

A photograph of a woman in traditional, possibly Bulgarian, folk clothing. She is wearing a white blouse with a yellow sash and a dark blue apron over a green dress. She has a colorful headband and is holding a large red sign with the letters "MEAP" written in white, stylized, blocky font.

Natural Language Processing **IN ACTION**

SECOND EDITION

Hobson Lane
Maria Dyshel



MANNING



Natural Language Processing IN ACTION

SECOND EDITION

Hobson Lane
Maria Dyshel

MEAP

MANNING

Natural Language Processing in Action, Second Edition MEAP V09

1. [MEAP VERSION 9](#)
2. [Welcome](#)
3. [1 Machines that read and write \(NLP overview\)](#)
4. [2 Tokens of thought \(natural language words\)](#)
5. [3 Math with words \(TF-IDF vectors\)](#)
6. [4 Finding meaning in word counts \(semantic analysis\)](#)
7. [5 Word brain \(neural networks\)](#)
8. [6 Reasoning with word embeddings \(word vectors\)](#)
9. [7 Finding kernels of knowledge in text with Convolutional Neural](#)
10. [8 Reduce, reuse, recycle your words \(RNNs and LSTMs\)](#)
11. [9 Stackable deep learning \(Transformers\)](#)

A detailed illustration of a woman from the waist up, wearing a white blouse with a yellow sash and a dark blue skirt. She is holding a large red rectangular sign with the letters "MEAP" written in white, torn-edged font.

Natural Language Processing IN ACTION

SECOND EDITION

Hobson Lane
Maria Dyshel

 MANNING

MEAP VERSION 9



Welcome

Thank you for supporting *Natural Language Processing in Action - 2nd edition* with your purchase of the MEAP.

Natural Language Processing may be the fastest-developing and most important field of Artificial Intelligence and Data Science. If you want to change the world you will need to understand how machines read and process natural language text. That's what we hope to do with this latest edition of this book. We are going to show you how to change the world for the better using prosocial Natural Language Processing. This book will show you how to build machines that understand and generate text almost as well as a human, in many situations.

Immediately after the first edition of *NLPiA* was published, we started seeing the technologies we used in it become outdated. Faster more powerful algorithms and more prosocial applications for NLP were being released each year. BERT was released in 2019 and then GPT-3 in 2020. Inspired by a renewed sense of urgency the ethical AI and open source AI community quickly released GPT-J (GPT-J-6B) in responded to less-than-prosocial applications of the proprietary GPT-3 and Codex models. These ground-breaking models are based on the Transformer architecture, so we've added an entire chapter to help democratize utilization and understanding of this powerful technology.

And the demonstrations you've seen for transformers likely included some form of conversational AI. Finally machines are able to carry on a reasonably coherent conversation within a limited domain. All that is required for these models to perform well are large amounts of compute power and training data. Advances such as sparse attention and GPT-J are rapidly improving the efficiency of these transformer architectures to soon be within reach of the individual practitioner not becoming to Big Tech. In addition, promoters of proprietary GPT-based models often gloss over the biases and brittleness of Transformers. So some contributing authors have provided insights into creating more robust and fair NLP pipelines.

In addition, chatbots and conversational AI has emerged as a critical tool in the influence of the collective consciousness and a useful tool for changing the world. In 2018, when the first edition was written, chatbots were still recovering from the bust of the chatbot bubble of 2016-2017. The recent plunge into a once-in-a-century global pandemic has created renewed urgency among organizations seeking to communicate with their customers, employees, and beneficiaries in human language. Chatbots served millions in providing COVID-19 information, onboarding training as employees switched jobs, and benefits information for people reliant on the social safety net. Chatbots help fill the gap as conventional in-person customer service chains were disrupted. For us, who are passionate about the potential of virtual assistants, it was a sign that we are on the right track. We are doubling down on that daring bet of the first edition by further democratizing NLP and prosocial conversational AI. With that proliferation of a diverse "gene pool" of prosocial NLP algorithms we hope some will emerge to outcompete the dark pattern alternatives.

In the second edition we are rewriting every chapter to incorporate the understanding of a new team of authors and contributing authors. We have incorporated our conversational AI expertise into the fabric of the book with chatbot code examples in nearly every chapter. We added review questions and exercises at the end of each chapter to help in your active learning. We also upgraded and simplified the core python packages we use for all of the code in the 2nd Edition:

1. NLTK and Gensim upgraded to **SpaCy**
2. Keras and TensorFlow upgraded to **PyTorch**
3. Added **HuggingFace** transformers and sentence-transformers

Not only do we like the advanced features of SpaCy and PyTorch but we also like their more vibrant and prosocial open source communities. These more powerful communities and packages will stand the test of time.

Though we rewrote nearly every chapter we kept the structure the same. In part 1, you will start by learning about the very building blocks of natural language: characters, tokens, lemmas, sentences and documents. You will learn to represent natural language text as numerical vectors so they can be processed using the computer-friendly language of mathematics and linear

algebra. And you will learn how to use this vector representation of text for classifying, searching and analyzing natural language documents.

Part 2 will introduce the artificial **brain** that boosted NLP's and brought it to the forefront of AI today: **Artificial Neural Networks or Deep Learning**. Together, we will explore the concept of neural networks from the very basics. You will build layer upon layer of neurons onto that foundation, deepening your understanding of the latest network architectures, from fully connected feed-forward networks to transformers. In part 3, we'll dive into the production application and scaling of natural language processing technology.

With you the reader, we are working hard to ensure that the technology explosion sparked by advances in AI and NLP will end well for **all** humans and not just a select few. With this updated and upgraded version of NLPIA you will join a vibrant global community on a quest to develop AI that interacts with humans in prosocial ways - truly **beneficial AI**.

We believe in the power of the community, especially one where prosocial algorithms and NL interfaces are helping steer the conversation. We know that the collective intelligence of our readers is what will take this book to the next level. Co-create this book along side us with your questions, [comments and suggestions on liveBook!](#)

— Maria Dyshel and Hobson Lane

In this book

[MEAP VERSION 9](#) [About this MEAP](#) [Welcome](#) [Brief Table of Contents](#) [1 Machines that read and write \(NLP overview\)](#) [2 Tokens of thought \(natural language words\)](#) [3 Math with words \(TF-IDF vectors\)](#) [4 Finding meaning in word counts \(semantic analysis\)](#) [5 Word brain \(neural networks\)](#) [6 Reasoning with word embeddings \(word vectors\)](#) [7 Finding kernels of knowledge in text with Convolutional Neural Networks \(CNNs\)](#) [8 Reduce, reuse, recycle your words \(RNNs and LSTMs\)](#) [9 Stackable deep learning \(Transformers\)](#)

1 Machines that read and write (NLP overview)

This chapter covers

- The power of human language
- How natural language processing (NLP) is changing society
- The kinds of NLP tasks that machines can now do well
- Why unleashing the NLP genie is profitable ... and dangerous
- How to start building a simple chatbot
- How NLP technology is programming itself and making itself smarter

Words are powerful. They can change minds. And they can change the world. Natural language processing puts the power of words into algorithms. In fact, those algorithms are changing your world right before your eyes. You are about to see how the majority of the words and ideas that enter your mind are filtered and generated by NLP.

Imagine what you would do with a machine that could understand and act on every word it reads on the Internet? Imagine the information and knowledge you'd be able to harvest and profit from. NLP promises to create the second information revolution by turning vast amounts of unstructured data into actionable knowledge and understanding.

Early on, Big Tech discovered this power of NLP to harvest knowledge from natural language text. They use that knowledge to affect our behavior and our minds in order to improve their bottom line.^[1] Governments too are waking up to the impact NLP has on culture, society and humanity. Some advanced liberal democracies are attempting to free your mind by steering business towards sustainable and ethical uses for NLP.

On the other end of the spectrum, authoritarian governments are using NLP to coopt our prosocial instincts to make us easier to track and control. The Chinese government uses NLP to prevent you from even talking about Tibet

or Hong Kong in the games you play.^[2] And governments that censor public media are beginning to affect even the most benign and open source AI algorithms by corrupting NLP training sets.^[3] And surprisingly, even the US is using NLP to control the public discourse about COVID-19, endangering countless lives.^[4]

In this chapter you will begin to build your NLP understanding and skill so you can take control of the information and ideas that affect what you believe and think. You first need to see all the ways NLP is used in the modern world. This chapter will open your eyes to these NLP applications happening behind the scenes in your everyday life. This will help you write software to track, classify, and influence the packets of thought bouncing around on the Internet. Your understanding of natural language processing will give you greater influence and control over the words and ideas in your world. And it will give you and your business the ability to escape Big Tech's stranglehold on information, so you can succeed.

1.1 Programming language vs. natural language

Programming languages are very similar to natural languages like English. Both kinds of languages are used to communicate instructions from one information processing system to another—human to human or human to computer. Both employ grammars to define what acceptable or meaningful statements look like. And these grammars employ vocabularies that define the tokens with shared meaning for the agents that process these languages—humans or machines.

Despite these similarities, programming languages are quite different from natural languages. Programming languages are artificially designed languages we use to tell a computer what to do. Computer programming languages are used to explicitly define a sequence of mathematical operations on bits of information, ones and zeros. And programming languages only need to be *processed* by machines rather than *understood*. A machine needs to do *what* the programmer asks it to do. It does not need to understand *why* the program is the way it is. And it doesn't need abstractions or mental models of the computer program to understand anything outside of the world of ones and zeroes that it is processing. And almost all computers use the Von Neumann

architecture developed in 1945.^[5] Modern CPUs (Central Processing Units) implement the *Von Neumann architecture* as a register machine, a version of the *universal Turing machine* idea of 1936.^[6]

Natural languages, however, *evolved naturally, organically*. Natural languages communicate ideas, understanding, and knowledge between living organisms that have brains rather than CPUs. These natural languages must be "runnable" or *understandable* on a wide variety of wetware (brains). In some cases natural language even enables communication across animal species. Koko (gorilla), Woshoe (chimpanzee), Alex (parrot) and other famous animals have demonstrated command of some English words.^[7] Reportedly, Alex discovered the meaning of the word "none" on its own. Alex's dying words to its grieving owner were "Be good, I love you" (<https://www.wired.com/2007/09/super-smart-par>). And Alex's words inspired Ted Chiang's masterful short story "The Great Silence"—that's profound cross-species communication.

Given how differently natural languages and programming languages evolved, it is no surprise they're used for different things. We do not use programming languages to tell each other about our day or to give directions to the grocery store. Similarly, natural languages did not evolve to be readily compiled into thought packets that can be manipulated by machines to derive conclusions. But that's exactly what you are going to learn how to do with this book. With NLP you can program machines to process natural language text to derive conclusions, infer new facts, create meaningful abstractions, and even respond meaningfully in a conversation.

Even though there are no compilers for natural language there are *parsers* and *parser generators*, such as PEGN^[8] and SpaCy's Matcher class. But there aren't any end-to-end compilers that can generate responses or perform the natural language instructions. And SpaCy allows you to define patterns and grammars similar to the ones There is no algorithm that takes natural language text and turns it into machine instructions to brew a cup of coffee, even when the natural language text consisted of an explicit list of brewing steps. There is single silver bullet that you can use to programmatically turn natural language text into computer instructions or a conversational response. But if you understand all the different tools described in this book, you will

be able to combine them all together to create remarkably intelligent conversational chatbots that understand and generate meaningful text.



Natural language processing

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

This chapter shows you how your software can *process* natural language to produce useful output. You might even think of your program as a natural language interpreter, similar to how the Python interpreter processes source code. When the computer program you develop processes natural language, it will be able to act on those statements or even reply to them.

Unlike a programming language where each keyword has an unambiguous interpretation, natural languages are much more fuzzy. This fuzziness of natural language leaves open to you the interpretation of each word. So, you get to choose how the bot responds to each situation. Later you will explore advanced techniques in which the machine can learn from examples, without you knowing anything about the content of those examples.



Pipeline

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

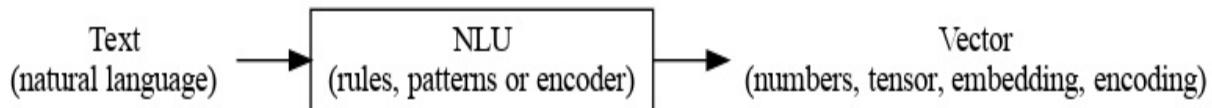
You will soon have the power to write software that does interesting, human-like things with text. This book will teach you how to teach machines to carry on a conversation. It may seem a bit like magic, as new technology often does, at first. But you will pull back the curtain and explore the technology

behind these magic shows. You will soon discover all the props and tools you need to do the magic tricks yourself.

1.1.1 Natural Language Understanding (NLU)

A really important part of NLP is the automatic processing of text to extract a numerical representation of the *meaning* of that text. This is the *natural language understanding* (NLU) part of NLP. The numerical representation of the meaning of natural language usually takes the form of a vector called an embedding. Machines can use embeddings to do all sorts of useful things. Embeddings are used by search engines to understand what your search query means and then find you web pages that contain information about that topic. And the embedding vectors for emails in your inbox are used by your email service to classify those emails as Important or not.

Figure 1.1. Natural Language Understanding (NLU)



Machines can accomplish many common NLU tasks with high accuracy:

- semantic search
- text alignment (for translation or plagiarism detection)
- paraphrase recognition
- intent classification
- authorship attribution

And recent advances in deep learning have made it possible to solve many NLU tasks that were impossible only ten years ago:

- analogy problem solving
- reading comprehension
- extractive summarization
- medical diagnosis based on symptom descriptions

However, there remain many NLU tasks where humans significantly out-

perform machines. Some problems require the machine to have common sense knowledge, learn the logical relationships between those common sense facts, and to use all of this on the context surrounding a particular piece of text. This makes these problems much more difficult for machines:

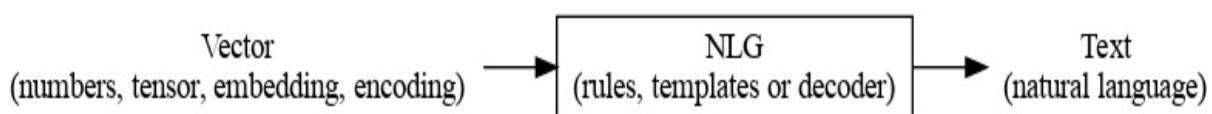
- euphemism & pun recognition
- humor & sarcasm recognition
- hate-speech & troll detection
- logical entailment and fallacy recognition
- database schema discovery
- knowledge extraction

You'll learn the current state of the art approaches to NLU and what is possible for these difficult problems. And your *behind the scenes* understanding of NLU will help you increase the effectiveness of your NLU pipelines for your particular applications, even on these hard problems.

1.1.2 Natural Language Generation (NLG)

You may not be aware that machines can also compose text that sounds human-like. Machines can create human-readable text based on a numerical representation of the meaning and sentiment you would like to convey. This is the *natural language generation* (NLG) side of NLP.

Figure 1.2. Natural Language Generation (NLG)



You will soon master many common NLG tasks.

- synonym substitution
- search-based question answering (information retrieval)
- spelling and grammar correction
- frequently-asked question answering
- casual conversation

And even the more advanced NLG tasks will soon be within your reach.

- machine translation
- sentence summarization and simplification
- sentence paraphrasing
- therapeutic conversational AI
- factual question generation
- discussion facilitation and moderation
- debate

And this will give you the foundation to customize your NLG pipeline for even the most challenging NLG tasks.

- compose poetry and song lyrics
- compose jokes and sarcastic comments
- generate text that fools NLU pipelines into doing what you want
- measure the robustness of NLP pipelines
- automatically summarize long technical documents
- compose programming language expressions from natural language

This last development in NLG is particularly powerful. Machines can now write correct code to match your intent based only on a natural language description.

The combination of NLU and NLG will give you the tools to create machines that interact with humans in surprising ways.^[9]

1.1.3 Plumbing it all together for positive impact

Once you understand how NLG and NLU work, you will be able to assemble them into your own NLP pipelines, like a plumber. Businesses are already using pipelines like these to extract value from their users.

You too can use these pipelines to further *your* own objectives in life, business, and social impact. This technology explosion is a rocket that you can ride and maybe steer a little bit. You can use it in your life to handle your inbox and journals while protecting your privacy and maximizing your mental well-being. Or you can advance your career by showing your peers

how machines that understand and generate words can improve the efficiency and quality of almost any information-age task. And as an engineer who thinks about the impact of your work on society, you can help nonprofits build NLU and NLG pipelines that lift up the needy. As an entrepreneur you can help create a regenerative prosocial business that spawn whole new industries and communities that thrive together.

And understanding how NLP works will open your eyes and empower you. You will soon see all the ways machines are being used to mine your words for profit, often at your expense. And you will see how machines are training you to become more easily manipulated. This will help you insulate yourself, and perhaps even fight back. You will soon learn how to survive in a world overrun with algorithms that manipulate you. You will harness the power of NLP to protect your own well-being and contribute to the health of society as a whole.

Machines that can understand and generate natural language harness the power of words. Because machines can now understand and generate text that seems human, they can act on your behalf in the world. You'll be able to create bots that will automatically follow your wishes and accomplish the goals you program them to achieve. But, beware Aladdin's Three Wishes trap. Your bots may create a tsunami of blowback for your business or your personal life. Be careful about the goals you give your bots.^[10] Like the age old three wishes problem, you may find yourself trying to undo all the damage caused by your earlier wishes and bots.

[1] In 2013 The Guardian and other news organizations revealed Facebook's experiments to maniuplate users' emotions using NLP (<https://www.theguardian.com/technology/2014/jun/29/facebook-users-emotions-news-feeds>). Search engine giants and their algorithms perform these same kinds of experiments each time you enter text into the search box (<https://www.computerservicesolutions.in/all-google-search-algorithm-updates/>).

[2] "Genshin Impact won't let players write 'Tibet', 'Hong Kong', 'Taiwan' because of Chinese censorship" (<https://www.msn.com/en-us/news/technology/genshin-impact-won-t-let-players-write-tibet-hong-kong->

[taiwan-because-of-chinese-censorship/ar-BB19MQYE](#))

[3] "Censorship of Online Encyclopedias Implications for NLP Models"
https://www.researchgate.net/publication/348757384_Censorship_of_Online

[4] Lex Frideman interview of Bret Weinstein titled "Truth, Science, and Censorship in the Time of a Pandemic" (<https://lexfridman.com/bret-weinstein/>)

[5] Von Neumann Architecture on Wikipedia
https://en.wikipedia.org/wiki/Von_Neumann_architecture)

[6] "The secrets of computer power revealed" by Daniel Dennett
<https://sites.tufts.edu/rodrego/>)

[7] Animal Language" on Wikipedia
https://en.wikipedia.org/wiki/Animal_language)

[8] Parsing Expression Grammar Notation home page (<https://pegn.dev/>)

[9] You may have heard of Microsoft's and OpenAI's Copilot project. GPT-J can do almost as well, and it's completely open source and open data.
<https://huggingface.co/models?sort=likes&search=gpt-j>)

[10] *Human Compatible AI* by Stuart Russell

1.2 The magic

What is so magical about a machine that can read and write in a natural language? Machines have been processing languages since computers were invented. But those were computer languages, such as Ada, Bash, and C, designed to prevent ambiguity. Computer languages can only be interpreted (or compiled) in one correct way. With NLP we can talk to machines in our own language. When software can process languages not designed for machines to understand, it is magic—something we thought only humans could do.

Moreover, machines can access a massive amount of natural language text, such as Wikipedia, to learn about the world and human thought. Google's index of natural language documents is well over 100 million gigabytes,[\[11\]](#) and that is just the index. And that index is incomplete. The size of the actual natural language content currently online probably exceed 100 billion gigabytes.[\[12\]](#) This massive amount of natural language text makes NLP a useful tool.



Note

Today, Wikipedia lists approximately 700 programming languages. Ethnologue_ [\[13\]](#) identifies more than 7,000 natural languages. And that doesn't include many other natural language sequences that can be processed using the techniques you'll learn in this book. The sounds, gestures, and body language of animals as well as the DNA and RNA sequences within their cells can all be processed with NLP.[\[14\]](#)[\[15\]](#)

Machines with the capability to process something natural is not natural. It is kind of like building a building that can do something useful with architectural designs. When software can process languages not designed for machines to understand, it seems magical—something we thought was a uniquely human capability.

For now you only need to think about one natural language— English. You'll ease into more difficult languages like Mandarine Chinese later in the book. But you can use the techniques you learn in this book to build software that can process any language, even a language you do not understand or has yet to be deciphered by archaeologists and linguists. We are going to show you how to write software to process and generate that language using only one programming language, Python.

Python was designed from the ground up to be a readable language. It also exposes a lot of its own language processing "guts." Both of these characteristics make it a natural choice for learning natural language processing. It is a great language for building maintainable production pipelines for NLP algorithms in an enterprise environment, with many

contributors to a single codebase. We even use Python in lieu of the "universal language" of mathematics and mathematical symbols, wherever possible. After all, Python is an unambiguous way to express mathematical algorithms, [\[16\]](#) and it is designed to be as readable as possible by programmers like you.

1.2.1 Language and thought

Linguists and philosophers such as Sapir and Whorf postulated that our vocabulary affects the thoughts we think. For example Australian Aborigines have words to describe the position of objects on their body according to the cardinal points of the compass. They don't talk about the boomerang in their right hand, they talk about the boomerang on the north side of their body. This makes them adept at communicating and orienteering during hunting expeditions. Their brains are constantly updating their understanding of their orientation in the world.

Stephen Pinker flips that notion around and sees language as a window into our brains and how we think: "Language is a collective human creation, reflecting human nature, how we conceptualize reality, how we relate to one another."[\[17\]](#) Whether you think of words as affecting your thoughts or as helping you see and understand your thoughts, either way, they are packets of thought. You will soon learn the power of NLP to manipulate those packets of thought and amp up your understanding of words, ... and maybe thought itself. It's no wonder many businesses refer to NLP and chatbots as AI - Artificial Intelligence.

What about math? We think with precise mathematical symbols and programming languages as well as with fuzzier natural language words and symbols. And we can use fuzzy words to express logical thought like mathematics concepts, theorems, and proofs. But words aren't the only way we think. Jordan Ellenberg, a geometer at Harvard, writes in his new book *Shape* about how he first "discovered" the commutative property of algebra while staring at a stereo speaker with a grid of dots, 6x8. He'd memorized the multiplication table, the symbols for numbers. And he knew that you could reverse the order of symbols on either side of a multiplication symbol. But he didn't really *know* it until he realized that he could visualize the 48 dots as 6

columns of 8 dots, or 8 rows of 6 dots. And it was the same dots! So it had to be the same number. It hit him at a deeper level, even deeper than the symbol manipulation rules that he learned in algebra class.

So you use words to communicate thoughts with others and with yourself. When ephemeral thoughts can be gathered up into words or symbols, they become compressed packets of thought that are easier to remember and work with in your brain. You may not realize it, but as you are composing sentences you are actually rethinking and manipulating and repackaging these thoughts. What you want to say, and the idea you want to share is crafted while you are speaking or writing. This act of manipulating packets of thought in your mind is called "symbol manipulation" by AI researchers and neuroscientists. In fact, in the age of GOFAI (Good Old-Fashioned AI) researchers assumed that AI would need to learn to manipulate natural language symbols and logical statements the same way it compiles programming languages. In this book you're going to learn how to teach a machine to do symbol manipulation on natural language in chapter 11.

But that's not the most impressive power of NLP. Think back to a time when you had a difficult e-mail to send to someone close. Perhaps you needed to apologize to a boss or teacher, or maybe your partner or a close friend. Before you started typing, you probably started thinking about the words you would use, the reasons or excuses for why you did what you did. And then you imagined how your boss or teacher would perceive those words. You probably reviewed in your mind what you would say many many times before you finally started typing. You manipulated packets of thought as words in your mind. And when you did start typing, you probably wrote and rewrote twice as many words as you actually sent. You chose your words carefully, discarding some words or ideas and focusing on others.

The act of revision and editing is a thinking process. It helps you gather your thoughts and revise them. And in the end, whatever comes out of your mind is not at all like the first thoughts that came to you. The act of writing improves how you think, and it will improve how machines think as they get better and better at reading and writing.

So reading and writing is thinking. And words are packets of thought that you can store and manipulate to improve those thoughts. We use words to put

thoughts into clumps or compartments that we can play with in our minds. We break complicated thoughts into several sentences. And we reorder those thoughts so they make more sense to our reader, or even our future self. Every sentence in this 2nd edition of the book has been edited several times - sometimes with the help of generous readers of the LiveBook. [\[18\]](#) I've deleted, rewritten and reordered these paragraphs several times just now, with the help of suggestions and ideas from friends and readers like you. [\[19\]](#)

But words and writing aren't the *only* way to think logically and deeply. Drawing, diagramming, even dancing and acting out. And we visually imagine these drawings in our minds—sketching ideas and concepts and thoughts in our head. And sometimes you just physically move things around or act things out in the real world. But the act of composing words into sentences and sentences into paragraphs is something that we do almost constantly.

Reading and writing is also a special kind of thought. It seems to compress our thoughts and make them easier to remember and manage within our heads. Once we know the perfect word for a concept, we can file it away in our minds. We don't have to keep refreshing it to understand it. We know that once we think of the word again, the concept will come flooding back and we can use it again.

This is all thinking or what is sometimes called *cognition*. So by teaching machines to understand and compose text, you are in some small way, teaching them to think. This is why people think of NLP as artificial intelligence (AI). And conversational AI is one of the most widely recognized and useful forms of AI

1.2.2 Machines that converse

Though you spend a lot of time working with words as packets of thought internally within your head, the real fun is when you use those words to interact with others. The act of conversations brings two (or more!) people into your thinking. This can create a powerful positive feedback loop that reinforces good ideas, and weeds out weak ones.

Words are critical to this process. They are our shared thought vocabulary. When you want to trigger a thought in another person's brain, all you need to do is to say the word that they will often understand some of the thought in your mind. For example, when you are feeling great pain, frustration or shock, you can use a curse word. And you can almost be guaranteed to cause that shock and discomfort to be conveyed to your listener or reader. That is the sole purpose of curse words—to shock (and awe?) your listener.



Note

There is *another_NLP* that takes this idea to the extreme. Neuro-linguistic programming (the *other_NLP*) is a pseudoscientific psychotherapy approach that claims to change your behavior through the use of words. It is essentially an expensive group therapy session. Because there is money to be made in claiming to help people achieve their life goals, this pseudoscience has taken on a cult status for the practitioners who teach it (preach it?).[\[20\]](#)

As with astrology, fortune telling, hypnotherapy, conspiracy theories, religions and cults, there is usually a small hint of truth somewhere within it. Words do indeed affect our thoughts. And thoughts do affect our behavior.

Though we cannot "program" another human with our words, we can use them to communicate extremely complex ideas. When you engage in conversation you are acting as a neuron in the collective consciousness, the hive mind. Unfortunately, when profit motives and unfettered competition is the rule of the day, the hornets nest of social media is the result.

Natural language cannot be directly translated into a precise set of mathematical operations. But natural language does contain information and instructions that can be extracted. Those pieces of information and instruction can be stored, indexed, searched, or immediately acted upon. One of those actions could be to generate a sequence of words in response to a statement. This is the function of the "dialog engine" or chatbot that you will build.

This book focuses entirely on English text documents and messages, not spoken statements. Chapter 7 does give you a brief foray into processing audio files, Morse code. But most of NLPIA is focused on the words that

have been put to paper... or at least put to transitors in a computer. There are whole books on speech recognition and speech to text (STT) systems and text to speech (TTS) systems. There are ready-made open source projects for STT and TTS. If your application is mobile, modern smartphone SDKs provide you with speech recognition and speech generation APIs. If you want your virtual assistant to live in the cloud, there are Python packages to accomplish SST and TTS on any Linux server with access to your audio stream.

In this book you will focus on what happens between the *ears* of the machine. This can help you build a smarter voice assistant when you add your *brains* to open source projects such as MyCrost AI [21] or OVAL Genie, [22]. And you'll understand all the helpful NLP that the big boys could be giving you within their voice assistants ... assuming commercial voice assistants wanted to help you with more than just lightening your wallet.

