automatically flip and the necessary inputs and outputs and simulateously calculate a bi-directional RNN for us.

```
from keras.layers.wrappers import Bidirectional
```

and

```
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 (this part can be handled in Keras with the go_backward keyword in the RNN layer of the model definition).

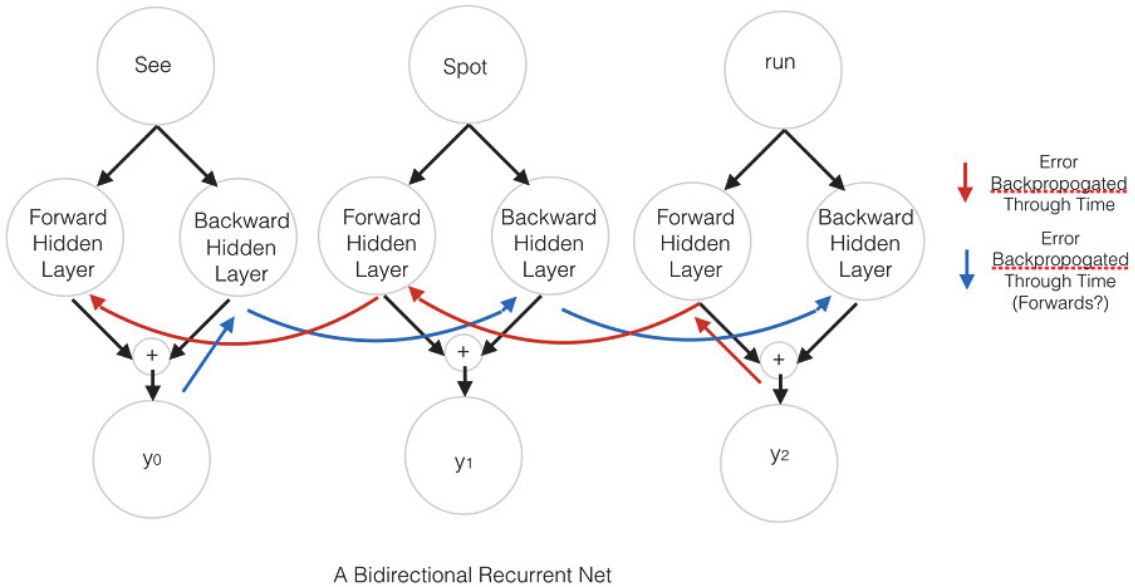


Figure 8.10 BiDirectional Recurrent Neural Net

With these tools we are well on our 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!

8.8 Summary

In this chapter we started remembering things:

- In sequence data like language, what came before is important
- We can backpropagate error back in time
- 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 backward and forward propagate AND backpropagate simultaneously