Speech recognition systems

If you want to build a customized speech recognition or generation system, that undertaking is a whole book in itself; we leave that as an "exercise for the reader." It requires a lot of high quality labeled data, voice recordings annotated with their phonetic spellings, and natural language transcriptions aligned with the audio files. Some of the algorithms you learn in this book might help, but most of the algorithms are quite different. [23]

1.2.3 The math

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

The techniques you will learn, however, are powerful enough to create machines that can surpass humans in both accuracy and speed for some surprisingly subtle tasks. For example, you might not have guessed that

recognizing sarcasm in an isolated Twitter message can be done more accurately by a machine than by a human.^[24] Do not worry, humans are still better at recognizing humor and sarcasm within an ongoing dialog because we are able to maintain information about the context of a statement. However, machines are getting better and better at maintaining context. This book helps you incorporate context (metadata) into your NLP pipeline if you want to try your hand at advancing the state of the art.

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

So extracting information is not at all like building a programming language compiler (fortunately for you). The most promising techniques bypass the rigid rules of regular grammars (patterns) or formal languages. You can rely on statistical relationships between words instead of a deep system of logical rules.^[25] Imagine if you had to define English grammar and spelling rules in a nested tree of if...then statements. Could you ever write enough rules to deal with every possible way that words, letters, and punctuation can be combined to make a statement? Would you even begin to capture the semantics, the meaning of English statements? Even if it were useful for some kinds of statements, imagine how limited and brittle this software would be. Unanticipated spelling or punctuation would break or befuddle your algorithm.

Natural languages have an additional "decoding" challenge that is even harder to solve. Speakers and writers of natural languages assume that a human is the one doing the processing (listening or reading), not a machine. So when I say "good morning," I assume that you have some knowledge about what makes up a morning, including that the morning comes before

noon, afternoon, and evening, but it also comes after midnight. You need to know that a morning can represent times of day as well as a general period of time. The interpreter is assumed to know that "good morning" is a common greeting, and that it does not contain much information at all about the morning. Rather, it reflects the state of mind of the speaker and her readiness to speak with others.

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

[11] See the web page titled, "How Google's Site Crawlers Index Your Site - Google Search" (<https://proai.org/google-search>).

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

[13] <http://ethnologue.com> maintains statistics about natural languages. ISO 639-3 lists 7,486 three-letter language codes (<http://proai.org/language-codes>).

[14] *The Great Silence* by Ted Chiang (<https://proai.org/great-silence>) describes an imagined dialog with an endangered species of parrot that concludes with the parrot saying to humanity, "Be Good. I love you."

[15] Dolphin Communication Project (<https://proai.org/dolphin-communication>)

[16] Mathematical notation is ambiguous. See the "Mathematical notation" section of the Wikipedia article "Ambiguity" (https://en.wikipedia.org/wiki/Ambiguity#Mathematical_notation).

[17] Thank you to "Tudor" on MEAP for setting me straight about this.

(https://www.ted.com/talks/steven_pinker_what_our_language_habits_reveal/)

[18] Thank you "Tudor" for improving this section and my thinking about linguistic relativism

[19] Thank you Leo Hepis!

[20] From the Wikipedia article on Neuro-linguistic-programming (https://en.wikipedia.org/wiki/Neuro-linguistic_programming)

[21] You can install MyCroft AI on any RaspberryPi with a speaker and a microphone (<https://mycroft.ai/>)

[22] Stanford's Open Virtual Assistant Lab within their Human-centered AI Institute (<https://hai.stanford.edu/news/open-source-challenger-popular-virtual-assistants>)

[23] Some open source voice assistants you could contribute to (<https://gitlab.com/tangibleai/team/-/tree/main/exercises/1-voice/>).

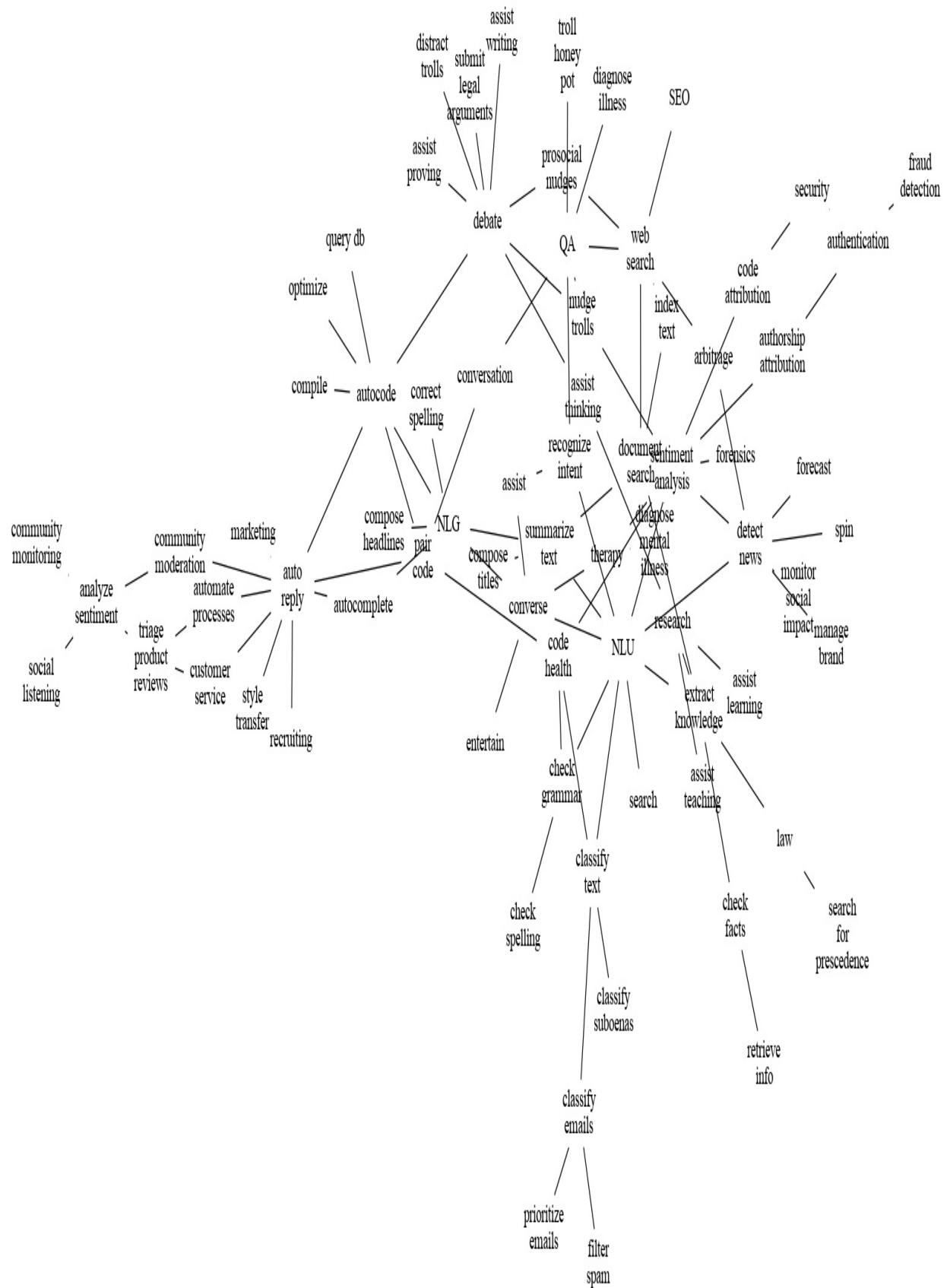
[24] Gonzalo-Ibanez et al found that educated and trained human judges could not match the performance of their simple classification algorithm of 68% reported in their ACM paper. The Sarcasm Detector (https://github.com/MathieuCliche/Sarcasm_detector) and web app (<https://www.thesarcasmdetector.com/>) by Matthew Cliche at Cornell achieves similar accuracy (>70%).

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

1.3 Applications

Natural language processing is everywhere. It is so ubiquitous that you'd have a hard time getting through the day without interacting with several NLP algorithms every hour. Some of the examples here may surprise you.

Figure 1.3. Graph of NLP applications



At the core of this network diagram are the NLU and NLG **sides** of NLP. Branching out from the NLU hub node are foundational applications like sentiment analysis and search. These eventually connect up with foundational NLG tools such as spelling correctors and automatic code generators to create conversational AI and even pair programming assistants.

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

NLP pipelines that generate text can be used not only to compose short replies in chatbots and virtual assistants, but also to assemble much longer passages of text. The Associated Press uses NLP "robot journalists" to write entire financial news articles and sporting event reports.^[26] Bots can compose weather forecasts that sound a lot like what your hometown weather person might say, perhaps because human meteorologists use word processors with NLP features to draft scripts.

More and more businesses are using NLP to automate their business processes. This can improve team productivity, job satisfaction, and improve the quality of the product. For example chatbots can automate the responses to many customer service requests.^[27] NLP spam filters in early email programs helped email overtake telephone and fax communication channels in the '90s. And some teams use NLP to automate and personalize e-mails between teammates or communicate with job applicants.

NLP pipelines, like all algorithms, make mistakes and are almost always biased in many ways. So if you use NLP to automate communication with humans, be careful. At Tangible AI we use NLP for the critical job of helping us find developers to join our team, so we were extremely cautious. We used NLP to help us filter out job applications only when the candidate was nonresponsive or did not appear to understand several questions on the

application. We had rigorous quality control on the NLP pipeline with periodic random sampling of the model predictions. We used simple models and sample-efficient NLP models [\[28\]](#) to focus human attention on those predictions where the machine learning was least confident—see the `predict_proba` method on SciKit Learn classifiers. As a result NLP for HR (human relations) actually cost us more time and attention and did not save us money. But it did help us cast a broader net when looking for candidates. We had hundreds of applications from around the globe for a junior developer role, including applicants located in Ukraine, Africa, and South America. NLP helped us quickly evaluate English and technical skill before proceeding with interviews and paid take-home assignments.

And the spam filters have retained their edge in the cat and mouse game between spam filters and spam generators for email, but may be losing in other environments like social networks. An estimated 20% of the tweets about the 2016 US presidential election were composed by chatbots. [\[29\]](#) These bots amplify their owners' and developers' viewpoints with the resources and motivation to influence popular opinion. And these "puppet masters" tend to be foreign governments or large corporations.

NLP systems can generate more than just short social network posts. NLP can be used to compose lengthy movie and product reviews on online shop websites and elsewhere. Many reviews are the creation of autonomous NLP pipelines that have never set foot in a movie theater or purchased the product they are reviewing. And it's not just movies, a large portion of all product reviews that bubble to the top in search engines and online retailers are fake. You can use NLP to help search engines and prosocial social media communities (Mastodon) [\[30\]](#) detect and remove misleading or fake posts and reviews. [\[31\]](#)

There are chatbots on Slack, IRC, and even customer service websites—places where chatbots have to deal with ambiguous commands or questions. And chatbots paired with voice recognition and generation systems can even handle lengthy conversations with an indefinite goal or "objective function" such as making a reservation at a local restaurant. [\[32\]](#) NLP systems can answer phones for companies that want something better than a phone tree, but they do not want to pay humans to help their customers.



Warning

Consider the ethical implications whenever you, or your boss, decide to deceive your users. With its **Duplex** demonstration at Google IO, engineers and managers overlooked concerns about the ethics of teaching chatbots to deceive humans. In most "entertainment" social networks, bots are not required to reveal themselves. We unknowingly interact with these bots on Facebook, Reddit, Twitter and even dating apps. Now that bots and deep fakes can so convincingly deceive us, the AI control problem [\[33\]](#). Yuval Harari's cautionary forecast of "Homo Deus" [\[34\]](#) may come sooner than we think.

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

The most surprising and powerful application of NLP is in psychology. If you think that a chatbot could never replace your therapist, you may be surprised by recent advancements. [\[35\]](#) Commercial virtual companions such as Xiaoice in China and Replika.AI in the US helped hundreds of millions of lonely people survive the emotional impact of social isolation during Covid-19 lockdowns in 2020 and 2021. [\[36\]](#) Fortunately, you don't have to rely on engineers at large corporations to look out for your best interests. Many psychotherapy and cognitive assistant technology is completely free and open source. [\[37\]](#)

1.3.1 Processing programming languages with NLP

Modern deep-learning NLP pipelines have proven so powerful and versatile that they can now accurately understand and generate programming languages. Rule-based compilers and generators for NLP were helpful for

simple tasks like autocomplete, and providing snippet suggestions. And users can often use information retrieval systems, or search engines, to find snippets of code to complete their software development project.

And these tools just got a whole lot smarter. Previous code generation tools were **extractive**. Extractive text generation algorithms find the most relevant text in your history and just regurgitate it, verbatim as a suggestion to you. So if the term "prosocial artificial intelligence" appears a lot in the text an algorithm was trained on, an auto-complete will recommend the word "artificial intelligence" to follow prosocial rather than just "intelligence". You can see how this might start to influence what you type and how you think.

And transformers have advanced NLP even further recently with massive deep learning networks that are more **abstractive**, generating new text you haven't seen or typed before. For example the 175 billion parameter version of GPT-3 was trained on all of GitHub to create a model called Codex. Codex is part of the Copilot plugin for VSCode. It suggests entire function and class definitions and all you have to supply is supply a short comment and the first line of the function definition. Here is the example for typescript shown on the copilot home page: [\[38\]](#)

```
// Determine whether the sentiment of text is positive
// Use a web service
async function isPositive(text: string): Promise<boolean> {
```

In the demo animation, Copilot then generated the rest of the typescript required for a working function that estimated the sentiment of a body of text. Think about that for a second. Microsoft's algorithm is writing code for you to analyze the sentiment of natural language text, such as the text you might be writing up in your emails or personal essay. And the examples shown on the Copilot home page all lean towards Microsoft products and services. This means you will end up with an NLP pipeline that has **Microsoft's** perspective on what is positive and what is not. It values what **Microsoft** told it to value. Just as Google Search influenced the kind of code you wrote indirectly, now Microsoft algorithms are directly writing code for you.

Since you're reading this book, you are probably planning to build some pretty cool NLP pipelines. You may even build a pipeline that helps you

write blog posts and chatbots and core NLP algorithms. This can create a positive feedback loop that shifts the kinds of NLP pipelines and models that are built and deployed by engineers like you. So pay attention to the **meta** tools that you use to help you code and think. These have huge leverage on the direction of your code, and the direction of your life.

[26] "AP's 'robot journalists' are writing their own stories now", The Verge, Jan 29, 2015, <http://www.theverge.com/2015/1/29/7939067/ap-journalism-automation-robots-financial-reporting>

[27] Many chatbot frameworks, such as qary (<http://gitlab.com/tangibleai.com/qary>) allow importing of legacy FAQ lists to automatically compose a rule-based dialog engine for your chatbot.

[28] "Are Sample-Efficient NLP Models More Robust?" by Nelson F. Liu, Ananya Kumar, Percy Liang, Robin Jia (<https://arxiv.org/abs/2210.06456>)

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

[30] "A beginners guide to Mastodon" on Tech Crunch (<https://techcrunch.com/2022/11/08/what-is-mastodon/>) by Amanda Silberling on Mastodon (<https://mstdn.social/@amanda@journa.host>)

[31] 2021, E.Madhorubagan et al "Intelligent Interface for Fake Product Review Monitoring and Removal"
(https://ijirt.org/master/publishedpaper/IJIRT151055_PAPER.pdf)

[32] Google Blog May 2018 about their *Duplex* system
<https://ai.googleblog.com/2018/05/advances-in-semantic-textual-similarity.html>

[33] Wikipedia is probably your most objective reference on the "AI control

problem" (https://en.wikipedia.org/wiki/AI_control_problem).

[34] WSJ Blog, March 10, 2017 <https://blogs.wsj.com/cio/2017/03/10/homo-deus-author-yuval-noah-harari-says-authority-shifting-from-people-to-ai/>

[35] John Michael Innes and Ben W. Morrison at the University of South Australia "Machines can do most of a psychologist's job", 2021, (<https://theconversation.com/machines-can-do-most-of-a-psychologists-job-the-industry-must-prepare-for-disruption-154064>)

[36] C.S. Voll "Humans Bonding with Virtual Companions"
(<https://medium.com/predict/humans-bonding-with-virtual-companions-6d19beae0077>)

[37] Tangible AI builds open source cognitive assistants that help users take control of their emotions such as Syndee (<https://syndee.ai>) and qary (<https://gitlab.com/tangibleai/qary>) Some of Replika.AI's core technologies are open source (<https://github.com/lukalabs>)

[38] Taken from animation on copilot.github.com (<https://copilot.github.com/>)

1.4 Language through a computer's "eyes"

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

What if you decided to search a memory bank (database) for the exact same string of bits, characters, or words, and use one of the responses that other

humans and authors have used for that statement in the past? But imagine if there was a typo or variation in the statement. Our bot would be sent off the rails. And bits aren't continuous or forgiving—they either match or they do not. There is no obvious way to find similarity between two streams of bits that takes into account what they signify. The bits for "good" will be just as similar to "bad!" as they are to "okay".

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

1.4.1 The language of locks

Surprisingly the humble combination lock is actually a simple language processing machine. So, if you are mechanically inclined, this section may be illuminating. But if you do not need mechanical analogies to help you understand algorithms and how regular expressions work, then you can skip this section.

After finishing this section, you will never think of your combination bicycle lock the same way again. A combination lock certainly can't read and understand the textbooks stored inside a school locker, but it can understand the language of locks. It can understand when you try to "tell" it a "password": a combination. A padlock combination is any sequence of symbols that matches the "grammar" (pattern) of lock language. Even more importantly, the padlock can tell if a lock "statement" matches a particularly meaningful statement, the one for which there is only one correct "response," to release the catch holding the U-shaped hasp so you can get into your locker.

This lock language (regular expressions) is a particularly simple one. But it's not so simple that we can't use it in a chatbot. We can use it to recognize a key phrase or command to unlock a particular action or behavior.

For example, we'd like our chatbot to recognize greetings such as "Hello Rosa," and respond to them appropriately. This kind of language, like the language of locks, is a formal language because it has strict rules about how

an acceptable statement must be composed and interpreted. If you've ever written a math equation or coded a programming language expression, you've written a formal language statement.

Formal languages are a subset of natural languages. Many natural language statements can be matched or generated using a formal language grammar, such as regular expressions or regular grammars. That's the reason for this diversion into the mechanical, "click, whirr"^[39] language of locks.

1.4.2 Regular expressions

Regular expressions use a special class of formal language grammars called a regular grammar. Regular grammars have predictable, provable behavior, and yet are flexible enough to power some of the most sophisticated dialog engines and chatbots on the market. Amazon Alexa and Google Now are mostly pattern-based engines that rely on regular grammars. Deep, complex regular grammar rules can often be expressed in a single line of code called a regular expression. There are successful chatbot frameworks in Python, like `will`, ^[40] and `qary` ^[41] that rely exclusively on this kind of language processing to produce some effective chatbots.



Note

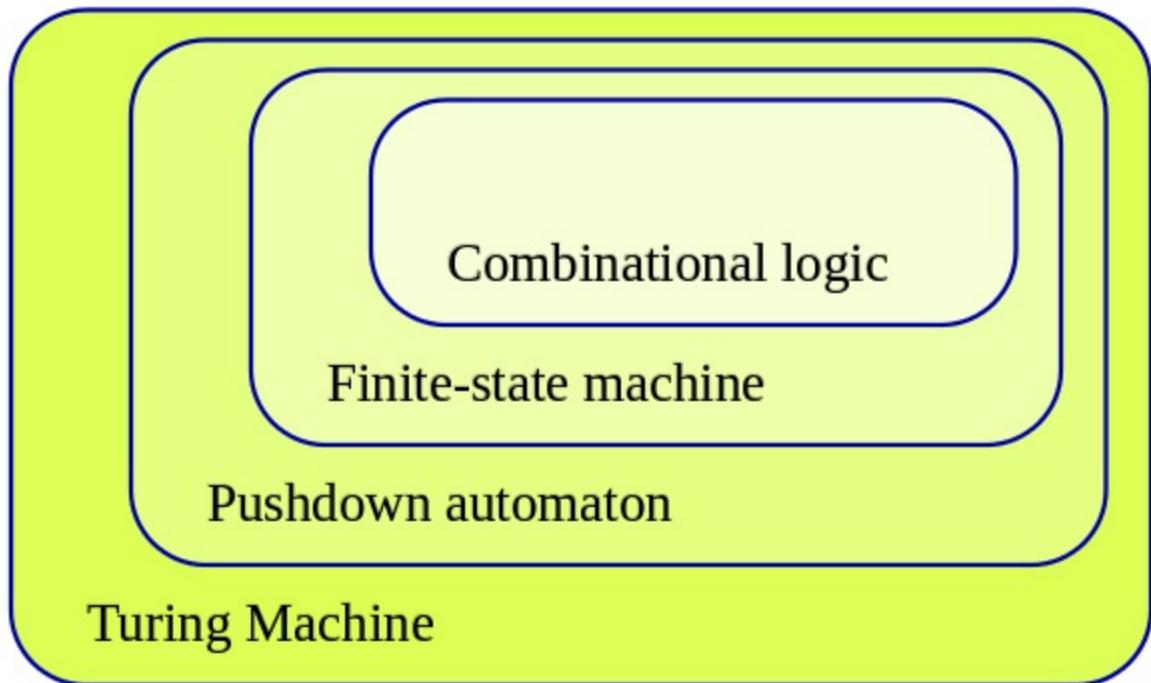
Regular expressions implemented in Python and in Posix (Unix) applications such as grep are not true regular grammars. They have language and logic features such as look-ahead and look-back that make leaps of logic and recursion that aren't allowed in a regular grammar. As a result, regular expressions aren't provably halting; they can sometimes "crash" or run forever. ^[42]

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

then trigger a scripted response appropriate to that particular regular expression match. And that same regular expression can also be used to extract a useful piece of information from a statement. A chatbot can add that bit of information to its knowledge base about the user or about the world the user is describing.

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

Figure 1.4. Kinds of automata



Formal mathematical explanation of formal languages

Kyle Gorman describes programming languages this way:

- Most (if not all) programming languages are drawn from the class of context-free languages.
- Context free languages are parsed with context-free grammars, which provide efficient parsing.
- The regular languages are also efficiently parsable and used extensively in computing for string matching.
- String matching applications rarely require the expressiveness of context-free.
- There are a number of formal language classes, a few of which are shown here (in decreasing complexity):^[43]
 - Recursively enumerable
 - Context-sensitive
 - Context-free
 - Regular

Natural languages are:

- Not regular ^[44]
- Not context-free ^[45]
- Can't be defined by any formal grammar ^[46]

^[39] One of Cialdini's six psychology principles in his popular book *Influence*
<http://changingminds.org/techniques/general/cialdini/click-whirr.htm>

^[40] Steven Skoczen's Will chatbot framework
<https://github.com/skoczen/will>)

^[41] Tangible AI's chatbot framework called qary (<https://docs.qary.ai>) with examples deployed for WeSpeakNYC
<https://wespeaknyc.cityofnewyork.us/>) and others

^[42] Stack Exchange Went Down for 30 minutes on July 20, 2016 when a regex "crashed" (<http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016>)

[43] See the web page titled "Chomsky hierarchy - Wikipedia"
(https://en.wikipedia.org/wiki/Chomsky_hierarchy).

[44] "English is not a regular language"
(<http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=20>) by
Shuly Wintner

[45] "Is English context-free?"
(<http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=24>) by
Shuly Wintner

[46] See the web page titled "1.11. Formal and Natural Languages — How to Think like a Computer Scientist: Interactive Edition"
(<https://runestone.academy/ns/books/published/fopp/GeneralIntro/FormalandNaturallanguages.html>)

1.5 A simple chatbot

Let us build a quick and dirty chatbot. It will not be very capable, and it will require a lot of thinking about the English language. You will also have to hardcode regular expressions to match the ways people may try to say something. But do not worry if you think you couldn't have come up with this Python code yourself. You will not have to try to think of all the different ways people can say something, like we did in this example. You will not even have to write regular expressions (regexes) to build an awesome chatbot. We show you how to build a chatbot of your own in later chapters without hardcoded anything. A modern chatbot can learn from reading (processing) a bunch of English text. And we show you how to do that in later chapters.

This pattern matching chatbot is an example of a tightly controlled chatbot. Pattern matching chatbots were common before modern machine learning chatbot techniques were developed. And a variation of the pattern matching approach we show you here is used in chatbots like Amazon Alexa and other virtual assistants.

For now let's build a FSM, a regular expression, that can speak lock language (regular language). We could program it to understand lock language

statements, such as "01-02-03." Even better, we'd like it to understand greetings, things like "open sesame" or "hello Rosa."

An important feature for a prosocial chatbot is to be able to respond to a greeting. In high school, teachers often chastised me for being impolite when I'd ignore greetings like this while rushing to class. We surely do not want that for our benevolent chatbot.

For communication between two machines, you would define a handshake with something like an ACK (acknowledgement) signal to confirm receipt of each message. But our machines are going to be interacting with humans who say things like "Good morning, Rosa". We do not want it sending out a bunch of chirps, beeps, or ACK messages, like it's syncing up a modem or HTTP connection at the start of a conversation or web browsing session.

Human greetings and handshakes are a little more informal and flexible. So recognizing the greeting *intent* won't be as simple as building a machine handshake. So you will want a few different approaches in your toolbox.



Note

An intent is a category of possible intentions the user has for the NLP system or chatbot. Words "hello" and "hi" might be collected together under the *greeting* intent, for when the user wants to start a conversation. Another intent might be to carry out some task or command, such as a "translate" command or the query "How do I say 'Hello' in Ukrainian?". You'll learn about intent recognition throughout the book and put it to use in a chatbot in chapter 12.

1.6 Keyword-based greeting recognizer

Your first chatbot will be straight out of the 80's. Imagine you want a chatbot to help you select a game to play, like chess... or a Thermonuclear War. But first your chatbot must find out if you are professor Falken or General Beringer, or some other user that knows what they are doing.^[47] It will only be able to recognize a few greetings. But this approach can be extended to

help you implement simple keyword-based intent recognizers on projects similar to those mentioned earlier in this chapter.

Listing 1.1. Keyword detection using `str.split`

```
>>> greetings = "Hi Hello Greetings".split()
>>> user_statement = "Hello Joshua"
>>> user_token_sequence = user_statement.split()
>>> user_token_sequence
['Hello', 'Joshua']
>>> if user_token_sequence[0] in greetings:
...     bot_reply = "The mononuclear War is a strange game. " #1
...     bot_reply += "The only winning move is NOT TO PLAY."
>>> else:
...     bot_reply = "Would you like to play a nice game of chess?"
```

This simple NLP pipeline (program) has only two intent categories: "greeting" and "unknown" (else). And it uses a very simple algorithm called keyword detection. Chatbots that recognize the user's intent like this have capabilities similar to modern command line applications or phone trees from the 90's.

Rule-based chatbots can be much much more fun and flexible than this simple program. Developers have so much fun building and interacting with chatbots that they build chatbots to make even deploying and monitoring servers a lot of fun. *Chatops*, or devops with chatbots, has become popular on most software development teams. You can build a chatbot like this to recognize more intents by adding `elif` statements before the `else`. Or you can go beyond keyword-based NLP and start thinking about ways to improve it using regular expressions.

1.6.1 Pattern-based intent recognition

A keyword based chatbot would recognize "Hi", "Hello", and "Greetings", but it wouldn't recognize "Hiiii" or "Hiiiiiiiiii" - the more excited renditions of "Hi". Perhaps you could hardcode the first 200 versions of "Hi", such as `["Hi", "Hii", "Hiii", ...]`. Or you could programmatically create such a list of keywords. Or you could save yourself a lot of trouble and make your bot deal with literally infinite variations of "Hi" using *regular expressions*.

Regular expression *patterns* can match text much more robustly than any hard-coded rules or lists of keywords.

Regular expressions recognize patterns for any sequence of characters or symbols.^[48] With keyword based NLP, you and your users need to spell keywords and commands exactly the same way for the machine to respond correctly. So your keyword greeting recognizer would miss greetings like "Hey" or even "hi" because those strings aren't in your list of greeting words. And what if your "user" used a greeting that starts or ends with punctuation, such as "sup" or "Hi,". You could do *case folding* with the `str.split()` method on both your greetings and the user statement. And you could add more greetings to your list of greeting words. You could even add misspellings and typos to ensure they aren't missed. But that is a lot of manual "hard-coding" of data into your NLP pipeline.

You will soon learn how to use machine learning for more data-driven and automated NLP pipelines. And when you graduate to the much more complex and accurate *deep learning* models of chapter 7 and beyond, you will find that there is still much "brittleness" in modern NLP pipelines. Robin Jia's thesis explains how to measure and improve NLP robustness in his thesis (<https://proai.org/robinjia-thesis>)] But for now, you need to understand the basics. When your user wants to specify actions with precise patterns of characters similar to programming language commands, that's where regular expressions shine.

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

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

characters if they are all within that character class.

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

```
>>> r = r"[^a-z]*([y]o|[h']?ello|ok|hey|(good[ ])|(morn[gin']){0,3}"+
>>> r += r"afternoon|even[gin']{0,3})][\s,;:]{1,3}([a-z]{1,20})")"
>>> re_greeting = re.compile(r, flags=re.IGNORECASE) #1
>>> re_greeting.match('Hello Rosa')
<_sre.SRE_Match object; span=(0, 10), match='Hello Rosa'>
>>> re_greeting.match('Hello Rosa').groups()
('Hello', None, None, 'Rosa')
>>> re_greeting.match("Good morning Rosa")
<_sre.SRE_Match object; span=(0, 17), match="Good morning Rosa">
>>> re_greeting.match("Good Manning Rosa") #2
>>> re_greeting.match('Good evening Rosa Parks').groups() #3
('Good evening', 'Good ', 'evening', 'Rosa')
>>> re_greeting.match("Good Morn'n Rosa")
<_sre.SRE_Match object; span=(0, 16), match="Good Morn'n Rosa">
>>> re_greeting.match("yo Rosa")
<_sre.SRE_Match object; span=(0, 7), match='yo Rosa'>
```



Tip

The "r" before the quote symbol (r') indicates that the quoted string literal is a **raw** string. The "r" does not mean **regular** expression. A Python raw string just makes it easier to use the backslashes used to escape special symbols within a regular expression. Telling Python that a str is "raw" means that Python will skip processing the backslashes and pass them on to the regular expression parser (re package). Otherwise you would have to escape each and every backslash in your regular expression with a double-backslash ('\\'). So the whitespace matching symbol '\s' would become '\\s', and special characters like literal curly braces would become '\\{' and '\\}'.

There is a lot of logic packed into that first line of code, the regular expression. It gets the job done for a surprising range of greetings. But it missed that "Manning" typo, which is one of the reasons NLP is hard. In machine learning and medical diagnostic testing, that's called a *false negative* classification error. Unfortunately, it will also match some statements that humans would be unlikely to ever say—a *false positive*, which is also a bad

thing. Having both false positive and false negative errors means that our regular expression is both too liberal (inclusive) and too strict (exclusive). These mistakes could make our bot sound a bit dull and mechanical. We'd have to do a lot more work to refine the phrases it matches for the bot to behave in a more intelligent human-like way.

And this tedious work would be highly unlikely to ever succeed at capturing all the slang and misspellings people use. Fortunately, composing regular expressions by hand isn't the only way to train a chatbot. Stay tuned for more on that later (the entire rest of the book). So we only use them when we need precise control over a chatbot's behavior, such as when issuing commands to a voice assistant on your mobile phone.

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

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

So if you run this little script and chat to our bot with a phrase like "Hello Rosa", it will respond by asking about your day. If you use a slightly rude name to address the chatbot, she will be less responsive, but not inflammatory, to encourage politeness.^[49] 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 is no one else out there watching our `input()` line, but if this were a function within a larger chatbot, you want to deal with these sorts of things.

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



Theory of a computational mind

This classical NLP pattern-matching approach is based on the computational theory of mind (CTM). CTM theorizes that thinking is a deterministic computational process that acts in a single logical thread or sequence.^[50] Advancements in neuroscience and NLP led to the development of a "connectionist" theory of mind around the turn of the century. This newer theory inspired the artificial neural networks of deep learning used that process natural language sequences many different ways simultaneously, in parallel.^{[51] [52]}

In chapter 2 you will learn more about pattern-based approaches to tokenizing—splitting text into tokens or words with algorithms such as the "Treebank tokenizer." You will also learn how to use pattern matching to stem (shorten and consolidate) tokens with something called a Porter stemmer. But in later chapters we take advantage of the exponentially greater computational resources, as well as our larger datasets, to shortcut this laborious hand programming and refining.

If you are new to regular expressions and want to learn more, you can check out appendix B or the online documentation for Python regular expressions. But you do not have to understand them just yet. We'll continue to provide you with example regular expressions as we use them for the building blocks

of our NLP pipeline. So, do not worry if they look like gibberish. Human brains are pretty good at generalizing from a set of examples, and I'm sure it will become clear by the end of this book. And it turns out machines can learn this way as well...

1.6.2 Another way

Imagine a giant database containing sessions of dialog between humans. You might have statements paired with responses from thousands or even millions of conversations. One way to build a chatbot would be to search such a database for the exact same string of characters the user just "said" to your chatbot. And then you could use one of the responses to that statement that other humans have said in the past. That would result in a statistical or data-driven approach to chatbot design. And that could take the place of all that tedious pattern matching algorithm design.

Think about how a single typo or variation in the statement would trip up pattern-matching bot or even a data-driven both with millions of statements (utterances) in its database. Bit and character sequences are discrete and very precise. They either match or they do not. And people are creative. It may not seem like it sometimes, but very often people say something with new patterns of characters never ever seen before. So you'd like your bot to be able to measure the difference in **meaning** between character sequences. In later chapters you'll get better and better at extracting *meaning* from text!

When we use character sequence matches to measure distance between natural language phrases, we'll often get it wrong. Phrases with similar meaning, like "good" and "okay", can often have different character sequences and large distances when we count up character-by-character matches to measure distance. And sometimes two words look almost the same but mean completely different things: "bad" and "bag." You can count the number of characters that change from one word to another with algorithms such as Jaccard and Levenshtein algorithms. But these distance or "change" counts fail to capture the essence of the relationship between two dissimilar strings of characters such as "good" and "okay.".= And they fail to account for how small spelling differences might not really be typos but rather completely different words, such as "bad" and "bag".

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

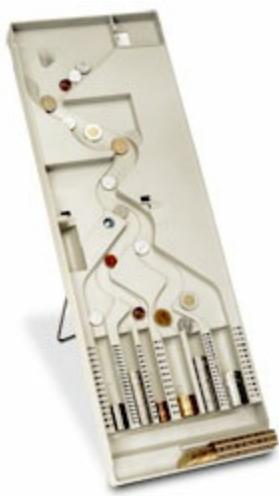
We do not stay in this confusing binary world of logic for long, but let's imagine we're famous World War II-era code-breaker Mavis Batey at Bletchley Park and we have just been handed that binary, Morse code message intercepted from communication between two German military officers. It could hold the key to winning the war. Where would we start? Well the first layer of deciding would be to do something statistical with that stream of bits to see if we can find patterns. We can first use the Morse code table (or ASCII table, in our case) to assign letters to each group of bits. Then, if the characters are gibberish to us, as they are to a computer or a cryptographer in WWII, we could start counting them up, looking up the short sequences in a dictionary of all the words we have 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 have read before. This collection of documents is called a *corpus*, and the words or sequences we have listed in our index are called a *lexicon*.

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

we show you how to teach a machine our language using Word2Vec in later chapters, it may seem magical, but it's not. It's just math, computation.

But let's think for a moment about what information has been lost in our effort to count all the words in the messages we receive. We assign the words to bins and store them away as bit vectors like a coin or token sorter (see figure 1.2) directing different kinds of tokens to one side or the other in a cascade of decisions that piles them in bins at the bottom. Our sorting machine must take into account hundreds of thousands if not millions of possible token "denominations," one for each possible word that a speaker or author might use. Each phrase or sentence or document we feed into our token sorting machine will come out the bottom, where we have a "vector" with a count of the tokens in each slot. Most of our counts are zero, even for large documents with verbose vocabulary. But we have not lost any words yet. What have we lost? Could you, as a human understand a document that we presented you in this way, as a count of each possible word in your language, without any sequence or order associated with those words? I doubt it. But if it was a short sentence or tweet, you'd probably be able to rearrange them into their intended order and meaning most of the time.

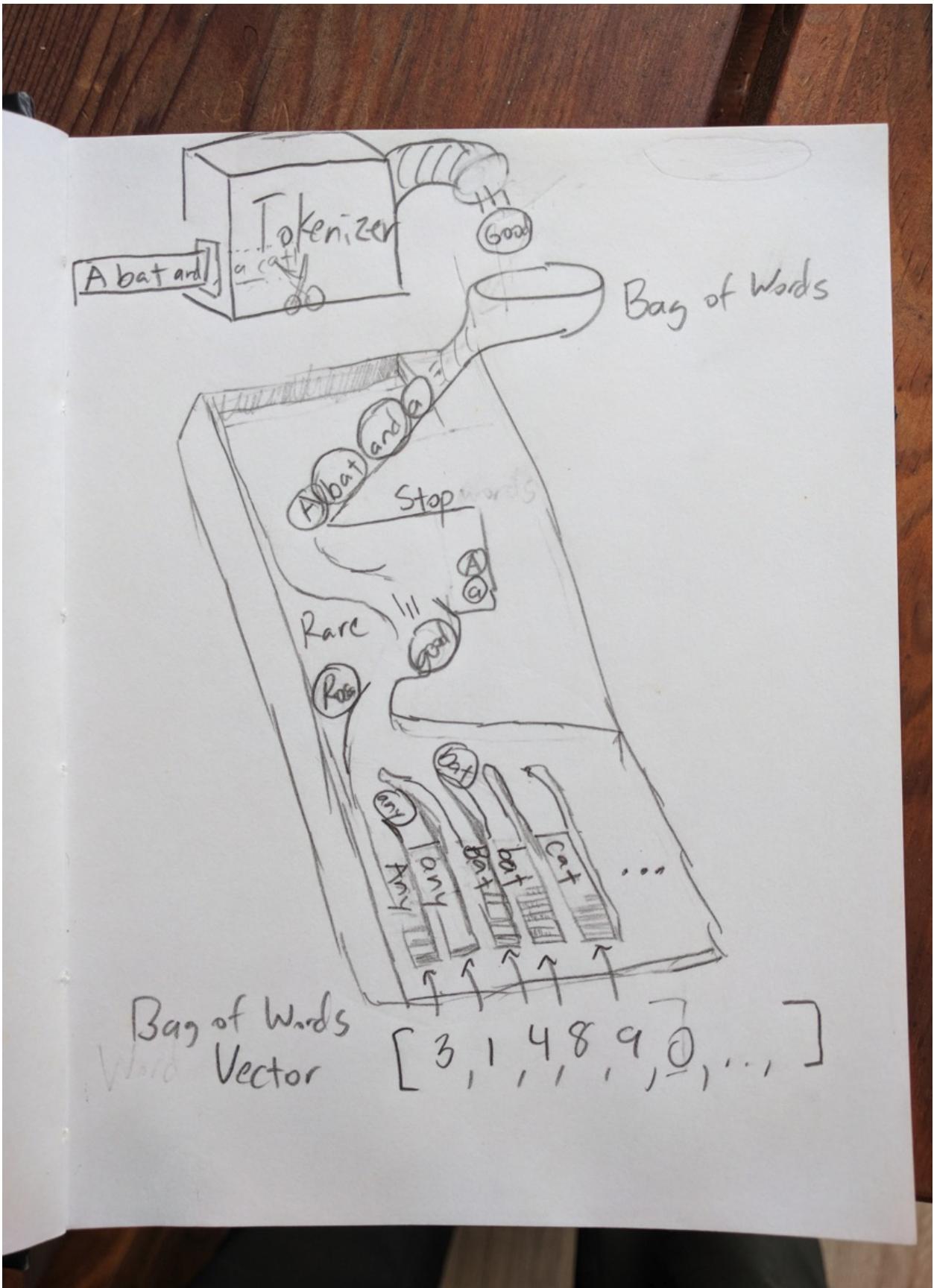
Figure 1.5. Canadian coin sorter



Here's how our token sorter fits into an NLP pipeline right after a tokenizer (see chapter 2). We have 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 1.6. Token sorting tray



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

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

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

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

And we can imagine feeding into this machine, one at a time, all the documents, statements, sentences, and even single words we could find. We'd count up the tokens in each slot at the bottom after each of these statements was processed, and we'd call that a vector representation of that

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

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

[47] The code here simplifies the behavior of the chatbot called "Joshua" in the "War Games" movie. See wikiquote (<https://en.wikiquote.org/wiki/WarGames>) for more chatbot script ideas.

[48] SpaCy 'Matcher' (<https://spacy.io/api/matcher>) is a regular expression interpreter for patterns of words, parts of speech, and other symbol sequences.

[49] The idea for this defusing response originated with Viktor Frankl's *Man's Search for Meaning*, his Logotherapy (<https://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.

[50] Stanford Encyclopedia of Philosophy, Computational Theory of Mind,

<https://plato.stanford.edu/entries/computational-mind/>

[51] Stanford Encyclopedia of Philosophy, Connectionism,
<https://plato.stanford.edu/entries/connectionism/>

[52] Christiansen and Chater, 1999, Southern Illinois University
(<https://crl.ucsd.edu/~elman/Bulgaria/christiansen-chater-soa.pdf>)

1.7 A brief overflight of hyperspace

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

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

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

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

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

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

There is 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. The vector for each character would contain the answer to binary (yes/no) questions about every letter and punctuation mark in your alphabet:

"Is the first letter an 'A'?" "Is the first letter an 'B'?" ... "Is the first letter an 'z'?"

And the next vector would answer the same boring questions about the next letter in the sequence.

"Is the second letter an A?" "Is the second letter an B?" ...

Despite all the "no" answers or zeroes in this vector sequence, it does have one advantage over all other possible representations of text - it retains every tiny detail, every bit of information contained in the original text, including the order of the characters and words. This like the paper representation of a song for a player piano that only plays a single note at a time. The "notes" for this natural language mechanical player piano are the 26 uppercase and lowercase letters plus any punctuation that the piano must know how to "play." The paper roll wouldn't have to be much wider than for a real player piano and the number of notes in some long piano songs doesn't exceed the number of characters in a small document.

But this one-hot character sequence encoding representation is mainly useful for recording and then replaying an exact piece rather than composing something new or extracting the essence of a piece. We can't easily compare the piano paper roll for one song to that of another. And this representation is longer than the original ASCII-encoded representation of the document. The number of possible document representations just exploded in order to retain information about each sequence of characters. We retained the order of characters and words but expanded the dimensionality of our NLP problem.

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

1.8 Word order and grammar

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

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

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

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

```
>>> s = """Find textbooks with titles containing 'NLP',  
...      or 'natural' and 'language', or  
...      'computational' and 'linguistics'."""  
>>> len(set(s.split()))  
12  
>>> import numpy as np  
>>> np.arange(1, 12 + 1).prod() # factorial(12) = arange(1, 13).  
479001600
```

The number of permutations exploded from `factorial(3) == 6` in our simple greeting to `factorial(12) == 479001600` in our longer statement! And it's clear that the logic contained in the order of the words is important to any machine that would like to reply with the correct response. Even though common greetings are not usually garbled by bag-of-words processing, more complex statements can lose most of their meaning when thrown into a bag. A bag of words is not the best way to begin processing a database query, like the natural language query in the preceding example.

Whether a statement is written in a formal programming language like SQL, or in an informal natural language like English, word order and grammar are important when a statement intends to convey logical relationships between things. That's why computer languages depend on rigid grammar and syntax rule parsers. Fortunately, recent advances in natural language syntax tree parsers have made possible the extraction of syntactical and logical relationships from natural language with remarkable accuracy (greater than 90%).^[53] In later chapters, we show you how to use packages like SyntaxNet (Parsey McParseface) and Spacy to identify these relationships.

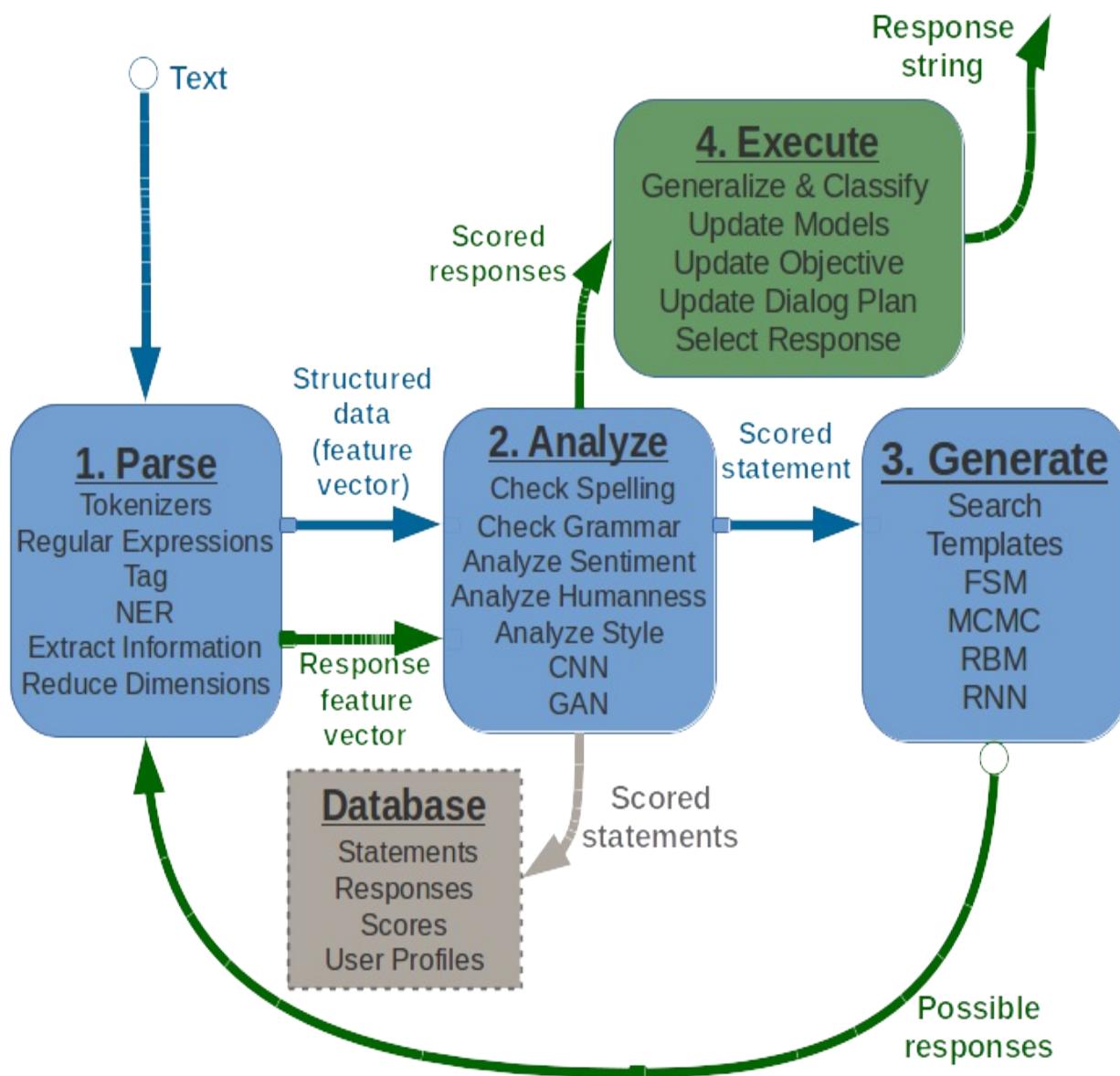
And just as in the Bletchley Park example greeting, even if a statement doesn't rely on word order for logical interpretation, sometimes paying attention to that word order can reveal subtle hints of meaning that might facilitate deeper responses. These deeper layers of natural language processing are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

[53] A comparison of the syntax parsing accuracy of SpaCy (93%), SyntaxNet (94%), Stanford's CoreNLP (90%), and others is available at <https://spacy.io/docs/api/>

1.9 A chatbot natural language pipeline

The NLP pipeline required to build a dialog engine, or chatbot, is similar to the pipeline required to build a question answering system described in *Taming Text* (Manning, 2013).^[54] However, some of the algorithms listed within the five subsystem blocks may be new to you. We help you implement these in Python to accomplish various NLP tasks essential for most applications, including chatbots.

Figure 1.7. Chatbot recirculating (recurrent) pipeline



A chatbot requires four kinds of processing as well as a database to maintain

a memory of past statements and responses. Each of the four processing stages can contain one or more processing algorithms working in parallel or in series (see figure 1.4).

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

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

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

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

The NLP pipeline for a chatbot shown in figure 1.4 contains all the building blocks for most of the NLP applications that we described at the start of this chapter. As in *Taming Text*, we break out our pipeline into four main

subsystems or stages. In addition we have 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 have shown a "feedback loop" on our generated text responses so that our responses can be processed using the same algorithms used to process the user statements. The response "scores" or features can then be combined in an objective function to evaluate and select the best possible response, depending on the chatbot's plan or goals for the dialog. This book is focused on configuring this NLP pipeline for a chatbot, but you may also be able to see the analogy to the NLP problem of text retrieval or "search," perhaps the most common NLP application. And our chatbot pipeline is certainly appropriate for the question answering application that was the focus of *Taming Text*.

The application of this pipeline to financial forecasting or business analytics may not be so obvious. But imagine the features generated by the analysis portion of your pipeline. These features of your analysis or feature generation can be optimized for your particular finance or business prediction. That way they can help you incorporate natural language data into a machine learning pipeline for forecasting. Despite focusing on building a chatbot, this book gives you the tools you need for a broad range of NLP applications, from search to financial forecasting.

One processing element in figure 1.4 that is not typically employed in search, forecasting, or question answering systems is natural language *generation*. For chatbots this is their central feature. Nonetheless, the text generation step is often incorporated into a search engine NLP application and can give such an engine a large competitive advantage. The ability to consolidate or summarize search results is a winning feature for many popular search engines (DuckDuckGo, Bing, and Google). And you can imagine how valuable it is for a financial forecasting engine to be able to generate statements, tweets, or entire articles based on the business-actionable events it detects in natural language streams from social media networks and news feeds.

The next section shows how the layers of such a system can be combined to

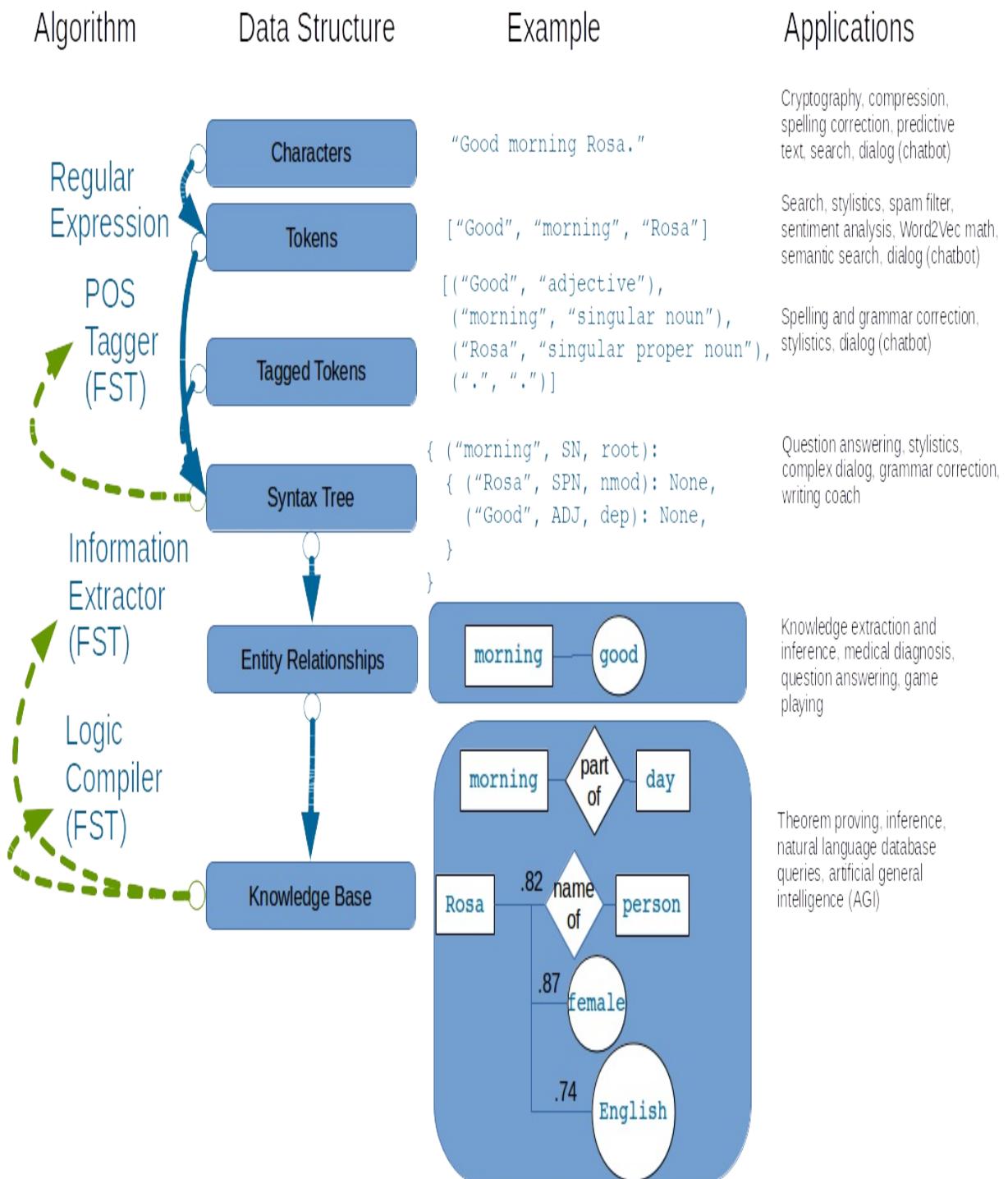
create greater sophistication and capability at each stage of the NLP pipeline.

[54] Ingwersol, Morton, and Farris, http://www.manning.com/books/taming-text/?a_aid=totalgood

1.10 Processing in depth

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

Figure 1.8. Example layers for an NLP pipeline



The top four layers in figure 1.8 correspond to the first two stages in the chatbot pipeline (feature extraction and feature analysis) in the previous section. For example the part-of-speech tagging (POS tagging), is one way to generate features within the Analyze stage of our chatbot pipeline. POS tags

are generated automatically by the default Spacy pipeline, which includes all the top four layers in this diagram. POS tagging is typically accomplished with a finite state transducer like the methods in the `nltk.tag` package.

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

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

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

mechanical robot built from salvaged parts by Gunther Cox.^[55]

Will is an open source Python chatbot framework by Steven Skoczen with a completely different approach.^[56] 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" footnote:[The Python regex package is backward compatible with re and adds fuzziness among other features. The regex will replace the re package in future python versions (<https://pypi.python.org/pypi/regex>)].

Similarly TRE agrep, or "approximate grep," (<https://github.com/laurikari/tre>) is an alternative to the UNIX command-line application grep.] 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 pattern-matching chatbots requires a lot of difficult 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.

A lot of powerful things can be done with shallow NLP. And little, if any, human supervision (labeling or curating of text) is required. Often a machine can be left to learn perpetually from its environment (the stream of words it can pull from Twitter or some other source).^[57] We show you how to do this in chapter 6.

[55] ChatterBot by Gunther Cox and others at
<https://github.com/gunthercox/ChatterBot>

[56] See the GitHub page for "Will," a chatbot for HipChat, by Steven Skoczen and the HipChat community (<https://github.com/skoczen/will>). In

2018 it was updated to integrate with Slack

[57] Simple neural networks are often used for unsupervised feature extraction from character and word sequences.

1.11 Natural language IQ

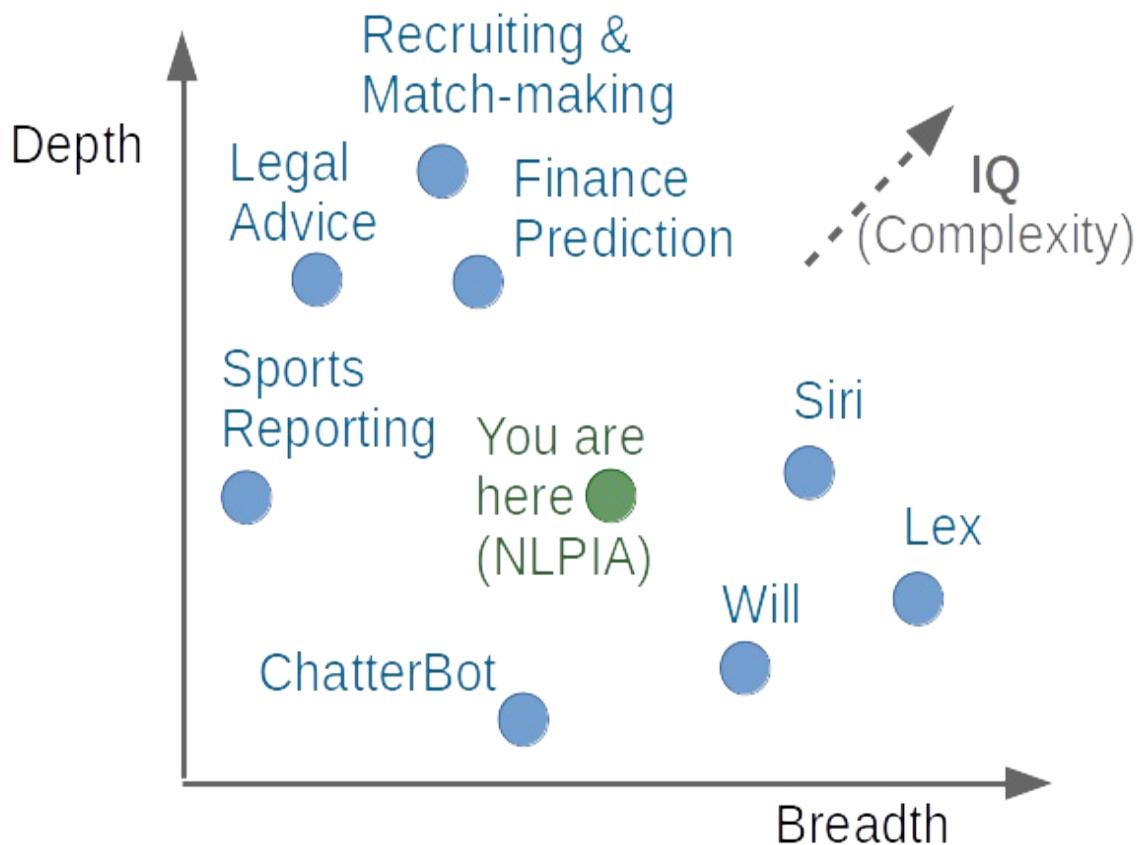
Like human brainpower, the power of an NLP pipeline cannot be easily gauged with a single IQ score without considering multiple "smarts" dimensions. A common way to measure the capability of a robotic system is along the dimensions of complexity of behavior and degree of human supervision required. But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

A consumer product chatbot or virtual assistant like Alexa or Allo is usually designed to have extremely broad knowledge and capabilities. However, the logic used to respond to requests tends to be shallow, often consisting of a set of trigger phrases that all produce the same response with a single if-then decision branch. Alexa (and the underlying Lex engine) behave like a single layer, flat tree of (if, elif, elif, ...) statements.^[58] Google Dialogflow (which was developed independently of Google's Allo and Google Assistant) has similar capability to Amazon Lex, Contact Flow, and Lambda, but without the drag-and-drop user interface for designing your dialog tree.

On the other hand, the Google Translate pipeline (or any similar machine translation system) relies on a deep tree of feature extractors, decision trees, and knowledge graphs connecting bits of knowledge about the world. Sometimes these feature extractors, decision trees, and knowledge graphs are explicitly programmed into the system, as in figure 1.5. Another approach rapidly overtaking this "hand-coded" pipeline is the deep learning data-driven approach. Feature extractors for deep neural networks are learned rather than hard-coded, but they often require much more training data to achieve the same performance as intentionally designed algorithms.

You will use both approaches (neural networks and hand-coded algorithms) as you incrementally build an NLP pipeline for a chatbot capable of conversing within a focused knowledge domain. This will give you the skills you need to accomplish the natural language processing tasks within your industry or business domain. Along the way you will probably get ideas about how to expand the breadth of things this NLP pipeline can do. Figure 1.6 puts the chatbot in its place among the natural language processing systems that are already out there. Imagine the chatbots you have interacted with. Where do you think they might fit on a plot like this? Have you attempted to gauge their intelligence by probing them with difficult questions or something like an IQ test?^[59] You will get a chance to do exactly that in later chapters, to help you decide how your chatbot stacks up against some of the others in this diagram.

Figure 1.9. 2D IQ of some natural language processing systems



As you progress through this book, you will be building the elements of a

chatbot. Chatbots require all the tools of NLP to work well:

- Feature extraction (usually to produce a vector space model)
- Information extraction to be able to answer factual questions
- Semantic search to learn from previously recorded natural language text or dialog
- Natural language generation to compose new, meaningful statements

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

And machine learning NLP pipelines are easier to "program." We do not 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 do not match. As long as we label the example phrases during training, so that the chatbot knows which is which, it will learn to discriminate between them. And there are even machine learning approaches that require little if any "labeled" data.

We have given you some exciting reasons to learn about natural language processing. You want to help save the world, do you not? And we have attempted to pique your interest with some practical NLP applications that are revolutionizing the way we communicate, learn, do business, and even think. It will not be long before you are able to build a system that approaches human-like conversational behavior. And you should be able to see in upcoming chapters how to train a chatbot or NLP pipeline with any domain knowledge that interests you—from finance and sports to psychology and literature. If you can find a corpus of writing about it, then you can train a machine to understand it.

This book is about using machine learning to build smart text reading machines without you having to anticipate all the ways people can say things. Each chapter incrementally improves on the basic NLP pipeline for the

chatbot introduced in this chapter. As you learn the tools of natural language processing, you will be building an NLP pipeline that can not only carry on a conversation but help you accomplish your goals in business and in life.

[58] More complicated logic and behaviors are now possible when you incorporate Lambdas into an AWS Contact Flow dialog tree. See "Creating Call Center Bot with AWS Connect" (<https://greenice.net/creating-call-center-bot-aws-connect-amazon-lex-can-speak-understand>).

[59] A good question suggested by Byron Reese is: "What's larger? The sun or a nickel?" (<https://gigaom.com/2017/11/20/voices-in-ai-episode-20-a-conversation-with-marie-des-jardins>). Here are a couple more (https://gitlab.com/tangibleai/nlpia2/-/raw/main/src/nlpia2/data/iq_test.csv) to get you started.

1.12 Review

Chapter 1 review questions

Here are some review questions for you to test your understanding:

1. Why is NLP considered to be a core enabling feature for AGI (human-like AI)?
2. Why do advanced NLP models tend to show significant discriminatory biases?
3. How is it possible to create a prosocial chatbot using training data from sources that include antisocial examples?
4. What are 4 different approaches or architectures for building a chatbot?
5. How is NLP used within a search engine?
6. Write a regular expression to recognize your name and all the variations on its spelling (including nicknames) that you've seen.
7. Write a regular expression to try to recognize a sentence boundary (usually a period ("."), question mark "?", or exclamation mark "!")



Tip

Active learning, quizzing yourself with questions such as these, is a fast way to gain deep understanding for any new topic. It turns out, this same approach is effective for machine learning and model evaluation as well.^{footnote}: [Suggested answers are provided within the Python packages `nlpia` (<https://gitlab.com/tangibleai/nlpia>) and `qary` (<https://gitlab.com/tangibleai/qary>) where they are used to evaluate advanced NLP models for reading comprehension and question answering. Pooja Sethi will share active learning NLP insights on Substack (<https://activelearning.substack.com>) and github (<https://poojasethi.github.io>) by the time this book goes to print. ProAI.org, the team of contributing authors for this book is doing the same on substack (<https://proai.substack.com>) and their home page (<https://proai.org>).

1.13 Summary

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

2 Tokens of thought (natural language words)

This chapter covers

- Parsing your text into words and n -grams (tokens)
- Tokenizing punctuation, emoticons, and even Chinese characters
- Consolidating your vocabulary with stemming, lemmatization, and case folding
- Building a structured numerical representation of natural language text
- Scoring text for sentiment and prosocial intent
- Using character frequency analysis to optimize your token vocabulary
- Dealing with variable length sequences of words and tokens

So you want to help save the world with the power of natural language processing (NLP)? First your NLP pipeline will need to compute something about text, and for that you'll need a way to represent text in a numerical data structure. The part of an NLP pipeline that breaks up your text to create this structured numerical data is called a *parser*. For many NLP applications, you only need to convert your text to a sequence of words, and that can be enough for searching and classifying text.

You will now learn how to split a document, any string, into discrete tokens of meaning. You will be able to parse text documents as small as a single word and as large as an entire Encyclopedia. And they will all produce a consistent representation that you can use to compare them. For this chapter your tokens will be words, punctuation marks, and even pictograms such as Chinese characters, emojis and emoticons.

Later in the book you will see that you can use these same techniques to find packets of meaning in any discrete sequence. For example, your tokens could be the ASCII characters represented by a sequence of bytes, perhaps with ASCII emoticons. Or they could be Unicode emojis, mathematical symbols, Egyption, hieroglyphics, pictographs from languages like Kanji and

Cantonese. You could even define the tokens for DNA and RNA sequences with letters for each of the five base nucleotides: adenine (A), guanine (G), cytosine (C), thymine (T), and uracil (U). Natural language sequences of tokens are all around you ... and even inside you.

Is there something you can do with tokens that doesn't require a lot of complicated deep learning? If you have a good tokenizer you can use it to identify statistics about the occurrence of tokens in a set of documents, such as your blog posts or a business website. Then you can build a search engine in pure Python with just a dictionary to represent to record links to the set of documents where those words occur. That Python dictionary that maps words to document links or pages is called a reverse index. It's just like the index at the back of this book. This is called *information retrieval*—a really powerful tool in your NLP toolbox.

Statistics about tokens are often all you need for keyword detection, full text search, and information retrieval. You can even build customer support chatbots using text search to find answers to customers' questions in your documentation or FAQ (frequently asked question) lists. A chatbot can't answer your questions until it knows where to look for the answer. Search is the foundation of many state of the art applications such as conversational AI and open domain question answering. A tokenizer forms the foundation for almost all NLP pipelines.

2.1 Tokens of emotion

Another practical use for your tokenizer is called *sentiment analysis*, or analysis of text to estimate emotion. You'll see an example of a sentiment analysis pipeline later in this chapter. For now you just need to know how to build a tokenizer. And your tokenizer will almost certainly need to handle the tokens of emotion called *emoticons* and *emojis*.

Emoticons are a textual representations of a writer's mood or facial expression, such as the *smiley* emoticon: : -). They are kind-of like a modern hieroglyph or picture-word for computer users that only have access to an ASCII terminal for communication. *Emojis* are the graphical representation of these characters. For example, the smilie emoji has a small yellow circle

with two black dots for eyes and a U shaped curve for a mouth. The smiley emoji is a graphical representation of the : -) smiley emoticon.

Both emojis and emoticons have evolved into their own language. There are hundreds of popular emojis. People have created emojis for everything from company logos to memes and innuendo. Noncommercial social media networks such Mastodon even allow you to create your own custom emojis. [60] [61]



Emojis and Emoticons

Emoticons were first typed into an ASCII text message in 1972 when Carnegie Mellon researchers mistakenly understood a text message about a mercury spill to be a joke. The professor, Dr. Scott E. Fahlman, suggested that : -) should be appended to messages that were jokes, and : - (emoticons should be used for serious warning messages. Gosh, how far we've come.

The plural of "emoji" is either "emoji" (like "sushi") or "emojis" (like "Tsunamis"), however the the Atlantic and NY Times style editors prefer "emojis" to avoid ambiguity. Your NLP pipeline will learn what you mean no matter how you type it.



[60] Mastodon servers you can join (<https://proai.org/mastoserv>)

[61] Mastodon custom emoji documentation
(https://docs.joinmastodon.org/methods/custom_emojis/)

2.2 What is a token?

A token can be almost any chunk of text that you want to treat as a packet of thought and emotion. So you need to break your text into chunks that capture individual thoughts. You may be thinking that *words* are the obvious choice for tokens. So that's what you will start with here. You'll also learn how to

include punctuation marks, emojis, numbers, and other word-like things in your vocabulary of words. Later you'll see that you can use these same techniques to find packets of meaning in any discrete sequence. And later you will learn some even more powerful ways to split discrete sequences into meaningful packets. Your tokenizers will be soon able to analyze and structure any text document or string, from a single word, to a sentence, to an entire book.

Think about a collection of documents, called a *corpus*, that you want to process with NLP. Think about the *vocabulary* that would be important to your NLP algorithm—the set of tokens you will need to keep track of. For example your tokens could be the characters for ASCII emoticons, if this is what is important in your NLP pipeline for a particular corpus. Or your tokens could be Unicode emojis, mathematical symbols, hieroglyphics, even pictographs like Kanji and Cantonese characters. Your tokenizer and your NLP pipeline would even be useful for the nucleotide sequences of DNA and RNA where your tokens might be A, C, T, G, U, and so on. And neuroscientists sometimes create sequences of discrete symbols to represent neurons firing in your brain when you read text like this sentence. Natural language sequences of tokens are inside you, all around you, and flowing through you. Soon you'll be flowing streams of tokens through your machine learning NLP pipeline.

Retrieving tokens from a document will require some string manipulation beyond just the `str.split()` method employed in chapter 1. You'll probably want to split contractions like "you'll" into the words that were combined to form them, perhaps "you" and "ll", or perhaps "you" and "will." You'll want to separate punctuation from words, like quotes at the beginning and end of quoted statements or words, such as those in the previous sentence. And you need to treat some punctuation such as dashes ("—") as part of singly-hyphenated compound words such as "singly-hyphenated."

Once you have identified the tokens in a document that you would like to include in your vocabulary, you will return to the regular expression toolbox to build a tokenizer. And you can use regular expressions combine different forms of a word into a single token in your vocabulary—a process called *stemming*. Then you will assemble a vector representation of your documents

called a *bag of words*. Finally, you will try to use this bag of words vector to see if it can help you improve upon the basic greeting recognizer at the end of chapter 1.

2.2.1 Alternative tokens

Words aren't the only packets of meaning we could use for our tokens. Think for a moment about what a word or token represents to you. Does it represent a single concept, or some blurry cloud of concepts? Could you always be sure to recognize where a word begins and ends? Are natural language words like programming language keywords that have precise spellings, definitions and grammatical rules for how to use them? Could you write software that reliably recognizes a word?

Do you think of "ice cream" as one word or two? Or maybe even three? Aren't there at least two entries in your mental dictionary for "ice" and "cream" that are separate from your entry for the compound word "ice cream"? What about the contraction "don't"? Should that string of characters be split into one, or two, or even three packets of meaning?

You might even want to divide words into even smaller meaningful parts. Word pieces such as the prefix "pre", the suffix "fix", or the interior syllable "la" all have meaning. You can use these word pieces to transfer what you learn about the meaning of one word to another similar word in your vocabulary. Your NLU pipeline can even use these pieces to understand new words. And your NLG pipeline can use the pieces to create new words that succinctly capture ideas or memes circulating in the collective consciousness.

Your pipeline could break words into even smaller pieces. Letters, characters, or graphemes [\[62\]](#) carry sentiment and meaning too! [\[63\]](#) We haven't yet found the perfect encoding for packets of thought. And machines compute differently than brains. We explain language and concepts to each other in terms of words or terms. But machines can often see patterns in the use of characters that we miss. And for machines to be able to squeeze huge vocabularies into their limited RAM there are more efficient encodings for natural language.

The optimal tokens for efficient computation are different from the packets of thought (words) that we humans use. Byte Pair Encoding (BPE), Word Piece Encoding, and Sentence Piece Encoding, each can help machines use natural language more efficiently. BPE finds the optimal groupings of characters (bytes) for your particular set of documents and strings. If you want an **explainable** encoding, use the word tokenizers of the previous sections. If you want more flexible and accurate predictions and generation of text, then BPE, WPE, or SPE may be better for your application. Like the bias variance trade-off, there's often a explainability/accuracy trade-off in NLP.

What about invisible or implied words? Can you think of additional words that are implied by the single-word command "Don't!"? If you can force yourself to think like a machine and then switch back to thinking like a human, you might realize that there are three invisible words in that command. The single statement "Don't!" means "Don't you do that!" or "You, do not do that!" That's at least three hidden packets of meaning for a total of five tokens you'd like your machine to know about.

But don't worry about invisible words for now. All you need for this chapter is a tokenizer that can recognize words that are spelled out. You will worry about implied words and connotation and even meaning itself in chapter 4 and beyond. [\[64\]](#)

Your NLP pipeline can start with one of these five options as your tokens:

1. **Bytes** - ASCII characters
2. **Characters** - multi-byte Unicode characters
3. **Subwords** (Word pieces) - syllables and common character clusters
4. **Words** - dictionary words or their roots (stems, lemmas)
5. **Sentence pieces** - short, common word and multi-word pieces

As you work your way down this list your vocabulary size increases and your NLP pipeline will need more and more data to train. Character-based NLP pipelines are often used in translation problems or NLG tasks that need to generalize from a modest number of examples. The number of possible words that your pipeline can deal with is called its *vocabulary*. A character-based NLP pipeline typically needs fewer than 200 possible tokens to process many Latin-based languages. That small vocabulary ensures that byte- and

character-based NLP pipelines can handle new unseen test examples without too many meaningless OOV (out of vocabulary) tokens.

For word-based NLP pipelines your pipeline will need to start paying attention to how often tokens are used before deciding whether to "count it." You don't want your pipeline to do anything meaningful with junk words such asdf - the But even if you make sure your pipeline only pays attention to words that occur a lot, you could end up with a vocabulary that's as large as a typical dictionary - 20 to 50 thousand words.

Subwords are the optimal token to use for most Deep Learning NLP pipelines. Subword (Word piece) tokenizers are built into many state-of-the-art transformer pipelines. Words are the token of choice for any linguistics project or academic research where your results need to be interpretable and explainable.

Sentence pieces take the subword algorithm to the extreme. The sentence piece tokenizer allows your algorithm to combine multiple word pieces together into a single token that can sometimes span multiple words. The only hard limit on sentence pieces is that they do not extend past the end of a sentence. This ensures that the meaning of a token is associated with only a single coherent thought and is useful on single sentences as well as longer documents.

N-grams

No matter which kind of token you use for your pipeline, you will likely extract pairs, triplets, quadruplets, and even quintuplets of tokens. These are called *n*-grams_. [\[65\]](#) Using *n*-grams enables your machine to know about the token "ice cream" as well as the individual tokens "ice" and "cream" that make it up. Another 2-gram that you'd like to keep together is "Mr. Smith". Your tokens and your vector representation of a document will likely want to have a place for "Mr. Smith" along with "Mr." and "Smith."

You will start with a short list of keywords as your vocabulary. This helps to keep your data structures small and understandable and can make it easier to explain your results. Explainable models create insights that you can use to

help your stakeholders, hopefully the users themselves (rather than investors), accomplish their goals.

For now, you can just keep track of all the short n -grams of words in your vocabulary. But in chapter 3, you will learn how to estimate the importance of words based on their document frequency, or how often they occur. That way you can filter out pairs and triplets of words that rarely occur together. You will find that the approaches we show are not perfect. Feature extraction can rarely retain all the information content of the input data in any machine learning pipeline. That is part of the art of NLP, learning when your tokenizer needs to be adjusted to extract more or different information from your text for your particular applications.

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

[62] (<https://en.wikipedia.org/wiki/Grapheme>)

[63] Suzi Park and Hyopil Shin *Grapheme-level Awareness in Word Embeddings for Morphologically Rich Languages* (<https://www.aclweb.org/anthology/L18-1471.pdf>)

[64] If you want to learn more about exactly what a "word" really is, check out the introduction to *The Morphology of Chinese* by Jerome Packard where he discusses the concept of a "word" in detail. The concept of a "word" did not exist at all in the Chinese language until the 20th century when it was translated from English grammar into Chinese.

[65] Pairs of adjacent words are called 2-grams or bigrams. Three words in sequence are called 3-grams or trigrams. Four words in a row are called 4-grams. 5-grams are probably the longest n -grams you'll find in an NLP pipeline. Google counts all the 1 to 5-grams in nearly all the books ever written (<https://books.google.com/ngrams>).

2.3 Challenges (a preview of stemming)

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

Or imagine trying to discriminate between a pluralizing "s" at the end of a word like "words" and a normal "s" at the end of words like "bus" and "lens." Do isolated individual letters in a word or parts of a word provide any information at all about that word's meaning? Can the letters be misleading? Yes and yes.

In this chapter we show you how to make your NLP pipeline a bit smarter by dealing with these word spelling challenges using conventional stemming approaches. Later, in chapter 5, we show you statistical clustering approaches that only require you to amass a collection of natural language text containing the words you are interested in. From that collection of text, the statistics of word usage will reveal "semantic stems" (actually, more useful clusters of words like lemmas or synonyms), without any hand-crafted regular expressions or stemming rules.

2.3.1 Tokenization

In NLP, *tokenization* is a particular kind of document *segmentation*. Segmentation breaks up text into smaller chunks or segments. The segments of text have less information than the whole. Documents can be segmented into paragraphs, paragraphs into sentences, sentences into phrases, and phrases into tokens (usually words and punctuation). In this chapter, we focus

on segmenting text into *tokens* with a *tokenizer*.

You may have heard of tokenizers before. If you took a computer science class you likely learned about how programming language compilers work. A tokenizer that is used to compile computer languages is called a *scanner* or *lexer*. In some cases your computer language parser can work directly on the computer code and doesn't need a tokenizer at all. And for natural language processing, the only parser typically outputs a vector representation, rather than If the tokenizer functionality is not separated from the compiler, the parser is often called a scannerless *parser*.

The set of valid tokens for a particular computer language is called the *vocabulary* for that language, or more formally its *lexicon*. Linguistics and NLP researchers use the term "lexicon" to refer to a set of natural language tokens. The term "vocabulary" is the more natural way to refer to a set of natural language words or tokens. So that's what you will use here.

The natural language equivalent of a computer language compiler is a natural language parser. A natural language tokenizer is called a *scanner*, or *lexer*, or *lexical analyzer* in the computer language world. Modern computer language compilers combine the *lexer* and *parser* into a single lexer-parser algorithm. The vocabulary of a computer language is usually called a *lexicon*. And computer language compilers sometimes refer to tokens as *symbols*.

Here are five important NLP terms. Along side them are some roughly equivalent terms used in computer science when talking about programming language compilers:

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

Tokenization is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline. A tokenizer breaks unstructured data, natural language text, into chunks of information which can be counted as discrete elements. These counts of token occurrences in a document can be used

directly as a vector representing that document. This immediately turns an unstructured string (text document) into a numerical data structure suitable for machine learning. These counts can be used directly by a computer to trigger useful actions and responses. Or they might also be used in a machine learning pipeline as features that trigger more complex decisions or behavior. The most common use for bag-of-words vectors created this way is for document retrieval, or search.

2.4 Your tokenizer toolbox

So each application you encounter you will want to think about which kind of tokenizer is appropriate for your application. And once you decide which kinds of tokens you want to try, you'll need to configure a python package for accomplishing that goal.

You can chose from several tokenizer implementations: [\[66\]](#)

1. Python: `str.split`, `re.split`
2. NLTK: `PennTreebankTokenizer`, `TweetTokenizer`
3. spaCy: state of the art tokenization is its reason for being
4. Stanford CoreNLP: linguistically accurate, requires Java interpreter
5. Huggingface: `BertTokenizer`, a WordPiece tokenizer

2.4.1 The simplest tokenizer

The simplest way to tokenize a sentence is to use whitespace 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` object instances as well as on the `str` built-in class itself.

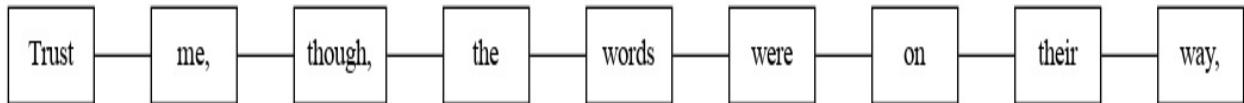
Let's say your NLP pipeline needs to parse quotes from WikiQuote.org, and it's having trouble with one titled *The Book Thief*.[\[67\]](#)

Listing 2.1. Example quote from *The Book Thief* split into tokens

```
>>> text = ("Trust me, though, the words were on their way, and w  
...           "they arrived, Liesel would hold them in her hands li
```

```
...           "the clouds, and she would wring them out, like the r
>>> tokens = text.split()
>>> tokens[:8]
['Trust', 'me,', 'though,', 'the', 'words', 'were', 'on', 'their'
```

Figure 2.1. Tokenized phrase



As you can see, this built-in Python method does an OK job of tokenizing this sentence. Its only "mistake" is to include commas within the tokens. This would prevent your keyword detector from detecting quite a few important tokens: ['me', 'though', 'way', 'arrived', 'clouds', 'out', "rain"]. Those words "clouds" and "rain" are pretty important to the meaning of this text. So you'll need to do a bit better with your tokenizer to ensure you can catch all the important words and "hold" them like Liesel.

2.4.2 Rule-based tokenization

It turns out there is a simple fix to the challenge of splitting punctuation from words. You can use a regular expression tokenizer to create rules to deal with common punctuation patterns. Here's just one particular regular expression you could use to deal with punctuation "hanger-ons." And while we're at it, this regular expression will be smart about words that have internal punctuation, such as possessive words and contractions that contain apostrophes.

You'll use a regular expression to tokenize some text from the book *Blindsight* by Peter Watts. The text describes how the most *adequate* humans tend to survive natural selection (and alien invasions).[\[68\]](#) The same goes for your tokenizer. You want to find an *adequate* tokenizer that solves your problem, not the perfect tokenizer. You probably can't even guess what the *right* or *fittest* token is. You will need an accuracy number to evaluate your NLP pipeline with and that will tell you which tokenizer should survive your selection process. The example here should help you start to develop your intuition about applications for regular expression tokenizers.

```
>>> import re
>>> pattern = r'\w+(?:\'\w+)?|[^\w\s]' #1
>>> texts = [text]
>>> texts.append("There's no such thing as survival of the fittes
...           "Survival of the most adequate, maybe.")
>>> tokens = list(re.findall(pattern, texts[-1]))
>>> tokens[:8]
['There''s', 'no', 'such', 'thing', 'as', 'survival', 'of', 'the']
>>> tokens[8:16]
['fittest', '.', 'Survival', 'of', 'the', 'most', 'adequate', ',', '
>>> tokens[16:]
['maybe', '.']
```

Much better. Now the tokenizer separates punctuation from the end of a word, but doesn't break up words that contain internal punctuation such as the apostrophe within the token "There's." So all of these words were tokenized the way we wanted: "There's", "fittest", "maybe". And this regular expression tokenizer will work fine on contractions even if they have more than one letter after the apostrophe such as "can't", "she'll", "what've". It will work even typos such as 'can't' and "she,ll", and "what`ve". But this liberal matching of internal punctuation probably isn't what you want if your text contains rare double contractions such as "couldn't've", "ya'll'll", and "y'ain't"



Tip

Pro tip: You can accommodate double-contractions with the regular expression `r'\w+(?:\'\w+){0,2}|[^\w\s]'`

This is the main idea to keep in mind. No matter how carefully you craft your tokenizer, it will likely destroy some amount of information in your raw text. As you are cutting up text, you just want to make sure the information you leave on the cutting room floor isn't necessary for your pipeline to do a good job. Also, it helps to think about your downstream NLP algorithms. Later you may configure a case folding, stemming, lemmatizing, synonym substitution, or count vectorizing algorithm. When you do, you'll have to think about what your tokenizer is doing, so your whole pipeline works together to accomplish your desired output.

Take a look at the first few tokens in your lexicographically sorted vocabulary for this short text:

```
>>> import numpy as np #1
>>> vocab = sorted(set(tokens)) #2
>>> ' '.join(vocab[:12]) #3
", . Survival There's adequate as fittest maybe most no of such"
>>> num_tokens = len(tokens)
>>> num_tokens
18
>>> vocab_size = len(vocab)
>>> vocab_size
15
```

You can see how you may want to consider lowercasing all your tokens so that "Survival" is recognized as the same word as "survival". And you may want to have a synonym substitution algorithm to replace "There's" with "There is" for similar reasons. However, this would only work if your tokenizer kept contraction and possessive apostrophes attached to their parent token.



Tip

Make sure you take a look at your vocabulary whenever it seems your pipeline isn't working well for a particular text. You may need to revise your tokenizer to make sure it can "see" all the tokens it needs to do well for your NLP task.

2.4.3 SpaCy

Maybe you don't want your regular expression tokenizer to keep contractions together. Perhaps you'd like to recognize the word "isn't" as two separate words, "is" and "n't". That way you could consolidate the synonyms "n't" and "not" into a single token. This way your NLP pipeline would understand "the ice cream isn't bad" to mean the same thing as "the ice cream is not bad". For some applications, such as full text search, intent recognition, and sentiment analysis, you want to be able to **uncontract** or expand contractions like this. By splitting contractions, you can use synonym substitution or contraction expansion to improve the recall of your search engine and the

accuracy of your sentiment analysis.



Important

We'll discuss case folding, stemming, lemmatization, and synonym substitution later in this chapter. Be careful about using these techniques for applications such as authorship attribution, style transfer, or text fingerprinting. You want your authorship attribution or style-transfer pipeline to stay true to the author's writing style and the exact spelling of words that they use.

SpaCy integrates a tokenizer directly into its state-of-the-art NLU pipeline. In fact the name "spaCy" is based on the word "space", as in the separator used in Western languages to separate words. And spaCy adds a lot of additional *tags* to tokens at the same time that it is applying rules to split tokens apart. So spaCy is often the first and last tokenizer you'll ever need to use.

Let's see how spaCy handles our collection of deep thinker quotes:

```
>>> import spacy #1
>>> nlp = spacy.load('en_core_web_sm') #2
>>> doc = nlp(texts[-1])
>>> type(doc)
<class 'spacy.tokens.doc.Doc'>

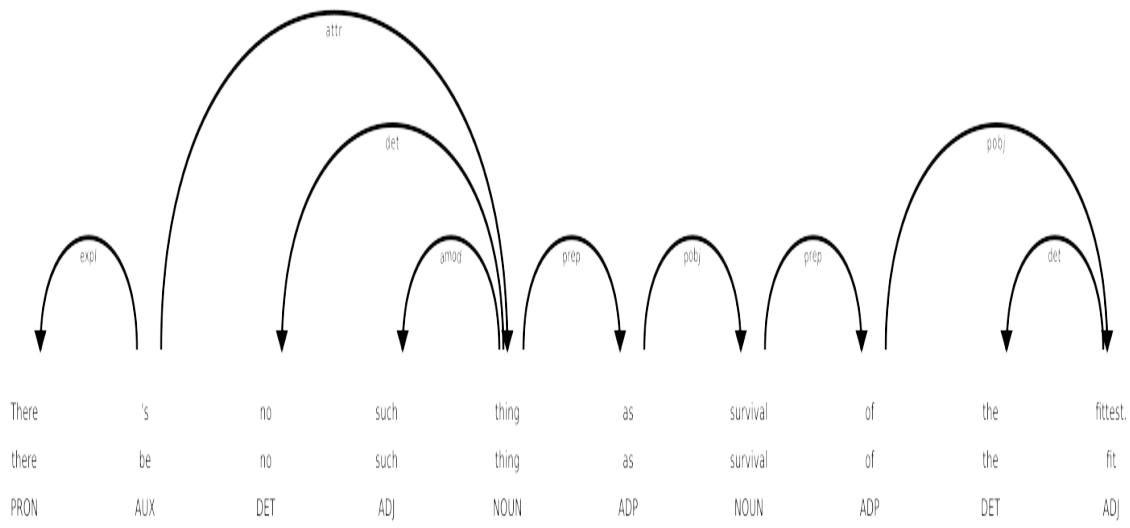
>>> tokens = [tok.text for tok in doc]
>>> tokens[:9]
['There', "'s", 'no', 'such', 'thing', 'as', 'survival', 'of', 't
>>> tokens[9:17]
['fittest', '.', 'Survival', 'of', 'the', 'most', 'adequate', ',', '
```

That tokenization may be more useful to you if you're comparing your results to academic papers or colleagues at work. Spacy is doing a lot more under the hood. That small language model you downloaded is also identifying sentence breaks with some **sentence boundary detection** rules. A language model is a collection of regular expressions and finite state automata (rules). These rules are a lot like the grammar and spelling rules you learned in English class. They are used in the algorithms that tokenize and label your

words with useful things like their part of speech and their position in a syntax tree of relationships between words.

```
>>> from spacy import displacy  
>>> sentence = list(doc.sents)[0] #1  
>>> displacy.serve(sentence, style="dep")  
>>> !firefox 127.0.0.1:5000
```

If you browse to your localhost on port 5000 you should see a sentence diagram that may be even more correct than what you could produce in school:



You can see that spaCy does a lot more than simply separate text into tokens. It identifies sentence boundaries to automatically segment your text into sentences. And it tags tokens with various attributes like their part of speech (PoS) and even their role within the syntax of a sentence. You can see the lemmas displayed by displacy beneath the literal text for each token.[\[70\]](#) Later in the chapter we'll explain how lemmatization and case folding and other vocabulary **compression** approaches can be helpful for some applications.

So spaCy seems pretty great in terms of accuracy and some "batteries included" features, such as all those token tags for lemmas and dependencies.

What about speed?

2.4.4 Tokenizer race

SpaCy can parse the AsciiDoc text for a chapter in this book in about 5 seconds. First download the AsciiDoc text file for this chapter:

```
>>> import requests  
>>> text = requests.get('https://proai.org/nlpia2-ch2.adoc').text  
>>> f'{round(len(text) / 10_000)}0k' #1  
'160k'
```

There were about 160 thousand ASCII characters in this AsciiDoc file where I wrote this sentence that you are reading right now. What does that mean in terms of words-per-second, the standard benchmark for tokenizer speed?

```
>>> import spacy  
>>> nlp = spacy.load('en_core_web_sm')  
>>> %timeit nlp(text) #1  
4.67 s ± 45.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)  
  
>>> f'{round(len(text) / 10_000)}0k'  
'160k'  
>>> doc = nlp(text)  
>>> f'{round(len(list(doc)) / 10_000)}0k'  
'30k'  
>>> f'{round(len(doc) / 1_000 / 4.67)}kWPS' #2  
'7kWPS'
```

That's nearly 5 seconds for about 150,000 characters or 34,000 words of English and Python text or about 7000 words per second.

That may seem fast enough for you on your personal projects. But on a medical records summarization project we needed to process thousands of large documents with a comparable amount of text as you find in this entire book. And the latency in our medical record summarization pipeline was a critical metric for the project. So this, full-featured spaCy pipeline would require at least 5 days to process 10,000 books such as NLPIA or typical medical records for 10,000 patients.

If that's not fast enough for your application you can disable any of the

tagging features of the spaCy pipeline that you do not need.

```
>>> nlp.pipe_names #1
['tok2vec', 'tagger', 'parser', 'attribute_ruler', 'lemmatizer',
>>> nlp = spacy.load('en_core_web_sm', disable=nlp.pipe_names)
>>> %timeit nlp(text)
199 ms ± 6.63 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

You can disable the pipeline elements you don't need to speed up the tokenizer:

- tok2vec: word embeddings
- tagger: part-of-speech (.pos and .pos_)
- parser: syntax tree role
- attribute_ruler: fine-grained POS and other tags
- lemmatizer: lemma tagger
- ner: named entity recognition tagger

NLTK's `word_tokenize` method is often used as the pace setter in tokenizer benchmark speed comparisons:

```
>>> import nltk
>>> nltk.download('punkt')
True
>>> from nltk.tokenize import word_tokenize
>>> %timeit word_tokenize(text)
156 ms ± 1.01 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
>>> tokens = word_tokenize(text)
>>> f'{round(len(tokens) / 10_000)}0k'
'30k'
```

Could it be that you found a winner for the tokenizer race? Not so fast. Your regular expression tokenizer has some pretty simple rules, so it should run pretty fast as well:

```
>>> pattern = r'\w+(?:(\w+)?|[^\w\s])'
>>> tokens = re.findall(pattern, text) #1
>>> f'{round(len(tokens) / 10_000)}0k'
'30k'
>>> %timeit re.findall(pattern, text)
8.77 ms ± 29.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Now that's not surprising. Regular expressions can be compiled and run very efficiently within low level C routines in Python.



Tip

Use a regular expression tokenizer when speed is more import than accuracy. If you do not need the additional linguistic tags that spaCy and other pipelines provide your tokenizer doesn't need to waste time trying to figure out those tags.^[71] And each time you use a regular expression in the `re` or `regex` packages, a compiled and optimized version of it is cached in RAM. So there's usually no need to *precompile* (using `re.compile()`) your regexes.

[66]^[66] Lysandre explains the various tokenizer options in the Huggingface documentation
(https://huggingface.co/transformers/tokenizer_summary.html)

[67]^[67] Markus Zusak, *The Book Thief*, p. 85
(https://en.wikiquote.org/wiki/The_Book_Thief)

[68]^[68] Peter Watts, Blindsight, (<https://rifters.com/real/Blindsight.htm>)

[69]^[69] Thank you Wiktor Stribiżew
(<https://stackoverflow.com/a/43094210/623735>).

[70]^[70] nlpia2 source code for chapter 2 (<https://proai.org/nlpia2-ch2>) has additional spaCy and displacy options and examples.

[71]^[71] Andrew Long, "Benchmarking Python NLP Tokenizers"
(<https://towardsdatascience.com/benchmarking-python-nlp-tokenizers-3ac4735100c5>)

2.5 Wordpiece tokenizers

It probably felt natural to think of words as indivisible atomic chunks of meaning and thought. However, you did find some words that didn't clearly split on spaces or punctuation. And many compound words or named entities

that you'd like to keep together have spaces within them. So it can help to dig a little deeper and think about the statistics of what makes a word. Think about how we can build up words from neighboring characters instead of cleaving text at separators such as spaces and punctuation.

2.5.1 Clumping characters into sentence pieces

Instead of thinking about breaking strings up into tokens, your tokenizer can look for characters that are used a lot right next to each other, such as "i" before "e". You can pair up characters and sequences of characters that belong together.^[72] These clumps of characters can become your tokens. An NLP pipeline only pays attention to the statistics of tokens. And hopefully these statistics will line up with our expectations for what a word is.

Many of these character sequences will be whole words, or even compound words, but many will be pieces of words. In fact, all *subword tokenizers* maintain a token within the vocabulary for every individual character in your vocabulary. This means it never needs to use an OOV (Out-of-Vocabulary) token, as long as any new text doesn't contain any new characters it hasn't seen before. Subword tokenizers attempt to optimally clump characters together to create tokens. Using the statistics of character n-gram counts it's possible for these algorithms to identify wordpieces and even sentence pieces that make good tokens.

It may seem odd to identify words by clumping characters. But to a machine, the only obvious, consistent division between elements of meaning in a text is the boundary between bytes or characters. And the frequency with which characters are used together can help the machine identify the meaning associated with subword tokens such as individual syllables or parts of compound words.

In English, even individual letters have subtle emotion (sentiment) and meaning (semantics) associated with them. However, there are only 26 unique letters in the English language. That doesn't leave room for individual letters to *specialize* on any one topic or emotion. Nonetheless savvy marketers know that some letters are cooler than others. Brands will try to portray themselves as technologically advanced by choosing names with

exotic letters like "Q" and "X" or "Z". This also helps with SEO (Search Engine Optimization) because rarer letters are more easily found among the sea of possible company and product names. Your NLP pipeline will pick up all these hints of meaning, connotation, and intent. Your token counters will provide the machine with the statistics it needs to infer the meaning of clumps of letters that are used together often.

The only disadvantage for subword tokenizers is the fact that they must pass through your corpus of text many times before converging on an optimal vocabulary and tokenizer. A subword tokenizer has to be trained or fit to your text just like a CountVectorizer. In fact you'll use a CountVectorizer in the next section to see how subword tokenizers work.

There are two main approaches to subword tokenization: BPE (Byte-Pair Encoding) and Wordpiece tokenization.

BPE

In the previous edition of the book we insisted that words were the smallest unit of meaning in English that you need consider. With the rise of Transformers and other deep learning models that use BPE and similar techniques, we've changed our minds.^[73] Character-based subword tokenizers have proven to be more versatile and robust for most NLP problems. By building up a vocabulary from building blocks of Unicode multi-byte characters you can construct a vocabulary that can handle every possible natural language string you'll ever see, all with a vocabulary of as few as 50,000 tokens.

You may think that Unicode characters are the smallest packet of meaning in natural language text. To a human, maybe, but to a machine, no way. Just as the BPE name suggests, characters don't have to be your fundamental atom of meaning for your *base vocabulary*. You can split characters into 8-bit bytes. GPT-2 uses a byte-level BPE tokenizer to naturally compose all the unicode characters you need from the bytes that make them up. Though some special rules are required to handle unicode punctuation within a byte-based vocabulary, no other adjustment to the character-based BPE algorithm is required. A byte-level BPE tokenizer allows you to represent all possible

texts with a base (minimum) vocabulary size of 256 tokens. The GPT-2 model can achieve state-of-the-art performance with its default BPE vocabulary of only 50,000 multibyte *merge tokens* plus 256 individual byte tokens.

You can think of the BPE (Byte Pair Encoding) tokenizer algorithm as a matchmaker or the hub in a social network. It connects characters together that appear next to each other a lot. It then creates a new token for these character combinations. And it keeps doing this until it has as many frequently used character sequences as you've allowed in your vocabulary size limit.

BPE is transforming the way we think about natural language tokens. NLP engineers are finally letting the data do the talking. Statistical thinking is better than human intuition when building an NLP pipeline. A machine can see how *most* people use language. You are only familiar with what *you* mean when you use particular words or syllables. Transformers have now surpassed human readers and writers at some natural language understanding and generation tasks, including finding meaning in subword tokens.

One complication you have not yet encountered is the dilemma of what to do when you encounter a new word. In the previous examples, we just keep adding new words to our vocabulary. But in the real world your pipeline will have been trained on an initial corpus of documents that may or may not represent all the kinds of tokens it will ever see. If your initial corpus is missing some of the words that you encounter later on, you will not have a slot in your vocabulary to put your counts of that new word. So when you train your initial pipeline, you will always reserve a slot (dimension) to hold the counts of your *out-of-vocabulary* (OOV) tokens. So if your original set of documents did not contain the girl's name "Aphra", all counts of the name Aphra would be lumped into the OOV dimension as counts of Amandine and other rare words.

To give Aphra equal representation in your vector space, you can use BPE. BPE breaks down rare words into smaller pieces to create a *periodic table* of the elements for natural language in your corpus. So, because "aphr" is a common English prefix, your BPE tokenizer would probably give Aphra **two** slots for her counts in your vocabulary: one for "aphr" and one for "a". Actually, you might actually discover that the vocabulary slots are for "

aphr" and "a ", because BPE keeps track of spaces no differently than any other character in your alphabet.[\[74\]](#)

BPE gives you multilingual flexibility to deal with Hebrew names like Aphra. And it give your pipeline robustness against common misspellings and typos, such as "aphradesiac." Every word, including minority 2-grams such as "African American", have representation in the voting system of BPE.[\[75\]](#) Gone are the days of using the kluge of OOV (Out-of-Vocabulary) tokens to handle the rare quirks of human communication. Because of this, state of the art deep learning NLP pipelines such as transformers all use word piece tokenization similar to BPE.[\[76\]](#)

BPE preserves some of the meaning of new words by using character tokens and word-piece tokens to spell out any unknown words or parts of words. For example, if "syzygy" is not in our vocabulary, we could represent it as the six tokens "s", "y", "z", "y", "g", and "y". Perhaps "smartz" could be represented as the two tokens "smart" and "z".

That sounds smart. Let's see how it works on our text corpus:

```
>>> import pandas as pd
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> vectorizer = CountVectorizer(ngram_range=(1, 2), analyzer='ch
>>> vectorizer.fit(texts)
CountVectorizer(analyzer='char', ngram_range=(1, 2))

>>> bpevocab = vectorizer.get_feature_names()
>>> bpevocab[:7]
[' ', ' a', ' c', ' f', ' h', ' i', ' l']
```

We configured the CountVectorizer to split the text into all the possible character 1-grams and 2-grams found in the texts. And CountVectorizer organizes the vocabulary in lexical order, so n-grams that start with a space character (' ') come first. Once the vectorizer knows what tokens it needs to be able to count, it can transform text strings into vectors, with one dimension for every token in your character n-gram vocabulary.

```
>>> vectors = vectorizer.transform(texts)
>>> df = pd.DataFrame(vectors.todense(), columns=bpevocab)
>>> df.index = [t[:8] + '...' for t in texts]
```

```

>>> df = df.T
>>> df['total'] = df.T.sum()
>>> df
   Trust me... There's ... total
      31          14        45
a            3          2        5
c            1          0        1
f            0          1        1
h            3          0        3
...
wr           1          0        1
y            2          1        3
y            1          0        1
y,
yb           0          1        1
<BLANKLINE>
[148 rows x 3 columns]

```

The DataFrame contains a column for each sentence and a row for each character 2-gram. Check out the top four rows where the byte pair (character 2-gram) of " a" is seen to occur five times in these two sentences. So even spaces count as "characters" when you're building a BPE tokenizer. This is one of the advantages of BPE, it will figure out what your token delimiters are, so it will work even in languages where there is no whitespace between words. And BPE will work on substitution cypher text like ROT13, a toy cypher that rotates the alphabet 13 characters forward.

```

>>> df.sort_values('total').tail()
   Trust me... There's ... total
he           10          3       13
h            14          5       19
t            11          9       20
e            18          8       26
31          14        45

```

A BPE tokenizer then finds the most frequent 2-grams and adds them to the permanent vocabulary. Over time it deletes the less frequent character pairs as it gets less and less likely that they won't come up a lot more later in your text.

```

>>> df['n'] = [len(tok) for tok in bpevocab]
>>> df[df['n'] > 1].sort_values('total').tail()
   Trust me... There's ... total n
,             6          1       7  2

```

| | | | | |
|----|----|---|----|---|
| e | 7 | 2 | 9 | 2 |
| t | 8 | 3 | 11 | 2 |
| th | 8 | 4 | 12 | 2 |
| he | 10 | 3 | 13 | 2 |

So the next round of preprocessing in the BPE tokenizer would retain the character 2-grams "he" and "th" and even " t" and "e ". Then the BPE algorithm would make another pass through the text with this smaller character bigram vocabulary. It would look for frequent pairings of these character bigrams with each other and individual characters. This process would continue until the maximum number of tokens is reached and the longest possible character sequences have been incorporated into the vocabulary.



Note

You may see mention of *wordpiece* tokenizers which are used within some advanced language models such as BERT and its derivatives.^[77] It works the same as BPE, but it actually uses the underlying language model to predict the neighboring characters in string. It eliminates the characters from its vocabulary that hurt the accuracy of this language model the least. The math is subtly different and it produces subtly different token vocabularies, but you don't need to select this tokenizer intentionally. The models that use it will come with it built into their pipelines.

One big challenge of BPE-based tokenizers is that they must be trained on your individual corpus. So BPE tokenizers are usually only used for Transformers and Large Language Models (LLMs) which you will learn about in chapter 9.

Another challenge of BPE tokenizers is all the book keeping you need to do to keep track of which trained tokenizer goes with each of your trained models. This was one of the big innovations of Huggingface. They made it easy to store and share all the preprocessing data, such as the tokenizer vocabulary, along side the language model. This makes it easier to reuse and share BPE tokenizers. If you want to become an NLP expert, you may want to imitate what they've done at HuggingFace with your own NLP

preprocessing pipelines.^[78]

[⁷²] In many applications the term "*n*-gram" refers to character *n*-grams rather than word *n*-grams. For example the leading relational database PostgreSQL has a Trigram index which tokenizes your text into character 3-grams not word 3-grams. In this book, we use "*n*-gram" to refer to sequences of word grams and "character *n*-grams" when talking about sequences of characters.

[⁷³] Hannes and Cole are probably screaming "We told you so!" as they read this.

[⁷⁴] Actually, the string representation of tokens used for BPE and Wordpiece tokenizer place marker characters at the beginning or end of the token string indicate the absence of a word boundary (typically a space or punctuation). So you may see the "aphr##" token in your BPE vocabulary for the prefix "aphr" in aphrodesiac (<https://stackoverflow.com/a/55416944/623735>)

[⁷⁵] Discriminatory voting restriction laws have recently been passed in US: (<https://proai.org/apnews-wisconsin-restricts-blacks>)

[⁷⁶] See chapter 12 for information about another similar tokenizer—sentence piece tokenizer

[⁷⁷] Lysandre Debut explains all the variations on subword tokenizers in the Hugging Face transformers documentation (https://huggingface.co/transformers/tokenizer_summary.html)

[⁷⁸] Huggingface documentation on tokenizers (https://huggingface.co/docs/transformers/tokenizer_summary)

2.6 Vectors of tokens

Now that you have broken your text into tokens of meaning, what do you do with them? How can you convert them to numbers that will be meaningful to the machine? The simplest most basic thing to do would be to detect whether a particular token you are interested in was present or not. You could hard-code the logic to check for important tokens, called a *keywords*.

This might work well for your greeting intent recognizer in chapter 1. Our greeting intent recognizer at the end of chapter 1 looked for words like "Hi" and "Hello" at the beginning of a text string. Your new tokenized text would help you detect the presence or absence of words such as "Hi" and "Hello" without getting confused by words like "Hiking" and "Hell." With your new tokenizer in place, your NLP pipeline wouldn't misinterpret the word "Hiking" as the greeting "Hi king":

```
>>> hi_text = 'Hiking home now'  
>>> hi_text.startswith('Hi')  
True  
>>> pattern = r'\w+(?:(\w+)?|[^\w\s])' #1  
>>> 'Hi' in re.findall(pattern, hi_text) #2  
False  
>>> 'Hi' == re.findall(pattern, hi_text)[0] #3  
False
```

So tokenization can help you reduce the number of false positives in your simple intent recognition pipeline that looks for the presence of greeting words. This is often called keyword detection, because your vocabulary of words is limited to a set of words you think are important. However, it's quite cumbersome to have to think of all the words that might appear in a greeting in order to recognize them all, including slang, misspellings and typos. And creating a for loop to iterate through them all would be inefficient. We can use the math of linear algebra and the vectorized operations of numpy to speed this process up.

In order to detect tokens efficiently you will want to use three new tricks:

1. matrix and vector representations of documents
2. vectorized operations in numpy
3. indexing of discrete vectors

You'll first learn the most basic, direct, raw and lossless way to represent words as a matrix, one-hot encoding.

2.6.1 One-hot Vectors

Now that you've successfully split your document into the kinds of words

you want, you're ready to create vectors out of them. Vectors of numbers are what we need to do the math or processing of NL*P* on natural language text.

```
>>> import pandas as pd
>>> onehot_vectors = np.zeros(
...     (len(tokens), vocab_size), int) #1
>>> for i, word in enumerate(tokens):
...     onehot_vectors[i, vocab.index(word)] = 1 #2
>>> df_onehot = pd.DataFrame(onehot_vectors, columns=vocab)
>>> df_onehot.shape
(18, 15)
>>> df_onehot.iloc[:, :8].replace(0, '') #3
   .  Survival  There's  adequate  as  fittest  maybe
0
1
2
3
4
5
6
7
8
9    1
10
11
12
13
14
15  1
16
17    1
```

In this representation of this two-sentence quote, each row is a vector representation of a single word from the text. The table has the 15 columns because this is the number of unique words in your vocabulary. The table has 18 rows, one for each word in the document. A "1" in a column indicates a vocabulary word that was present at that position in the document.

You can "read" a one-hot encoded (vectorized) text from top to bottom. You can tell that the first word in the text was the word "There's", because the 1 on the first row is positioned under the column label "There's". The next three rows (row indexes 1, 2, and 3) are blank, because we've truncated the table on the right to help it fit on the page. The fifth row of the text, with the

0-offset index number of 4 shows us that the fifth word in the text was the word "adequate", because there's a 1 in that column.

One-hot vectors are super-sparse, containing only one nonzero value in each row vector. For display, this code replaces the 0's with empty strings (''), to make it easier to read. But the code did not actually alter the DataFrame of data you are processing in your NLP pipeline. The Python code above was just to make it easier to read, so you can see that it looks a bit like a player piano paper roll, or maybe a music box drum.

The Pandas DataFrame made this output a little easier to read and interpret. The DataFrame .columns keep track of labels for each column. This allows you to label each column in your table with a string, such as the token or word it represents. A DataFrame can also keep track of labels for each row in an the DataFrame .index, for speedy lookup.



Important

Don't add strings to any DataFrame you intend to use in your machine learning pipeline. The purpose of a tokenizer and vectorizer, like this one-hot vectorizer, is to create a numerical array that your NLP pipeline can do math on. You can't do math on strings.

Each row of the table is a binary row vector, and you can see why it's also called a one-hot vector: all but one of the positions (columns) in a row are 0 or blank. Only one column, or position in the vector is "hot" ("1"). A one (1) means on, or hot. A zero (0) mean off, or absent.

One nice feature of this vector representation of words and tabular representation of documents is that no information is lost. The exact sequence of tokens is encoded in the order of the one-hot vectors in the table representing a document. As long as you keep track of which words are indicated by which column, you can reconstruct the original sequence of tokens from this table of one-hot vectors perfectly. And this reconstruction process is 100% accurate even though your tokenizer was only 90% accurate at generating the tokens you thought would be useful. As a result, one-hot word vectors like this are typically used in neural nets, sequence-to-sequence

language models, and generative language models. They are a good choice for any model or NLP pipeline that needs to retain all the meaning inherent in the original text.



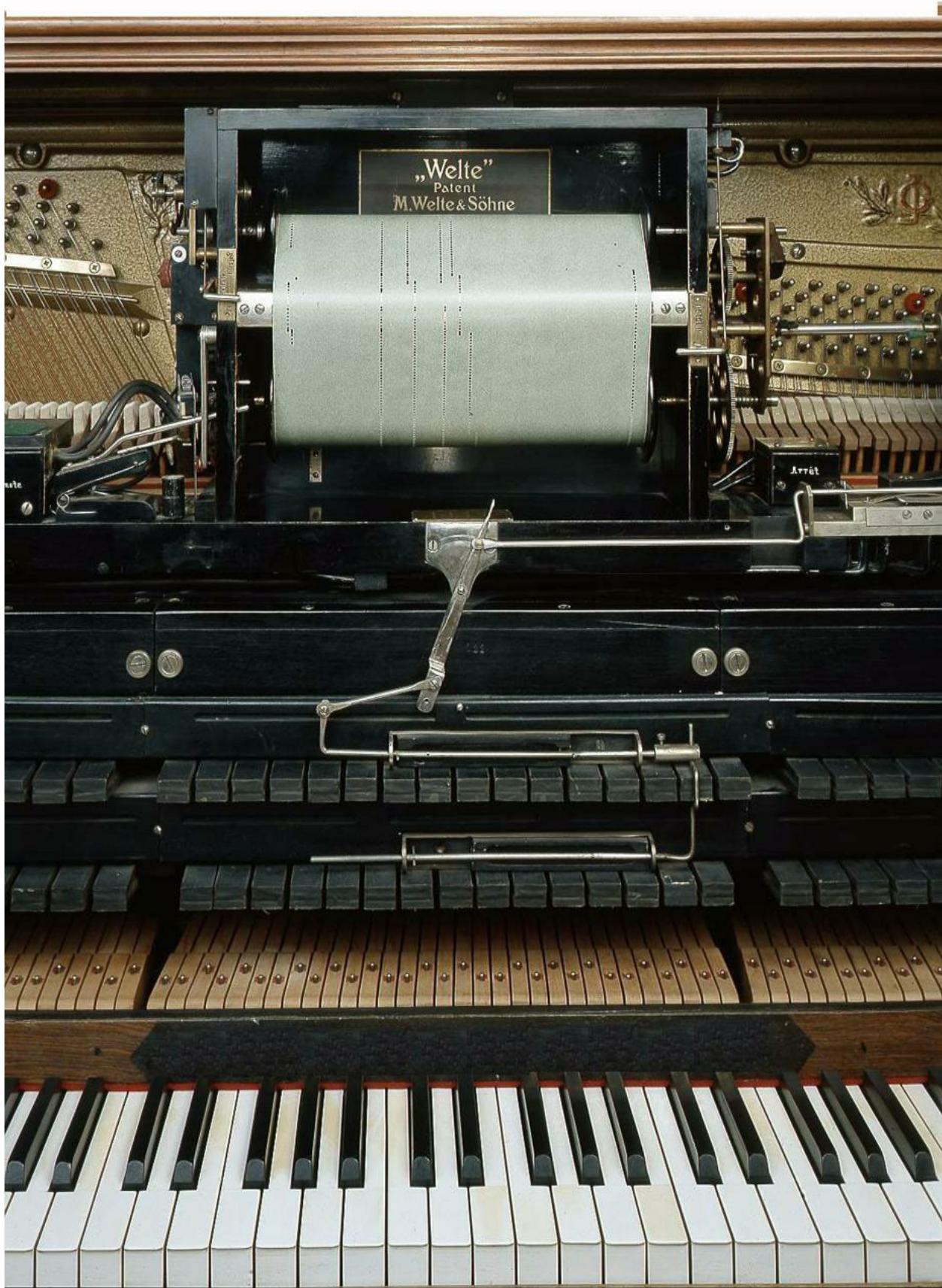
Tip

The one-hot encoder (vectorizer) did not discard any information from the text, but our tokenizer did. Our regular expression tokenizer discarded the whitespace characters (\s) that sometimes occur between words. So you could not perfectly reconstruct the original text with a *detokenizer*.

Tokenizers like spaCy, however, keep track of these whitespace characters and can in fact detokenize a sequence of tokens perfectly. SpaCy was named for this feature of accurately accounting for white-space efficiently and accurately.

This sequence of one-hot vectors is like a digital recording of the original text. If you squint hard enough you might be able to imagine that the matrix of ones and zeros above is a player piano roll.^[79] Or maybe it's the bumps on the metal drum of a music box.^[80] The vocabulary key at the top tells the machine which "note" or word to play for each row in the sequence of words or piano music.

Figure 2.2. Player piano roll



Unlike a player-piano or a music box, your mechanical word recorder and player is only allowed to use one "finger" at a time. It can only play one "note" or word at a time. It's one-hot. And there is no variation in the spacing of the words.

The important thing is that you've turned a sentence of natural language words into a sequence of numbers, or vectors. Now you can have the computer read and do math on the vectors just like any other vector or list of numbers. This allows your vectors to be input into any natural language processing pipeline that requires this kind of vector. The Deep Learning pipelines of chapter 5 through 10 typically require this representation, because they can be designed to extract "features" of meaning from these raw representations of text. And Deep Learning pipelines can generate text from numerical representations of meaning. So the stream of words emanating from your NLG pipelines in later chapters will often be represented by streams of one-hot encoded vectors, just like a player piano might play a song for a less artificial audience in West World.[\[81\]](#)

Now all you need to do is figure out how to build a "player piano" that can *understand* and combine those word vectors in new ways. Ultimately, you'd like your chatbot or NLP pipeline to play us a song, or say something, you haven't heard before. You'll get to do that in chapters 9 and 10 when you learn about recurrent neural networks that are effective for sequences of one-hot encoded tokens like this.

This representation of a sentence in one-hot word vectors retains all the detail, grammar, and order of the original sentence. And you have successfully turned words into numbers that a computer can "understand." They are also a particular kind of number that computers like a lot: binary numbers. But this is a big table for a short sentence. If you think about it, you have expanded the file size that would be required to store your document. For a long document this might not be practical.

How big is this **lossless** numerical representation of your collection of documents? Your vocabulary size (the length of the vectors) would get huge. The English language contains at least 20,000 common words, millions if you include names and other proper nouns. And your one-hot vector

representation requires a new table (matrix) for every document you want to process. This is almost like a raw "image" of your document. If you have done any image processing, you know that you need to do dimension reduction if you want to extract useful information from the data.

Let's run through the math to give you an appreciation for just how big and unwieldy these "piano rolls" are. In most cases, the vocabulary of tokens you'll use in an NLP pipeline will be much more than 10,000 or 20,000 tokens. Sometimes it can be hundreds of thousands or even millions of tokens. Let's assume you have a million tokens in your NLP pipeline vocabulary. And let's say you have a meager 3000 books with 3500 sentences each and 15 words per sentence—reasonable averages for short books. That's a whole lot of big tables (matrices), one for each book. That would use 157.5 terabytes. You probably couldn't even store that on disk.

That is more than a million million bytes, even if you are super-efficient and use only one byte for each number in your matrix. At one byte per cell, you would need nearly 20 terabytes of storage for a small bookshelf of books processed this way. Fortunately you do not ever use this data structure for storing documents. You only use it temporarily, in RAM, while you are processing documents one word at a time.

So storing all those zeros, and recording the order of the words in all your documents does not make much sense. It is not practical. And it's not very useful. Your data structure hasn't abstracted or generalized from the natural language text. An NLP pipeline like this doesn't yet do any real feature extraction or dimension reduction to help your machine learning work well in the real world.

What you really want to do is compress the meaning of a document down to its essence. You would like to compress your document down to a single vector rather than a big table. And you are willing to give up perfect "recall." You just want to capture most of the meaning (information) in a document, not all of it.

2.6.2 BOW (Bag-of-Words) Vectors

Is there any way to squeeze all those *player piano music rolls* into a single vector? Vectors are a great way to represent any object. With vectors we could compare documents to each other just by checking the Euclidian distance between them. Vectors allow us to use all your linear algebra tools on natural language. And that's really the goal of NLP, doing math on text.

Let us assume you can ignore the order of the words in our texts. For this first cut at a vector representation of text you can just jumble them all up together into a "bag," one bag for each sentence or short document. It turns out just knowing what words are present in a document can give your NLU pipeline a lot of information about what's in it. This is in fact the representation that powers big Internet search engine companies. Even for documents several pages long, a bag-of-words vector is useful for summarizing the essence of a document.

Let's see what happens when we jumble and count the words in our text from *The Book Thief*:

```
>>> bow = sorted(set(re.findall(pattern, text)))
>>> bow[:9]
[',', '.', 'Liesel', 'Trust', 'and', 'arrived', 'clouds', 'hands'
>>> bow[9:19]
['hold', 'in', 'like', 'me', 'on', 'out', 'rain', 'she', 'the', '
>>> bow[19:27]
['them', 'they', 'though', 'way', 'were', 'when', 'words', 'would
```

Even with this jumbled up bag of words, you can get a general sense that this sentence is about: "Trust", "words", "clouds", "rain", and someone named "Liesel". One thing you might notice is that Python's `sorted()` puts punctuation before characters, and capitalized words before lowercase words. This is the ordering of characters in the ASCII and Unicode character sets. However, the order of your vocabulary is unimportant. As long as you are consistent across all the documents you tokenize this way, a machine learning pipeline will work equally well with any vocabulary order.

You can use this new bag-of-words vector approach to compress the information content for each document into a data structure that is easier to work with. For keyword search, you could **OR** your one-hot word vectors from the player piano roll representation into a binary bag-of-words vector.

In the play piano analogy this is like playing several notes of a melody all at once, to create a "chord". Rather than "replaying" them one at a time in your NLU pipeline, you would create a single bag-of-words vector for each document.

You could use this single vector to represent the whole document in a single vector. Because vectors all need to be the same length, your BOW vector would need to be as long your vocabulary size which is the number of unique tokens in your documents. And you could ignore a lot of words that would not be interesting as search terms or keywords. This is why stop words are often ignored when doing BOW tokenization. This is an extremely efficient representation for a search engine index or the first filter for an information retrieval system. Search indexes only need to know the presence or absence of each word in each document to help you find those documents later.

This approach turns out to be critical to helping a machine "understand" a collection of words as a single mathematical object. And if you limit your tokens to the 10,000 most important words, you can compress your numerical representation of your imaginary 3500 sentence book down to 10 kilobytes, or about 30 megabytes for your imaginary 3000-book corpus. One-hot vector sequences for such a modest-sized corpus would require hundreds of gigabytes.

Another advantage of the BOW representation of text is that it allows you to find similar documents in your corpus in constant time ($O(1)$). You can't get any faster than this. BOW vectors are the precursor to a reverse index which is what makes this speed possible. In computer science and software engineering, you are always on the lookout for data structures that enable this kind of speed. All major full text search tools use BOW vectors to find what you're looking for fast. You can see this numerical representation of natural language in Elasticsearch, Solr, [\[82\]](#) PostgreSQL, and even state of the art web search engines such as Qwant, [\[83\]](#), SearX, [\[84\]](#), and Wolfram Alpha [\[85\]](#).

Fortunately, the words in your vocabulary are sparsely utilized in any given text. And for most bag-of-words applications, we keep the documents short, sometimes just a sentence will do. So rather than hitting all the notes on a piano at once, your bag-of-words vector is more like a broad and pleasant

piano chord, a combination of notes (words) that work well together and contain meaning. Your NLG pipeline or chatbot can handle these chords even if there is a lot of "dissonance" from words in the same statement that are not normally used together. Even dissonance (odd word usage) is useful information about a statement that a machine learning pipeline can make use of.

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

Sparse representations

You might be thinking that if you process a huge corpus you'll probably end up with thousands or even millions of unique tokens in your vocabulary. This would mean you would have to store a lot of zeros in your vector representation of our 20-token sentence about Liesel. A dict would use much less memory than a vector. Any paired mapping of words to their 0/1 values would be more efficient than a vector. But you can't do math on `dict's. So this is why CountVectorizer uses a sparse numpy array to hold the counts of words in a word frequency vector. Using a dictionary or sparse array for your vector ensures that it only has to store a 1 when any one of the millions of possible words in your dictionary appear in a particular document.

But if you want to look at an individual vector to make sure everything is working correctly, a Pandas Series is the way to go. And you will wrap that up in a Pandas DataFrame so you can add more sentences to your binary vector "corpus" of quotes.

2.6.3 Dot product

You'll use the dot product a lot in NLP, so make sure you understand what it is. Skip this section if you can already do dot products in your head.

The dot product is also called the *inner product* because the "inner" dimension of the two vectors (the number of elements in each vector) or matrices (the rows of the first matrix and the columns of the second matrix) must be the same because that is where the products happen. This is analogous to an "inner join" on two relational database tables.

The dot product is also called the *scalar product* because it produces a single scalar value as its output. This helps distinguish it from the *cross product*, which produces a vector as its output. Obviously, these names reflect the shape of the symbols used to indicate the dot product (\cdot) and cross product (\times) in formal mathematical notation. The scalar value output by the scalar product can be calculated by multiplying all the elements of one vector by all the elements of a second vector and then adding up those normal multiplication products.

Here is a Python snippet you can run in your Pythonic head to make sure you understand what a dot product is:

Listing 2.2. Example dot product calculation

```
>>> v1 = pd.np.array([1, 2, 3])
>>> v2 = pd.np.array([2, 3, 4])
>>> v1.dot(v2)
20
>>> (v1 * v2).sum() #1
20
>>> sum([x1 * x2 for x1, x2 in zip(v1, v2)]) #2
20
```



Tip

The dot product is equivalent to the *matrix product*, which can be accomplished in NumPy with the `np.matmul()` function or the `@` operator. Since all vectors can be turned into Nx1 or 1xN matrices, you can use this shorthand operator on two column vectors (Nx1) by transposing the first one so their inner dimensions line up, like this: `v1.reshape(-1, 1.T @`

```
v2.reshape(1, 1, which outputs your scalar product within a 1x1 matrix:  
array([[20]])
```

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

NLTK and Stanford CoreNLP have been around the longest and are the most widely used for comparison of NLP algorithms in academic papers. Even though the Stanford CoreNLP has a Python API, it relies on the Java 8 CoreNLP backend, which must be installed and configured separately. So if you want to publish the results of your work in an academic paper and compare it to what other researchers are doing, you may need to use NLTK. The most common tokenizer used in academia is the PennTreebank tokenizer:

```
>>> from nltk.tokenize import TreebankWordTokenizer  
>>> texts.append(  
...     "If conscience and empathy were impediments to the advancement  
...     "self-interest, then we would have evolved to be amoral sociopaths.  
... ) #1  
>>> tokenizer = TreebankWordTokenizer()  
>>> tokens = tokenizer.tokenize(texts[-1])[:6]  
>>> tokens[:8]  
['If', 'conscience', 'and', 'empathy', 'were', 'impediments', 'to',  
>>> tokens[8:16]  
['advancement', 'of', 'self-interest', ',', 'then', 'we', 'would'  
>>> tokens[16:]  
['evolved', 'to', 'be', 'amoral', 'sociopaths', '.']
```

The spaCy Python library contains a natural language processing pipeline that includes a tokenizer. In fact, the name of the package comes from the words "space" and "Cython". SpaCy was built using the Cython package to

speed the tokenization of text, often using the **space** character (" ") as the delimiter. SpaCy has become the **multitool** of NLP, because of its versatility and the elegance of its API. To use spaCy, you can start by creating an callable parser object, typically named `nlp`. You can customize your NLP pipeline by modifying the Pipeline elements within that parser object.

And spaCy has "batteries included." So even with the default smallest spaCy language model loaded, you can do tokenization and sentence segmentation, plus **part-of-speech** and **abstract-syntax-tree** tagging—all with a single function call. When you call `nlp()` on a string, spaCy tokenizes the text and returns a `Doc` (document) object. A `Doc` object is a container for the sequence of sentences and tokens that it found in the text.

The spaCy package tags each token with their linguistic function to provide you with information about the text's grammatical structure. Each token object within a `Doc` object has attributes that provide these tags.

For example: * `token.text` the original text of the word * `token.pos_` grammatical part of speech tag as a human-readable string * `token.pos` integer for the grammar part of speech tag * `token.dep_` indicates the tokens role in the syntactic dependency tree * `token.dep` integer corresponding to the syntactic dependency tree location

The `.text` attribute provides the original text for the token. This is what is provided when you request the `str` representation of a token. A spaCy `Doc` object is allowing you to detokenize a document object to recreate the entire input text. i.e., the relation between tokens. You can use these functions to examine the text in more depth.

```
>>> import spacy
>>> nlp = spacy.load("en_core_web_sm")
>>> text = "Nice guys finish first." #1
>>> doc = nlp(text)
>>> for token in doc:
>>>     print(f"{token.text}:<11>{token.pos_}:<10>{token.dep:<10}")
Nice          ADJ      amod
guys         NOUN     nsubj
finish        VERB     ROOT
first         ADV      advmod
```

.

| | |
|-------|-------|
| PUNCT | punct |
|-------|-------|

[79] See the "Player piano" article on Wikipedia
(https://en.wikipedia.org/wiki/Player_piano).

[80] See the web page titled "Music box - Wikipedia"
(https://en.wikipedia.org/wiki/Music_box).

[81] West World is a television series about particularly malevolent humans and human-like robots, including one that plays a piano in the main bar.

[82] Apache Solr home page and Java source code (<https://solr.apache.org/>)

[83] Qwant web search engine based in Europe (<https://www.qwant.com/>)

[84] SearX git repository (<https://github.com/searx/searx>) and web search (<https://searx.thegpm.org/>)

[85] (<https://www.wolframalpha.com/>)

[86] excerpt from Martin A. Nowak and Roger Highfield in *SuperCooperators: Altruism, Evolution, and Why We Need Each Other to Succeed*. New York: Free Press, 2011.

[87] excerpt from Martin A. Nowak and Roger Highfield *SuperCooperators: Altruism, Evolution, and Why We Need Each Other to Succeed*. New York: Free Press, 2011.

2.7 Challenging tokens

Chinese, Japanese, and other pictograph languages aren't limited to a small number of letters in alphabets used to compose tokens or words. Characters in these traditional languages look more like drawings and are called "pictographs." There are many thousands of unique characters in the Chinese and Japanese languages. And these characters are used much like we use words in alphabet-based languages such as English. But each Chinese character is usually not a complete word on its own. A character's meaning

depends on the characters to either side. And words are not delimited with spaces. This makes it challenging to tokenize Chinese text into words or other packets of thought and meaning.

The `jieba` package is a Python package you can use to segment traditional Chinese text into words. It supports three segmentation modes: 1) "full mode" for retrieving all possible words from a sentence, 2) "accurate mode" for cutting the sentence into the most accurate segments, 3) "search engine mode" for splitting long words into shorter ones, sort-of like splitting compound words or finding the roots of words in English. In the example below, the Chinese sentence "西安是一座举世闻名的文化古城" translates into "Xi'an is a city famous world-wide for its ancient culture." Or, a more compact and literal translation might be "Xi'an is a world-famous city for her ancient culture."

From a grammatical perspective, you can split the sentence into: 西安 (Xi'an), 是 (is), 一座 (a), 举世闻名 (world-famous), 的 (adjective suffix), 文化 (culture), 古城 (ancient city). The character "座" is the quantifier meaning "ancient" that is normally used to modify the word "city." The accurate mode in `jieba` causes it to segment the sentence this way so that you can correctly extract a precise interpretation of the text.

Listing 2.3. Jieba in accurate mode

```
>>> seg_list = jieba.cut("西安是一座举世闻名的文化古城") #1
>>> list(seg_list)
['西安', '是', '一座', '举世闻名', '的', '文化', '古城']
```

Jieba's accurate mode minimizes the total number of tokens or words. This gave you 7 tokens for this short Jieba attempts to keep as many possible characters together. This will reduce the false positive rate or type 1 errors for detecting boundaries between words.

In full mode, `jieba` will attempt to split the text into smaller words, and more of them.

Listing 2.4. Jieba in full mode

```
>>> import jieba
```

```
... seg_list = jieba.cut("西安是一座举世闻名的文化古城", cut_all=True)
>>> list(seg_list)
['西安', '是', '一座', '举世', '举世闻名', '闻名', '的', '文化', '古城']
```

Now you can try search engine mode to see if it's possible to break up these tokens even further:

Listing 2.5. Jieba in search engine mode

```
>>> seg_list = jieba.cut_for_search("西安是一座举世闻名的文化古城") #
>>> list(seg_list)
['西安', '是', '一座', '举世', '闻名', '举世闻名', '的', '文化', '古城']
```

Unfortunately later versions of Python (3.5+) aren't supported by Jieba's part-of-speech tagging model.

```
>>> import jieba
>>> from jieba import posseg
>>> words = posseg.cut("西安是一座举世闻名的文化古城")
>>> jieba.enable_paddle() #1
>>> words = posseg.cut("西安是一座举世闻名的文化古城", use_paddle=True
>>> list(words)
[pair('西安', 'ns'),
 pair('是', 'v'),
 pair('一座', 'm'),
 pair('举世闻名', 'i'),
 pair('的', 'uj'),
 pair('文化', 'n'),
 pair('古城', 'ns')]
```

You can find more information about jieba at (<https://github.com/fxsjy/jieba>). SpaCy also contains Chinese language models that do a decent job of segmenting and tagging Chinese text.

```
>>> import spacy
>>> spacy.cli.download("zh_core_web_sm") #1
>>> nlpzh = spacy.load("zh_core_web_sm")
>>> doc = nlpzh("西安是一座举世闻名的文化古城")
>>> [(tok.text, tok.pos_) for tok in doc]
[('西安', 'PROPN'),
 ('是', 'VERB'),
 ('一', 'NUM'),
 ('座', 'NUM'),
 ('举世闻名', 'VERB'),
```

```
('的', 'PART'),  
('文化', 'NOUN'),  
('古城', 'NOUN'))]
```

As you may notice, spaCy provides slightly different tokenization and tagging, which is more attached to the original meaning of each word rather than the context of this sentence.

2.7.1 A complicated picture

Unlike English, there is no concept of stemming or lemmatization in pictographic languages such as Chinese and Japanese (Kanji). However, there's a related concept. The most essential building blocks of Chinese characters are called *radicals*. To better understand *radicals*, you must first see how Chinese characters are constructed. There are six types of Chinese characters: 1) pictographs, 2) pictophonetic characters, 3) associative compounds, 4) self-explanatory characters, 5) phonetic loan characters, and 6) mutually explanatory characters. The top four categories are the most important and encompass most Chinese characters.

1. Pictographs (象形字)
2. Pictophonetic characters (形声字)
3. Associative compounds (会意字)

1. Pictographs (象形字)

Pictographs were created from images of real objects, such as the characters for 口 (mouth) and 门 (door).

2. Pictophonetic characters (形声字)

Pictophonetic characters were created from a radical and a single Chinese character. One part represents its meaning and the other indicates its pronunciation. For example, 妈 (mā, mother) = 女 (female) + 马 (mǎ, horse). Squeezing 女 into 马 gives 妈. The character 女 is the semantic radical that indicates the meaning of the character (female). 马 is a single character that has a similar pronunciation (mǎ). You can see that the character for mother

(妈) is a combination of the characters for female and This is comparable to the English concept of homophones—words that sound alike but mean completely different things. But in Chinese use additional characters to disambiguate homophones. The character for female

3. Associative compounds (会意字)

Associative compounds can be divided into two parts: one symbolizes the image, the other indicates the meaning.

For example, 旦 (dawn), the upper part (日) is the sun and the lower part (一) is like the horizon line.

Self-explanatory characters (指事字)

Self-explanatory characters cannot be easily represented by an image, so they are shown by a single abstract symbol. For example, 上 (up), 下 (down).

As you can see, procedures like stemming and lemmatization are harder or impossible for many Chinese characters. Separating the parts of a character may radically ; change its meaning. And there's not prescribed order or rule for combining radicals to create Chinese characters.

Nonetheless, some kinds of stemming are harder in English than they are in Chinese. For example, automatically removing the pluralization from words like "we", "us", "they" and "them" is hard in English but straightforward in Chinese. Chinese uses inflection to construct the plural form of characters, similar to adding s to the end of English words. In Chinese the pluralization suffix character is 们. The character 朋友 (friend) becomes 朋友们 (friends).

Even the characters for "we/us", "they/them", and "y'all" use the same pluralization suffix: 我们 (we/us), 他们 (they/them), 你们 (you). But in English, you can remove the 'ing' or 'ed' from many verbs to get the root word. However, in Chinese, verb conjugation uses an additional character in the front or the end to indicate tense. There's no prescribed rule for verb conjugation. For example, examine the character 学 (learn), 在学 (learning), and 学过 (learned). In Chinese, you can also use a suffix 学 to denote an

academic discipline, such as 心理学 (psychology) or 社会学 (sociology). In most cases, you want to keep the integrated Chinese character together rather than reducing it to its components.

It turns out this is a good rule of thumb for all languages. Let the data do the talking. Do not stem or lemmatize unless the statistics indicate that it will help your NLP pipeline perform better. Is there not a small amount of meaning that is lost when "smarter" and "smartest" reduce to "smart"? Make sure stemming does not leave your NLP pipeline dumb.

Let the statistics of how of how characters and words are used together help you decide how, or if, to decompose any particular word or n-gram. In the next chapter we'll show you some tools like Scikit-Learn's `TfidfVectorizer` that handle all the tedious account required to get this right.

Contractions

You might be wondering why you would want to split the contraction `wasn't` into `was` and `n't`. For some applications, like grammar-based NLP models that use 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.



Tokenize informal text from social networks such as Twitter and Facebook

The NLTK library includes a rule-based tokenizer to deal with short, informal, emoji-laced texts from social networks: `casual_tokenize`

It handles emojis, emoticons, and usernames. The `reduce_1en` option deletes less meaningful character repetitions. The `reduce_1en` algorithm retains three repetitions, to approximate the intent and sentiment of the original text.

```
>>> from nltk.tokenize.casual import casual_tokenize
```

```
>>> texts.append("@rickrau mind BL00000000WWWN by latest lex :*  
>>> casual_tokenize(texts[-1], reduce_len=True)  
['@rickrau', 'mind', 'BL000WWN', 'by', 'latest', 'lex', ':*'), '
```

2.7.2 Extending your vocabulary with *n*-grams

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

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

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

We all gram for *n*-grams

An *n*-gram is a sequence containing up to *n* elements that have been extracted from a sequence of those elements, usually a string. In general the "elements" of an *n*-gram can be characters, syllables, words, or even symbols like "A", "D", and "G" used to represent the chemical amino acid markers in a DNA or RNA sequence.[\[88\]](#)

In this book, we're only interested in *n*-grams of words, not characters.[\[89\]](#) So in this book, when we say 2-gram, we mean a pair of words, like "ice cream". When we say 3-gram, we mean a triplet of words like "beyond the pale" or "Johann Sebastian Bach" or "riddle me this". *n*-grams do not have to mean something special together, like compound words. They have to be frequent enough together to catch the attention of your token counters.

Why bother with *n*-grams? As you saw earlier, when a sequence of tokens is vectorized into a bag-of-words vector, it loses a lot of the meaning inherent in the order of those words. By extending your concept of a token to include multiword tokens, *n*-grams, your NLP pipeline can retain much of the meaning inherent in the order of words in your statements. For example, the meaning-inverting word "not" will remain attached to its neighboring words, where it belongs. Without *n*-gram tokenization, it would be free floating. Its meaning would be associated with the entire sentence or document rather

than its neighboring words. The 2-gram "was not" retains much more of the meaning of the individual words "not" and "was" than those 1-grams alone in a bag-of-words vector. A bit of the context of a word is retained when you tie it to its neighbor(s) in your pipeline.

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

Stop words

Stop words are common words in any language that occur with a high frequency but carry much less substantive information about the meaning of a phrase. Examples of some common stop words include [\[90\]](#)

- a, an
- the, this
- and, or
- of, on

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

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

In your NLP pipeline, you might create 4-grams such as reported to the

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

Designing a filter for stop words depends on your particular application. Vocabulary size will drive the computational complexity and memory requirements of all subsequent steps in the NLP pipeline. But stop words are only a small portion of your total vocabulary size. A typical stop word list has only 100 or so frequent and unimportant words listed in it. But a vocabulary size of 20,000 words would be required to keep track of 95% of the words seen in a large corpus of tweets, blog posts, and news articles.^[91] And that is just for 1-grams or single-word tokens. A 2-gram vocabulary designed to catch 95% of the 2-grams in a large English corpus will generally have more than 1 million unique 2-gram tokens in it.

You may be worried that vocabulary size drives the required size of any training set you must acquire to avoid overfitting to any particular word or combination of words. And you know that the size of your training set drives the amount of processing required to process it all. However, getting rid of 100 stop words out of 20,000 is not going to significantly speed up your work. And for a 2-gram vocabulary, the savings you would achieve by removing stop words is minuscule. In addition, for 2-grams you lose a lot more information when you get rid of stop words arbitrarily, without checking for the frequency of the 2-grams that use those stop words in your text. For example, you might miss mentions of "The Shining" as a unique title and instead treat texts about that violent, disturbing movie the same as you treat documents that mention "Shining Light" or "shoe shining".

So if you have sufficient memory and processing bandwidth to run all the NLP steps in your pipeline on the larger vocabulary, you probably do not

want to worry about ignoring a few unimportant words here and there. And if you are worried about overfitting a small training set with a large vocabulary, there are better ways to select your vocabulary or reduce your dimensionality than ignoring stop words. Including stop words in your vocabulary allows the document frequency filters (discussed in chapter 3) to more accurately identify and ignore the words and n -grams with the least information content within your particular domain.

Several websites and python packages include predefined sets of stop words specialized for various use cases.^[92] Here we use an exhaustive list so you can get a feel for the amount of meaning that can be lost if your hand-crafted list of stop words isn't well-designed.

```
>>> import requests
>>> url = ("https://gitlab.com/tangibleai/nlpia/-/raw/master/"
...         "src/nlpia/data/stopword_lists.json")
>>> response = requests.get(url)
>>> stop_words = response.json()['exhaustive'] #1
>>> tokens = 'the words were just as I remembered them'.split()
>>> tokens_without_stopwords = [x for x in tokens if x not in stop_words]
>>> print(tokens_without_stopwords)
['I', 'remembered']
```

You can see that some words carry more meaning than others. This is a sentence from a short story by Ted Chiang about machines helping us remember our statements so we don't have to rely on flawed memories.^[93] In this phrase you lost two thirds of the words and still retained the bulk of the phrase's meaning. However you can see that an import token "words" was discarded by this particular stop words list. 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 choose to always skip? That is how stop words are chosen.

Here's another common stop words list that isn't quite as exhaustive:

Listing 2.6. NLTK list of stop words

```
>>> import nltk
>>> nltk.download('stopwords')
```

```
>>> stop_words = nltk.corpus.stopwords.words('english')
>>> len(stop_words)
179
>>> stop_words[:7]
['i', 'me', 'my', 'myself', 'we', 'our', 'ours']
>>> [sw for sw in stopwords if len(sw) == 1]
['i', 'a', 's', 't', 'd', 'm', 'o', 'y']
```

A document that dwells on the first person is pretty boring, and more importantly for you, has low information content. The NLTK package includes pronouns (not just first person ones) in its list of stop words. And these one-letter stop words are even more curious, but they make sense if you have used the NLTK tokenizer and Porter stemmer a lot. These single-letter tokens pop up a lot when contractions are split and stemmed using NLTK tokenizers and stemmers.



Warning

The set of English stop words in sklearn, spacy, nltk, and SEO tools are very different, and they are constantly evolving. At the time of this writing, sklearn has 318 stop words, NLTK has 179 stop words, spaCy has 326, and our 'exhaustive' SEO list includes 667 stop words.

This is a good reason to consider **not** filtering stop words. If you do, others may not be able to reproduce your results.

Depending on how much natural language information you want to discard ;), you can take the union or the intersection of multiple stop word lists for your pipeline. Here are some stop_words lists we found, though we rarely use any of them in production:

Listing 2.7. Collection of stop words lists

```
>>> resp = requests.get(url)
>>> len(resp.json()['exhaustive'])
667
>>> len(resp.json()['sklearn'])
318
>>> len(resp.json()['spacy'])
326
```

```
>>> len(resp.json()['nltk'])
179
>>> len(resp.json()['reuters'])
28
```

2.7.3 Normalizing your vocabulary

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

Case folding

Case folding is when you consolidate multiple "spellings" of a word that differ only in their capitalization. So why would we use case folding at all? Words can become case "denormalized" when they are capitalized because of their presence at the beginning of a sentence, or when they're written in ALL CAPS for emphasis. Undoing this denormalization is called *case normalization*, or more commonly, *case folding*. Normalizing word and character capitalization is one way to reduce your vocabulary size and generalize your NLP pipeline. It helps you consolidate words that are intended to mean (and be spelled) the same thing under a single token.

However, some information is often communicated by capitalization of a word—for example, 'doctor' and 'Doctor' often have different meanings. Often capitalization is used to indicate that a word is a proper noun, the name of a person, place, or thing. You will want to be able to recognize proper nouns as distinct from other words, if named entity recognition is important to your pipeline. However, if tokens are not case normalized, your vocabulary will be approximately 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 dataset used for training must be "representative" of the space of all possible feature vectors your model must deal with, including variations in capitalization. For 100000-D bag-of-words vectors, you usually must have 100000 labeled examples, and sometimes even more than that, to train a supervised machine learning pipeline without overfitting. In some situations, cutting your vocabulary size by half can sometimes be worth the loss of information content.

In Python, you can easily normalize the capitalization of your tokens with a list comprehension.

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

And if you are certain that you want to normalize the case for an entire document, you can `lower()` the text string in one operation, before tokenization. But this will prevent advanced tokenizers that can split *camel case* words like "WordPerfect", "FedEx", or "stringVariableName."^[94] Maybe you want WordPerfect to be its own unique thing (token), or maybe you want to reminisce about a more perfect word processing era. It is up to you to decide when and how to apply case folding.

With case normalization, you are attempting to return these tokens to their "normal" state before grammar rules and their position in a sentence affected their capitalization. The simplest and most common way to normalize the case of a text string is to lowercase all the characters with a function like Python's built-in `str.lower()`.^[95] Unfortunately this approach will also "normalize" away a lot of meaningful capitalization in addition to the less meaningful first-word-in-sentence capitalization you intended to normalize away. A better approach for case normalization is to lowercase only the first word of a sentence and allow all other words to retain their capitalization.

Lowercasing on the first word in a sentence preserves the meaning of a proper nouns in the middle of a sentence, like "Joe" and "Smith" in "Joe Smith". And it properly groups words together that belong together, because they are only capitalized when they are at the beginning of a sentence, since

they are not proper nouns. This prevents "Joe" from being confused with "coffee" ("joe")^[96] during tokenization. And this approach prevents the blacksmith connotation of "smith" being confused with the proper name "Smith" in a sentence like "A word smith had a cup of joe." Even with this careful approach to case normalization, where you lowercase words only at the start of a sentence, you will still need to introduce capitalization errors for the rare proper nouns that start a sentence. "Joe Smith, the word smith, with a cup of joe." will produce a different set of tokens than "Smith the word with a cup of joe, Joe Smith." And you may not want that. In addition, case normalization is useless for languages that do not have a concept of capitalization, like Arabic or Hindi.

To avoid this potential loss of information, many NLP pipelines do not normalize for case at all. For many applications, the efficiency gain (in storage and processing) for reducing one's vocabulary size by about half is outweighed by the loss of information for proper nouns. But some information may be "lost" even without case normalization. If you do not identify the word "The" at the start of a sentence as a stop word, that can be a problem for some applications. Really sophisticated pipelines will detect proper nouns before selectively normalizing the case for words at the beginning of sentences that are clearly not proper nouns. You should implement whatever case normalization approach makes sense for your application. If you do not have a lot of "Smith"s and "word smiths" in your corpus, and you do not care if they get assigned to the same tokens, you can just lowercase everything. The best way to find out what works is to try several different approaches, and see which approach gives you the best performance for the objectives of your NLP project.

By generalizing your model to work with text that has odd capitalization, case normalization can reduce overfitting for your machine learning pipeline. Case normalization is particularly useful for a search engine. For search, normalization increases the number of matches found for a particular query. This is often called the "recall" performance metric for a search engine (or any other classification model).^[97]

For a search engine without normalization if you searched for "Age" you will get a different set of documents than if you searched for "age." "Age" would

likely occur in phrases like "New Age" or "Age of Reason". In contrast, "age" would be more likely to occur in phrases like "at the age of" in your sentence about Thomas Jefferson. By normalizing the vocabulary in your search index (as well as the query), you can ensure that both kinds of documents about "age" are returned regardless of the capitalization in the query from the user.

However, this additional recall accuracy comes at the cost of precision, returning many documents that the user may not be interested in. Because of this issue, modern search engines allow users to turn off normalization with each query, typically by quoting those words for which they want only exact matches returned. If you are building such a search engine pipeline, in order to accommodate both types of queries you will have to build two indexes for your documents: one with case-normalized *n*-grams, and another with the original capitalization.

Stemming

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

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

dataset size by half or even more, depending on the aggressiveness of the stemmer you chose.

Stemming is important for keyword search or information retrieval. It allows you to search for "developing houses in Portland" and get web pages or documents that use both the word "house" and "houses" and even the word "housing" because these words are all stemmed to the "hous" token. Likewise you might receive pages with the words "developer" and "development" rather than "developing" because all these words typically reduce to the stem "develop". As you can see, this is a "broadening" of your search, ensuring that you are less likely to miss a relevant document or web page. This broadening of your search results would be a big improvement in the "recall" score for how well your search engine is doing its job at returning all the relevant documents. [\[98\]](#)

But stemming could greatly reduce the "precision" score for your search engine because it might return many more irrelevant documents along with the relevant ones. In some applications this "false-positive rate" (proportion of the pages returned that you do not 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. Quoting indicates that you only want pages containing the exact spelling of a phrase such as "'Portland Housing Development software'." That would return a different sort of document than one that talks about a "'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 might be the effective query if you used a stemmer on that query.

Here's a simple 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 preceding stemmer function follows a few simple rules within that one short regular expression:

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

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

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

It also does not handle the "housing" example from your "Portland Housing" search.

Two of the most popular stemming algorithms are the Porter and Snowball stemmers. The Porter stemmer is named for the computer scientist Martin Porter.^[99] Porter is also responsible for enhancing the Porter stemmer to create the Snowball stemmer.^[100] Porter dedicated much of his lengthy career to documenting and improving stemmers, due to their value in information retrieval (keyword search). These stemmers implement more complex rules than our simple regular expression. This enables the stemmer to handle the complexities of English spelling and word ending rules.

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

Notice that the Porter stemmer, like the regular expression stemmer, retains the trailing apostrophe (unless you explicitly strip it), which ensures that possessive words will be distinguishable from nonpossessive words.

Possessive words are often proper nouns, so this feature can be important for applications where you want to treat names differently than other nouns.

More on the Porter stemmer

Julia Menchavez has graciously shared her translation of Porter's original stemmer algorithm into pure python (<https://github.com/jedijulia/porter-stemmer/blob/master/stemmer.py>). If you are ever tempted to develop your own stemmer, consider these 300 lines of code and the lifetime of refinement that Porter put into them.

There are eight steps to the Porter stemmer algorithm: 1a, 1b, 1c, 2, 3, 4, 5a, and 5b. Step 1a is a bit like your regular expression for dealing with trailing "S"es:[\[101\]](#)

```
def step1a(self, word):
    if word.endswith('sses'):
        word = self.replace(word, 'sses', 'ss') #1
    elif word.endswith('ies'):
        word = self.replace(word, 'ies', 'i')
    elif word.endswith('ss'):
        word = self.replace(word, 'ss', 'ss')
    elif word.endswith('s'):
        word = self.replace(word, 's', '')
    return word
```

The remaining seven steps are much more complicated because they have to deal with the complicated English spelling rules for the following:

- **Step 1a:** "s" and "es" endings
- **Step 1b:** "ed", "ing", and "at" endings
- **Step 1c:** "y" endings
- **Step 2:** "nounifying" endings such as "ational", "tional", "ence", and "able"
- **Step 3:** adjective endings such as "icate",[\[102\]](#), "ful", and "alize"
- **Step 4:** adjective and noun endings such as "ive", "ible", "ent", and "ism"
- **Step 5a:** stubborn "e" endings, still hanging around
- **Step 5b:** trailing double-consonants for which the stem will end in a single "l"

Snowball stemmer is more aggressive than the Porter stemmer. Notice that it stems 'fairly' to 'fair', which is more accurate than the Porter stemmer.

```
>>> from nltk.stem.snowball import SnowballStemmer  
>>> stemmer = SnowballStemmer(language='english')  
>>> ' '.join([stemmer.stem(w).strip("") for w in  
...     "dish washer's fairly washed dishes".split()])  
'dish washer fair wash dish'
```

Lemmatization

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

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

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

Sometimes lemmatization will be a better way to normalize the words in your

vocabulary. You may find that for your application stemming and case folding create stems and tokens that do not take into account a word's meaning. A lemmatizer uses a knowledge base of word synonyms and word endings to ensure that only words that mean similar things are consolidated into a single token.

Some lemmatizers use the word's part of speech (POS) tag in addition to its spelling to help improve accuracy. 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 "people, places, or things" within a phrase. An adjective POS 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. So some advanced lemmatizers cannot be run on words in isolation.

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".

So lemmatizers are better than stemmers for most applications. Stemmers are only really used in large scale information retrieval applications (keyword search). And if you really want the dimension reduction and recall improvement of a stemmer in your information retrieval pipeline, you should probably also use a lemmatizer right before the stemmer. Because the lemma of a word is a valid English word, stemmers work well on the output of a lemmatizer. This trick will reduce your dimensionality and increase your information retrieval recall even more than a stemmer alone. [\[103\]](#)

How can you identify word lemmas in Python? The NLTK package provides functions for this. Notice that you must tell the WordNetLemmatizer which part of speech you are interested in, if you want to find the most accurate lemma:

```
>>> nltk.download('wordnet')
True
```

```
>>> nltk.download('omw-1.4')
True
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize("better") #1
'better'
>>> lemmatizer.lemmatize("better", pos="a") #2
'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" did not change it at all. This is because the part of speech of a word can have a big effect on its meaning. If a POS is not specified for a word, then the NLTK lemmatizer assumes it is a noun. Once you specify the correct POS, 'a' for adjective, the lemmatizer returns the correct lemma. Unfortunately, the NLTK lemmatizer is restricted to the connections within the Princeton WordNet graph of word meanings. So the word "best" does not lemmatize to the same root as "better". This graph is also missing the connection between "goodness" and "good". A Porter stemmer, on the other hand, would make this connection by blindly stripping off the "ness" ending of all words.

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

You can easily implement lemmatization in spaCy by the following:

```
>>> import spacy
>>> nlp = spacy.load("en_core_web_sm")
>>> doc = nlp("better good goods goodness best")
>>> for token in doc:
>>> print(token.text, token.lemma_)
better well
good good
goods good
goodness goodness
```

best good

Unlike NLTK, spaCy lemmatizes "better" to "well" by assuming it is an adverb and returns the correct lemma for "best" ("good").

Synonym substitution

There are five kinds of "synonyms" that are sometime helpful in creating a consistent smaller vocabulary to help your NLP pipeline generalize well.

1. Typo correction
2. Spelling correction
3. Synonym substitution
4. Contraction expansion
5. Emoji expansion

Each of these synonym substitution algorithms can be designed to be more or less aggressive. And you will want to think about the language used by your users in your domain. For example, in the legal, technical, or medical fields, it's rarely a good idea to substitute synonyms. A doctor wouldn't want a chatbot telling his patient their "heart is broken" because of some synonym substitutions on the heart emoticon ("<3").

Nonetheless, the use cases for lemmatization and stemming apply to synonym substitution.

Use cases

When should you use a lemmatizer, stemmer, or synonym substitution? Stemmers are generally faster to compute and require less-complex code and datasets. But stemmers will make more errors and stem a far greater number of words, reducing the information content or meaning of your text much more than a lemmatizer would. Both stemmers and lemmatizers will reduce your vocabulary size and increase the ambiguity of the text. But lemmatizers do a better job retaining as much of the information content as possible based on how the word was used within the text and its intended meaning. As a result, some state of the art NLP packages, such as spaCy, do not provide

stemming functions and only offer lemmatization methods.

If your application involves search, stemming and lemmatization will improve the recall of your searches by associating more documents with the same query words. However, stemming, lemmatization, and even case folding will usually reduce the precision and accuracy of your search results. These vocabulary compression approaches may cause your information retrieval system (search engine) to return many documents not relevant to the words' original meanings. These are called "false positives", a incorrect matches to your search query. Sometimes "false positives" are less important than false negatives. A false negative for a search engine is when it fails to list the document you are looking for at all.

Because search results can be ranked according to relevance, search engines and document indexes typically use lemmatization when they process your query and index your documents. Because search results can be ranked according to relevance, search engines and document indexes typically use lemmatization in their NLP pipeline. This means a search engine will use lemmatization when they tokenize your search text as well as when they index their collection of documents, such as the web pages they crawl.

But they combine search results for unstemmed versions of words to rank the search results that they present to you.[\[104\]](#)

For a search-based chatbot, precision is usually more important than recall. A false positive match can cause your chatbot says something inappropriate. False negatives just cause your chatbot to have to humbly admit that it cannot find anything appropriate to say. Your chatbot will sound better if your NLP pipeline first searches for matches to your user's questions using unstemmed, unnormalized words. Your search algorithm can fall back to normalized token matches if it cannot find anything else to say. And you can rank these **fallback** matches for normalized tokens lower than the unnormalized token matches. You can even give your bot humility and transparency by introducing lower ranked responses with a caveat, such as "I haven't heard a phrase like that before, but using my stemmer I found..." In a modern world crowded with blowhard chatbots, your humbler chatbot can make a name for itself and win out![\[105\]](#)

There are 4 situations when synonym substitution of some sort may make sense.

1. Search engines
2. Data augmentation
3. Scoring the robustness of your NLP
4. Adversarial NLP

Search engines can improve their recall for rare terms by using synonym substitution. When you have limited labeled data, you can often expand your dataset 10 fold (10x) with synonym substitution alone. If you want to find a lower bound on the accuracy of your model you can aggressively substitute synonyms in your test set to see how robust your model is to these changes. And if you are searching for ways to poison or evade detection by an NLP algorithm, synonyms can give you a large number of probing texts to try. You can imagine that substituting the "currency" for the word "cash", "dollars", or "" might help evade a spam detector.



Important

Bottom line, try to avoid stemming, lemmatization, case folding, or synonym substitution, unless you have a limited amount of text with contains usages and capitalizations of the words you are interested in. And with the explosion of NLP datasets, this is rarely the case for English documents, unless your documents use a lot of jargon or are from a very small subfield of science, technology, or literature. Nonetheless, for languages other than English, you may still find uses for lemmatization. The Stanford information retrieval course dismisses stemming and lemmatization entirely, due to the negligible recall accuracy improvement and the significant reduction in precision.^[106]

[88] Linguistic and NLP techniques are often used to glean information from DNA and RNA, this site provides a list of amino acid symbols that can help you translate amino acid language into a human-readable language: "Amino Acid - Wikipedia"

(https://en.wikipedia.org/wiki/Amino_acid#Table_of_standard_amino_acids

[89] You may have learned about trigram indexes in your database class or the

documentation for PostgreSQL (`postgres`). But these are triplets of characters. They help you quickly retrieve fuzzy matches for strings in a massive database of strings using the `%` and `~*` SQL full text search queries.

[90] A more comprehensive list of stop words for various languages can be found in NLTK's corpora

(https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/stopwords.zip).

[91] See the web page titled "Analysis of text data and Natural Language Processing" (http://rstudio-pubs-static.s3.amazonaws.com/41251_4c55dff8747c4850a7fb26fb9a969c8f.html).

[92] An SEO company (<https://www.ranks.nl/stopwords>), the spaCy package (<https://stackoverflow.com/a/51627002/623735>) and NLTK (<https://pypi.org/project/nltk>) were combined to form this exhaustive list

[93] from Ted Chiang, *Exhalation*, "Truth of Fact, Truth of Fiction"

[94] See the web page titled "Camel case case - Wikipedia" (https://en.wikipedia.org/wiki/Camel_case_case).

[95] We're assuming the behavior of `str.lower()` in Python 3. In Python 2, bytes (strings) could be lowercased by just shifting all alpha characters in the ASCII number (`ord`) space, but in Python 3 `str.lower` properly translates characters so it can handle embellished English characters (like the "acute accent" diacritic mark over the e in *resumé*) as well as the particulars of capitalization in non-English languages.

[96] The trigram "cup of joe" (https://en.wiktionary.org/wiki/cup_of_joe) is slang for "cup of coffee."

[97] Check our Appendix D to learn more about *precision* and *recall*. Here's a comparison of the recall of various search engines on the Webology site (<http://www.webology.org/2005/v2n2/a12.html>).

[98] Review Appendix D if you have forgotten how to measure recall or visit

the wikipedia page to learn more (https://en.wikipedia.org/wiki/Precision_and_recall).

[99] See "An algorithm for suffix stripping", 1993 (http://www.cs.toronto.edu/~frank/csc2501/Readings/R2_Porter/Porter-1980.pdf) by M.F. Porter.

[100] See the web page titled "Snowball: A language for stemming algorithms" (<http://snowball.tartarus.org/texts/introduction.html>).

[101] This is a trivially abbreviated version of Julia Menchavez's implementation porter-stemmer on GitHub (<https://github.com/jedijulia/porter-stemmer/blob/master/stemmer.py>).

[102] Sorry Chick, Porter doesn't like your obsfucate username ;)

[103] Thank you Kyle Gorman for pointing this out

[104] Additional metadata is also used to adjust the ranking of search results. Duck Duck Go and other popular web search engines combine more than 400 independent algorithms (including user-contributed algorithms) to rank your search results (<https://duck.co/help/results/sources>).

[105] "Nice guys finish first!"—M.A. Nowak author of *SuperCooperators*"

[106] See the Stanford NLP Information Retrieval (IR) book section titled "Stemming and lemmatization" (<https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html>).

2.8 Sentiment

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

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

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

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

Another common application of sentiment analysis is junk mail and troll message filtering. You would like your chatbot to be able to measure the sentiment in the chat messages it processes so it can respond appropriately. And even more importantly, you want your chatbot to measure its own sentiment of the statements it is about to send out, which you can use to steer your bot to be kind and pro-social with the statements it makes. The simplest way to do this might be to do what Moms told us to do: If you cannot say something nice, do not say anything at all. So you need your bot to measure the niceness of everything you are about to say and use that to decide whether to respond.

What kind of pipeline would you create to measure the sentiment of a block of text and produce this sentiment positivity number? Say you just want to measure the positivity or favorability of a text—how much someone likes a product or service that they are writing about. Say you want your NLP pipeline and sentiment analysis algorithm to output a single floating point

number between -1 and +1. Your algorithm would output +1 for text with positive sentiment like "Absolutely perfect! Love it! :-) :-)". And your algorithm should output -1 for text with negative sentiment like "Horrible! Completely useless. :(". Your NLP pipeline could use values near 0, like say +0.1, for a statement like "It was OK. Some good and some bad things".

There are two approaches to sentiment analysis:

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

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

The second approach, machine learning, relies on a labeled set of statements or documents to train a machine learning model to create those rules. A machine learning sentiment model is trained to process input text and output a numerical value for the sentiment you are trying to measure, like positivity or spamminess or trolliness. For the machine learning approach, you need a lot of data, text labeled with the "right" sentiment score. Twitter feeds are often used for this approach because the hash tags, such as `\#awesome` or `\#happy` or `\#sarcasm`, can often be used to create a "self-labeled" dataset. Your company may have product reviews with five-star ratings that you could associate with reviewer comments. You can use the star ratings as a numerical score for the positivity of each text. We show you shortly how to process a dataset like this and train a token-based machine learning algorithm called *Naive Bayes* to measure the positivity of the sentiment in a set of reviews after you are done with VADER.

2.8.1 VADER—A rule-based sentiment analyzer

Hutto and Gilbert at GA Tech came up with one of the first successful rule-based sentiment analysis algorithms. They called their algorithm VADER, for **Valence Aware Dictionary for sEntiment Reasoning**.^[107] Many NLP packages implement some form of this algorithm. The NLTK package has an implementation of the VADER algorithm in `nltk.sentiment.vader`. Hutto himself maintains the Python package `vaderSentiment`. You will go straight to the source and use `vaderSentiment` here.

You will need to `pip install vaderSentiment` to run the following example.^[108] You have not included it in the `nlpia` package.

```
>>> from vaderSentiment.vaderSentiment import SentimentIntensityAnalyzer
>>> sa = SentimentIntensityAnalyzer()
>>> sa.lexicon #1
{ ...
':(':-1.9, #2
'):)': 2.0,
...
'pls': 0.3, #3
'plz': 0.3,
...
'great': 3.1,
...
}
>>> [(tok, score) for tok, score in sa.lexicon.items() #4
... if " " in tok]
[("({}{ )", 1.6),
("can't stand", -2.0),
('fed up', -1.8),
('screwed up', -1.5)]
>>> sa.polarity_scores(text=\n...     "Python is very readable and it's great for NLP.\")\n{'compound': 0.6249, 'neg': 0.0, 'neu': 0.661,\n'pos': 0.339} #5
>>> sa.polarity_scores(text=\n...     "Python is not a bad choice for most applications.\")\n{'compound': 0.431, 'neg': 0.0, 'neu': 0.711,\n'pos': 0.289} #6
```

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

```

>>> corpus = ["Absolutely perfect! Love it! :-) :-) :-)",  

...           "Horrible! Completely useless. :(,",  

...           "It was OK. Some good and some bad things."]
>>> for doc in corpus:  

...     scores = sa.polarity_scores(doc)  

...     print('{:+}: {}'.format(scores['compound'], doc))
+0.9428: Absolutely perfect! Love it! :-) :-) :-)
-0.8768: Horrible! Completely useless. :(
+0.3254: It was OK. Some good and some bad things.

```

This looks a lot like what you wanted. So the only drawback is that VADER does not look at all the words in a document. VADER only "knows" about the 7,500 words or so that were hard-coded into its algorithm. What if you want all the words to help add to the sentiment score? And what if you do not want to have to code your own understanding of the words in a dictionary of thousands of words or add a bunch of custom words to the dictionary in `SentimentIntensityAnalyzer.lexicon`? The rule-based approach might be impossible if you do not understand the language because you would not know what scores to put in the dictionary (`lexicon`)!

That is what machine learning sentiment analyzers are for.

2.8.2 Closeness of vectors

Why do we use bags of words rather than bags of characters to represent natural language text? For a cryptographer trying to decrypt an unknown message, frequency analysis of the characters in the text would be a good way to go. But for natural language text in your native language, words turn out to be a better representation. You can see this if you think about what we are using these BOW vectors for.

If you think about it, you have a lot of different ways to measure the closeness of things. You probably have a good feel for what a close family relative would be. Or the closeness of the cafes where you can meet your friend to collaborate on writing a book about AI. For cafes your brain probably uses Euclidean distance on the 2D position of the cafes you know about. Or maybe Manhattan or taxi-cab distance.

But do you know how to measure the closeness of two pieces of text? In

chapter 4 you'll learn about edit distances that check the similarity of two strings of characters. But that doesn't really capture the essence of what you care about.

How close are these sentences to each other, in your mind?

I am now coming over to see you.

I am not coming over to see you.

Do you see the difference? Which one would you prefer to receive an e-mail from your friend. The words "now" and "not" are very far apart in meaning. But they are very close in spelling. This is an example about how a single character can change the meaning of an entire sentence.

If you just counted up the characters that were different you'd get a distance of 1. And then you could divide by the length of the longest sentence to make sure your distance value is between 0 and 1. So your character difference or distance calculation would be 1 divided by 32 which gives 0.03125, or about 3%. Then, to turn a distance into a closeness you just subtract it from 1. So do you think these two sentences are 0.96875, or about 97% the same? They mean the opposite. So we'd like a better measure than that.

What if you compared words instead of characters? In that case you would have one word out of seven that was changed. That is a little better than one character out of 32. The sentences would now have a closeness score of six divided by seven or about 85%. That's a little lower, which is what we want.

For natural language you don't want your closeness or distance measure to rely only on a count of the differences in individual characters. This is one reason why you want to use words as your tokens of meaning when processing natural language text.

What about these two sentences?

She and I will come over to your place at 3:00.

At 3:00, she and I will stop by your apartment.

Are these two sentences close to each other in meaning? They have the exact same length in characters. And they use some of the same words, or at least synonyms. But those words and characters are not in the same order. So we need to make sure that our representation of the sentences does not rely on the precise position of words in a sentence.

Bag of words vectors accomplish this by creating a position or slot in a vector for every word you've seen in your vocabulary. You may have learned of a few measures of closeness in geometry and linear algebra.

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

Or imagine trying to discriminate between a pluralizing "s" at the end of a word like "words" and a normal "s" at the end of words like "bus" and "lens". Do isolated individual letters in a word or parts of a word provide any information at all about that word's meaning? Can the letters be misleading? Yes and yes.

In this chapter we show you how to make your NLP pipeline a bit smarter by dealing with these word spelling challenges using conventional stemming approaches. Later, in chapter 5, we show you statistical clustering approaches that only require you to amass a collection of natural language text containing the words you're interested in. From that collection of text, the statistics of word usage will reveal "semantic stems" (actually, more useful clusters of words like lemmas or synonyms), without any hand-crafted regular expressions or stemming rules.

2.8.3 Count vectorizing

In the previous sections you've only been concerned with keyword detection. Your vectors indicated the presence or absence of words. In order to handle longer documents and improve the accuracy of your NLP pipeline, you're going to start counting the occurrences of words in your documents.

You can put these counts into a sort-of histogram. Just like before you will create a vector for each document in your pipeline. Only instead of 0's and 1's in your vectors there will be counts. This will improve the accuracy of all the similarity and distance calculations you are doing with these counts. And just like normalizing histograms can improve your ability to compare two histograms, normalizing your word counts is also a good idea. Otherwise a really short wikipedia article that uses Barak Obama's name only once along side all the other presidents might get as much "Barack Obama" credit as a much longer page about Barack Obama that uses his name many times. Users and Question Answering bots like qary trying to answer questions about Obama might get distracted by pages listing all the presidents and might miss the main Barack Obama page entirely. So it's a good idea to normalize your count vectors by dividing the counts by the total length of the document. This more fairly represents the distribution of tokens in the document and will create better similarity scores with other documents, including the text from a search query from qary. [\[109\]](#)

Each position in your vector represents the count for one of your keywords. And having a small vocabulary keeps this vector small, low-dimensional, and easy to reason about. And you can use this *count vectorizing* approach even for large vocabularies.

And you can organize these counts of those keywords into a vector. This opens up a whole range of powerful tools for doing vector algebra.

In natural language processing, composing a numerical vector from text is a particularly "lossy" feature extraction process. Nonetheless the bag-of-words (BOW) vectors retain enough of the information content of the text to produce useful and interesting machine learning models. The techniques for

sentiment analyzers at the end of this chapter are the exact same techniques Google used to save email from a flood of spam that almost made it useless.

2.8.4 Naive Bayes

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

For any machine learning algorithm, you first need to find a dataset. You need a bunch of text documents that have labels for their positive emotional content (positivity sentiment). Hutto compiled four different sentiment datasets for us when he and his collaborators built VADER. You will load them from the `nlpia` package.[\[110\]](#)

```
>>> movies = pd.read_csv('https://proai.org/movie-reviews.csv.gz'  
...     index_col=0)  
>>> movies.head().round(2)  
      sentiment                         text  
id  
1      2.27  The Rock is destined to be the 21st Century's ...  
2      3.53  The gorgeously elaborate continuation of ''The...  
3     -0.60                  Effective but too tepid biopic  
4      1.47  If you sometimes like to go to the movies to h...  
5      1.73  Emerges as something rare, an issue movie that...  
  
>>> movies.describe().round(2)  
      sentiment  
count    10605.00  
mean      0.00 #1  
std       1.92  
min     -3.88 #2  
...  
max      3.94 #3
```

It looks like the movie reviews have been *centered*: normalized by

subtracting the mean so that the new mean will be zero and they aren't biased to one side or the other. And it seems the range of movie ratings allowed was -4 to +4.

Now you can tokenize all those movie review texts to create a bag of words for each one. If you put them all into a Pandas DataFrame that will make them easier to work with.

```
>>> import pandas as pd
>>> pd.options.display.width = 75 #1
>>> from nltk.tokenize import casual_tokenize #2
>>> bags_of_words = []
>>> from collections import Counter #3
>>> for text in movies.text:
...     bags_of_words.append(Counter(casual_tokenize(text)))
>>> df_bows = pd.DataFrame.from_records(bags_of_words) #4
>>> df_bows = df_bows.fillna(0).astype(int) #5
>>> df_bows.shape #6
(10605, 20756)

>>> df_bows.head()
      !   "   #   $   %   &   '   ...   zone   zoning   zzzzzzzz   ½   élan   -
0   0   0   0   0   0   0   0   4   ...   0   0   0   0   0   0   0
1   0   0   0   0   0   0   0   4   ...   0   0   0   0   0   0   0
2   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0
3   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0
4   0   0   0   0   0   0   0   0   ...   0   0   0   0   0   0   0

>>> df_bows.head()[list(bags_of_words[0].keys())]
    The   Rock   is   destined   to   be   ...   Van   Damme   or   Steven   S
0     1       1       1           1       2       1   ...       1       1       1       1
1     2       0       1           0       0       0   ...       0       0       0       0
2     0       0       0           0       0       0   ...       0       0       0       0
3     0       0       1           0       4       0   ...       0       0       0       0
4     0       0       0           0       0       0   ...       0       0       0       0
```

When you do not use case normalization, stop word filters, stemming, or lemmatization your vocabulary can be quite huge because you are keeping track of every little difference in spelling or capitalization of words. Try inserting some dimension reduction steps into your pipeline to see how they affect your pipeline's accuracy and the amount of memory required to store all these BOWs.

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

```
>>> from sklearn.naive_bayes import MultinomialNB
>>> nb = MultinomialNB()
>>> nb = nb.fit(df_bows, movies.sentiment > 0) #1
>>> movies['pred_senti'] = (
...     nb.predict_proba(df_bows)[:, 1] * 8 - 4 #2
>>> movies['error'] = movies.pred_senti - movies.sentiment
>>> mae = movies['error'].abs().mean().round(1) #3
>>> mae
1.9
```

To create a binary classification label you can use the fact that the centered movie ratings (sentiment labels) are positive (greater than zero) when the sentiment of the review is positive.

```
>>> movies['senti_ispos'] = (movies['sentiment'] > 0).astype(int)
>>> movies['pred_ispos'] = (movies['pred_senti'] > 0).astype(int)
>>> columns = [c for c in movies.columns if 'senti' in c or 'pred'
>>> movies[columns].head(8)
   sentiment  pred_senti  senti_ispos  pred_ispos
id
1      2.266667          4            1
2      3.533333          4            1
3     -0.600000         -4            0
4      1.466667          4            1
5      1.733333          4            1
6      2.533333          4            1
7      2.466667          4            1
8      1.266667         -4            1
>>> (movies.pred_ispos ==
...     movies.senti_ispos).sum() / len(movies)
0.9344648750589345 #1
```

This is a pretty good start at building a sentiment analyzer with only a few lines of code (and a lot of data). You did not have to guess at the sentiment associated with a list of 7500 words and hard code them into an algorithm such as VADER. Instead you told the machine the sentiment ratings for whole text snippets. And then the machine did all the work to figure out the sentiment associated with each word in those texts. That is the power of machine learning and NLP!

How well do you think this model will generalize to a completely different set text examples such as product reviews? Do people use the same words to describe things they like in movie and product reviews such as electronics and household goods? Probably not. But it's a good idea to check the robustness of your language models by running it against challenging text from a different domain. And by testing your model on new domains, you can get ideas for more examples and datasets to use in your training and test sets.

First you need to load the product reviews.

```
>>> products = pd.read_csv('https://proai.org/product-reviews.csv')
>>> for text in products['text']:
...     bags_of_words.append(Counter(casual_tokenize(text)))
>>> df_product_bows = pd.DataFrame.from_records(bags_of_words)
>>> df_product_bows = df_product_bows.fillna(0).astype(int)
>>> df_all_bows = df_bows.append(df_product_bows)
>>> df_all_bows.columns #1
Index(['!', "'", '#', '#38', '$', '%', '&', '''', '(', '(8',
      ...
      'zoomed', 'zooming', 'zooms', 'zx', 'zzzzzzzz', '~', '%',
      '_', "'"], dtype='object', length=23302)
>>> df_product_bows = df_all_bows.iloc[len(movies):][df_bows.colu
>>> df_product_bows.shape
(3546, 20756)

>>> df_bows.shape #3
(10605, 20756)
```

Now you need to convert the labels to mimic the binary classification data that you trained your model on.

```
>>> products['senti_ispos'] = (products['sentiment'] > 0).astype(
>>> products['pred_ispos'] = nb.predict(df_product_bows).astype(i
>>> products.head()
   id  sentiment  text
0  1_1    -0.90 troubleshooting ad-2500 and ad-2600 no picture.
1  1_2    -0.15  repost from january 13, 2004 with a better fit.
2  1_3    -0.20 does your apex dvd player only play dvd audio .
3  1_4    -0.10 or does it play audio and video but scrolling .
4  1_5    -0.50 before you try to return the player or waste h.

>>> tp = products['pred_ispos'] == products['senti_ispos'] #1
```

```
>>> tp.sum() / len(products)
0.5572476029328821
```

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

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

[¹⁰⁷] "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text" by Hutto and Gilbert
(<http://comp.social.gatech.edu/papers/icwsm14.vader.hutto.pdf>).

[¹⁰⁸] You can find more detailed installation instructions with the package source code on github (<https://github.com/cjhutto/vaderSentiment>).

[¹⁰⁹] Qary is an open source virtual assistant that actually assists you instead of manipulating and misinforming you (<https://docs.qary.ai>).

[¹¹⁰] If you have not already installed `nlpia`, check out the installation instructions at <http://gitlab.com/tangibleai/nlpia2>.

2.9 Review

1. How does a lemmatizer increase the likelihood that your DuckDuckGo search results contain what you are looking for?
2. Is there a way to optimally decide the *n* in the *n*-gram range you use to tokenize your documents?

3. Does lemmatization, case folding, or stopword removal help or hurt your performance on a model to predict misleading news articles with this Kaggle dataset:
4. How could you find out the best sizes for the word pieces or sentence pieces for your tokenizer?
5. Is there a website where you can download the token frequencies for most of the words and n-grams ever published?[\[111\]](#)
6. What are the risks and possible benefits of pair coding AI assistants built with NLP? What sort of organizations and algorithms do you trust with your mind and your code?

[\[111\]](#) Hint: A company that aspired to "do no evil", but now does, created this massive NLP corpus.

2.10 Summary

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

3 Math with words (TF-IDF vectors)

This chapter covers

- Counting words, *n*-grams and *term frequencies* to analyze meaning
- Predicting word occurrence probabilities with *Zipf's Law*
- Representing natural language texts as vectors
- Finding relevant documents in a collection of text using *document frequencies*
- Estimating the similarity of pairs of documents with *cosine similarity*

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 you would like to know which words are more important to a particular document and across the corpus as a whole. So you can use that "importance" value to find relevant documents in a corpus based on keyword importance within each document. That will make a spam detector a little less likely to get tripped up by a single curse word or a few slightly-spammy words within an email. And you would like to measure how positive and prosocial a tweet is when you have a broad range of words with various degrees of "positivity" scores or labels. If you have an idea about the frequency with which those words appear in a document *in relation to* the rest of the documents, you can use that to further refine the "positivity" of the document. In this chapter, you'll learn about a more nuanced, less binary measure of words and their usage within a document. This approach has been the mainstay for generating features from natural language for commercial search engines and spam filters for decades.

The next step in your adventure is to turn the words of chapter 2 into continuous numbers rather than just integers representing word counts or binary "bit vectors" that detect the presence or absence of particular words. With representations of words in a continuous space, you can operate on their

representation with more exciting math. Your goal is to find numerical representation of words that somehow capture the importance or information content of the words they represent. You'll have to wait until chapter 4 to see how to turn this information content into numbers that represent the **meaning** of words.

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

- *Bags of words*— Vectors of word counts or frequencies
- *Bags of n-grams*— Counts of word pairs (bigrams), triplets (trigrams), and so on
- *TF-IDF vectors*— Word scores that better represent their importance



Important

TF-IDF stands for *term frequency times inverse document frequency*. Term frequencies are the counts of each word in a document, which you learned about in previous chapters. Inverse document frequency means that you'll divide each of those word counts by the number of documents in which the word occurs.

Each of these techniques can be applied separately or as part of an NLP pipeline. These are all statistical models in that they are *frequency* based. Later in the book, you'll see various ways to peer even deeper into word relationships and their patterns and non-linearities.

But these "shallow" NLP machines are powerful and useful for many practical applications such as search, spam filtering, sentiment analysis, and even chatbots.

3.1 Bag of words

In the previous chapter, you created your first vector space model of a text. You used one-hot encoding of each word and then combined 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 such as a Pandas DataFrame.

You then looked at an even more useful vector representation that counts the number of occurrences, or frequency, of each word in the given text. As a first approximation, you assume that the more times a word occurs, the more meaning it must contribute to that document. 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 "cats" and "gravity". Or if you have classified some words as expressing positive emotions— words like "good", "best", "joy", and "fantastic"— the more a document that contains those words, the more likely it is to have positive "sentiment". You can imagine though how an algorithm that relied on these simple rules might be mistaken or led astray.

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

```
>>> import spacy
>>> spacy.cli.download("en_core_web_sm")
>>> nlp = spacy.load("en_core_web_sm")
>>> sentence = ('It has also arisen in criminal justice, healthca
...      'hirng, compounding existing racial, economic, and gende
>>> doc = nlp(sentence)
>>> tokens = [token.text for token in doc]
>>> tokens
['It', 'has', 'also', 'arisen', 'in', 'criminal', 'justice', ',', ,
'healthcare', ',', 'and', 'hirng', ',', 'compounding', 'existing',
'racial', ',', 'economic', ',', 'and', 'gender', 'biases', '.']
```

You could tokenize the entire Wikipedia article to get a sequence of tokens (words). With the Python `set()` type you could get the set of unique words in your document. This is called the *vocabulary* for a particular NLP pipeline (including the corpus used to train it). To count up the occurrences of all the words in your vocabulary, you can use a Python Counter type. In chapter 2 you learned that a Counter is a special kind of dictionary where the keys are all the unique objects in your array, and the dictionary values are the counts of each of those objects.

```
>>> from collections import Counter
>>> bag_of_words = Counter(tokens)
>>> bag_of_words
```

```
Counter({',': 5, 'and': 2, 'It': 1, 'has': 1, 'also': 1, 'arisen':
```

A `collections.Counter` object is a dict under the hood. And that means that the keys are technically stored in an unordered collection or set, also sometimes called a "bag." It may look like this dictionary has maintained the order of the words in your sentence, but that's just an illusion. You got lucky because your sentence didn't contain many repeated tokens. And the latest versions of Python (3.6 and above) maintain the order of the keys based on when you insert new keys into a dictionary.^[112] But you are about to create vectors out of these dictionaries of tokens and their counts. You need vectors in order to do linear algebra and machine learning on a collection of documents (sentences in this case). Your bag-of-words vectors will keep track of each unique token with a consistent index number for a position within your vectors. This way the counts for tokens like "and" or the comma add up across all the vectors for your documents—the sentences in the Wikipedia article titled "Algorithmic Bias."



Important

For NLP the order of keys in your dictionary won't matter, because you'll need to work with vectors. Just as in chapter 2, your token count vectors arrange the vocabulary (token count dict keys) according to when you processed each of the documents of your corpus. And sometimes you may want to alphabetize your vocabulary to make it easier to analyze. This means it's critical to record the order of the tokens of your vocabulary when you save or persist your pipeline to disk. And if you are trying to reproduce someone else's NLP pipeline you'll want to either use their vocabulary (token list) ordering or make sure you process the dataset in the same order the original code did.

For short documents like this one, the jumbled 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 such as detect spam, compute sentiment (positivity or other emotions), and even detect subtle intent, like sarcasm. It may be a bag, but it's full of meaning and information. So let's get these words ranked, sorted in some order that's easier to think about. The `Counter` object has a handy method, `most_common`,

for just this purpose.

```
>>> bag_of_words.most_common(3) #1
[(',', 5), ('and', 2), ('It', 1)]

>>> counts = pd.Series(dict(bag_of_words.most_common())) #2
>>> counts
,
      5
and      2
It       1
has      1
also     1
...
...
>>> len(tokens)
23
>>> counts.sum()
23
>>> counts / counts.sum() #3
,
      0.217391
and      0.086957
It       0.043478
has      0.043478
also     0.043478
...
...
```

The number of times a word occurs in a given document is called the *term frequency*, commonly abbreviated "TF." In some examples you may see the count of word occurrences normalized (divided) by the number of terms in the document. This would give you the relative frequency independent of the length of the document.

So your top two terms or tokens in this particular sentence are ",", and "and". This is a pretty common problem with natural language text—the most common words are often the least meaningful. The word "and" and the comma (",") aren't very informative about the intent of this document. And these uninformative tokens are likely to appear a lot as you wage battle against bias and injustice.

```
>>> counts['justice']
1
>>> counts['justice'] / counts.sum()
0.043478260869565216
```

This is the *normalized term frequency* of the term "justice" in this particular document which happens to be a single sentence. It's important to normalize word counts to help your NLP pipeline detect important words and to compare usages of words in documents of different lengths. But normalizing by document length doesn't really help you a whole lot in this case. But this is because you're only looking at a single document. Imagine you had one really long sentence and one really long document, say the entire Wikipedia article. If the sentence and the article were both talking about "justice" about the same amount, you would want this *normalized term frequency* score to produce roughly the same value.

Now you know how to calculate normalized term frequency and get the relative importance of each term to that document where it was used. So you've progressed nicely from just detecting the presence and absence of a word to counting up its usage frequency and now you know how to normalize this frequency. You're not done yet. This is where things get really meaningful. Now you know how important the word "justice" is to the meaning of this text. But how can a machine get that same sense that you have?

For that you're going to have to show the machine how much "justice" is used in a lot of other texts. Fortunately for budding NLP engineers Wikipedia is full of high quality accurate natural language text in many languages. You can use this text to "teach" your machine about the importance of "justice." And all you need is a few paragraphs from the Wikipedia article on algorithmic bias.

Algorithmic bias describes systematic and repeatable errors in a computer system that create unfair outcomes, such as privileging one arbitrary group of users over others. Bias can emerge due to many factors, including but not limited to the design of the algorithm or the unintended or unanticipated use or decisions relating to the way data is coded, collected, selected or used to train the algorithm. Algorithmic bias is found across platforms, including but not limited to search engine results and social media platforms, and can have impacts ranging from inadvertent privacy violations to reinforcing social biases of race, gender, sexuality, and ethnicity. The study of algorithmic bias is most concerned with algorithms that reflect "systematic and unfair"

discrimination. This bias has only recently been addressed in legal frameworks, such as the 2018 European Union's General Data Protection Regulation. More comprehensive regulation is needed as emerging technologies become increasingly advanced and opaque.

As algorithms expand their ability to organize society, politics, institutions, and behavior, sociologists have become concerned with the ways in which unanticipated output and manipulation of data can impact the physical world. Because algorithms are often considered to be neutral and unbiased, they can inaccurately project greater authority than human expertise, and in some cases, reliance on algorithms can displace human responsibility for their outcomes. Bias can enter into algorithmic systems as a result of pre-existing cultural, social, or institutional expectations; because of technical limitations of their design; or by being used in unanticipated contexts or by audiences who are not considered in the software's initial design.

Algorithmic bias has been cited in cases ranging from election outcomes to the spread of online hate speech. It has also arisen in criminal justice, healthcare, and hiring, compounding existing racial, economic, and gender biases. The relative inability of facial recognition technology to accurately identify darker-skinned faces has been linked to multiple wrongful arrests of men of color, an issue stemming from imbalanced datasets. Problems in understanding, researching, and discovering algorithmic bias persist due to the proprietary nature of algorithms, which are typically treated as trade secrets. Even when full transparency is provided, the complexity of certain algorithms poses a barrier to understanding their functioning. Furthermore, algorithms may change, or respond to input or output in ways that cannot be anticipated or easily reproduced for analysis. In many cases, even within a single website or application, there is no single "algorithm" to examine, but a network of many interrelated programs and data inputs, even between users of the same service.

-- Wikipedia *Algorithmic Bias*
(https://en.wikipedia.org/wiki/Algorithmic_bias)

Look at a sentence from this article and see if you can figure out how you could use the Counter dictionary to help your algorithm understand something about algorithmic bias.

```
>>> sentence = "Algorithmic bias has been cited in cases ranging  
...      "election outcomes to the spread of online hate speech."  
>>> tokens = [tok.text for tok in nlp(sentence)]  
>>> counts = Counter(tokens)  
>>> counts  
Counter({'Algorithmic': 1,  
         'bias': 1,  
         'has': 1,  
         'been': 1,  
         'cited': 1,  
         'in': 1,  
         'cases': 1,  
         'ranging': 1,  
         'from': 1,  
         'election': 1,  
         'outcomes': 1,  
         'to': 1,  
         'the': 1,  
         'spread': 1,  
         'of': 1,  
         'online': 1,  
         'hate': 1,  
         'speech': 1,  
         '.': 1})
```

Looks like this sentence doesn't reuse any words at all. The key to frequency analysis and term frequency vectors is the statistics of word usage. So we need to input the other sentences and create some useful word counts. To really grok "Algorithmic Bias" you could type all of the Wikipedia article into Python yourself. Or you can download it from Wikipedia directly into Python using the `wikipedia` Python package, so that you can save time to build less biased algorithms. And we've given you a head start by giving you these paragraphs in the `nlpia2` package that comes with this book:

```
>>> import requests  
>>> url = ('https://gitlab.com/tangibleai/nlpia2/'  
...          '-/raw/main/src/nlpia2/ch03/bias_intro.txt')  
>>> response = requests.get(url)  
>>> response  
<Response [200]>  
  
>>> bias_intro = response.content.decode() #1  
>>> bias_intro[:60]  
'Algorithmic bias describes systematic and repeatable errors '
```

```
>>> tokens = [tok.text for tok in nlp(bias_intro)]
>>> counts = Counter(tokens)
>>> counts
Counter({'Algorithmic': 3, 'bias': 6, 'describes': 1, 'systematic':
>>> counts.most_common(5)
[(',', 35), ('of', 16), ('.', 16), ('to', 15), ('and', 14)]
```

Okay, now that's a bit more statistically significant counts. But that is a lot of meaningless words and punctuation. It's not likely that this Wikipedia article is really about tokens like "of", "to", commas, and periods. But that's the magic of normalization. We just need to split our document so we can normalize by the counts of words across many different documents or sentences talking about different things! And it looks like you are going to want to pay attention to the least common words rather than the most common ones.

```
>>> counts.most_common()[-4:]
('inputs', 1), ('between', 1), ('same', 1), ('service', 1)]
```

Well that didn't work out so well. You were probably hoping to find terms such as "bias", "algorithmic", and "data." So you're going to have to use a formula that balances the counts to come up with the "Goldilocks" score for the ones that are "just right." The way you can do that is to come up with another useful count—the number of documents that a word occurs in, called the "document frequency." This is when things get really interesting.

```
>>> counts.most_common()[-4:]
('inputs', 1), ('between', 1), ('same', 1), ('service', 1)]
```

Across multiple documents in a corpus, things get even more interesting. That's when vector representations of counts really shine.

[112] StackOverflow discussion of whether to rely on this feature
<https://stackoverflow.com/questions/39980323/are-dictionaries-ordered-in-python-3-6/39980744#39980744>

3.2 Vectorizing text

Counter dictionaries are great for counting up tokens in text. But vectors are

where it's really at. And it turns out that dictionaries can be coerced into a DataFrame or Series just by calling the DataFrame constructor on a list of dictionaries. Pandas will take care of all the bookkeeping so that each unique token or dictionary key has its own column. And it will create NaNs whenever the Counter dictionary for a document is missing a particular key because the document doesn't contain that word.

So lets add a few more documents to your corpus of sentences from the Algorithmic Bias article. This will reveal the power of vector representations.

```
>>> docs = [nlp(s) for s in bias_intro.split_lines() if s.strip()]
>>> counts = []
>>> for doc in docs:
...     counts.append(Counter([t.text.lower() for t in doc])) #2
>>> df = pd.DataFrame(counts)
>>> df = df.fillna(0).astype(int) #3
>>> df.head()
   algorithmic bias describes systematic ... inputs between sa
0            1    1          1           1   ...      0      0
1            0    1          0           0   ...      0      0
2            1    1          0           0   ...      0      0
3            1    1          0           1   ...      0      0
4            0    1          0           0   ...      0      0
```

And when the dimensions of your vectors are used to hold scores for tokens or strings, that's when you want to use a Pandas DataFrame or Series to store your vectors. That way you can see what each dimension is for. Check out that sentence that we started this chapter with. It happens to be the eleventh sentence in the Wikipedia article.

```
>>> df.loc[10] #1
algorithmic    0
bias          0
describes     0
systematic    0
and           2
...
Name: 10, Length: 247, dtype: int64
```

Now this Pandas Series is a *vector*. That's something you can do math on. And when you do that math, Pandas will keep track of which word is where so that "bias" and "justice" aren't accidentally added together. Your row

vectors in this DataFrame have a "dimension" for each word in your vocabulary. In fact the `df.columns` attribute contains your vocabulary.

But wait, there are more than 30,000 words in a standard English dictionary. If you start processing a lot of Wikipedia articles instead of just a few paragraphs, that'll be a lot of dimensions to deal with. You are probably used to 2D and 3D vectors, because they are easy to visualize. But do concepts like distance and length even work with 30,000 dimensions? It turns out they do, and you'll learn how to improve on these high-dimensional vectors later in the book. For now just know that each element of a vector is used to represent the count, weight or importance of a word in the document you want the vector to represents.

You'll find every unique word in each document and then find all the unique words in all of your documents. In math this is the union of all the sets of words in each document. This master set of words for your documents is called the *vocabulary* of your pipeline. And if you decide to keep track of additional linguistic information about each word, such as spelling variations or parts of speech, you might call it a *lexicon*. And you might find academics that use the term *corpus* to describe a collection of documents will likely also use the word "lexicon," just because it is a more precise technical term than "vocabulary."

So take a look at the vocabulary or lexicon for this corpus. Ignoring proper nouns for now, you can lowercase your words and reduce the vocabulary size a little bit.

```
>>> docs = list(nlp(bias_intro).sents)
>>> counts = []
>>> for doc in docs:
...     counts.append(Counter([t.text.lower() for t in doc]))
>>> df = pd.DataFrame(counts)
>>> df = df.fillna(0).astype(int)
>>> df
>>> docs_tokens = []
>>> for doc_text in docs:
...     doc_text = doc_text.lower()
...     docs_tokens.append([tok.text for tok in nlp(doc_text)])
>>> len(docs_tokens[0])
```

```
>>> all_doc_tokens = []
>>> for doc_tokens in docs_tokens:
...     all_doc_tokens.extend(doc_tokens)
>>> len(all_doc_tokens)
33

>>> vocab = sorted(set(all_doc_tokens))
>>> len(vocab)
18
>>> lexicon
[',',
',',
'.',
'and',
'as',
'faster',
'get',
...
'would']
```

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

```
>>> from collections import OrderedDict
>>> zero_vector = OrderedDict((token, 0) for token in lexicon)
>>> list(zero_vector.items())[:10] #1
[('got', 0),
 ('to', 0),
 ('hairy', 0),
 ('.', 0),
 ('would', 0),
 (',', 0),
 ('harry', 0),
 ('as', 0),
 ('the', 0),
 ('faster', 0),
 ('and', 0)]
```

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

```
>>> import copy
>>> doc_vectors = []
```

```

>>> for doc in docs:
...     vec = copy.copy(zero_vector) #1
...     tokens = [token.text for token in nlp(doc.lower())]
...     token_counts = Counter(tokens)
...     for key, value in token_counts.items():
...         vec[key] = value / len(lexicon)
...     doc_vectors.append(vec)

```

3.2.1 An easier way to vectorize text

Now that you've manually created your Bag of Words vector, you might wonder if someone already found a faster way to do it. And indeed, there is! You can avoid going through tokenizing, frequency counting and manually vectorizing your bag of words vector using `scikit-learn` package.[\[113\]](#) 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.

```

pip install scipy
pip install sklearn

```

Here is how you would create the term frequency vector in `scikit-learn`. We'll use the `CountVectorizer` class. It is a *model* class with `.fit()` and `.transform()` methods that comply with the `sklearn` API for all machine learning models.

Listing 3.1. Using `scikit-learn` to compute word count vectors

```

>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = docs
>>> vectorizer = CountVectorizer()
>>> count_vectors = vectorizer.fit_transform(corpus) #1
>>> print(count_vectors.toarray()) #2
[[1 0 3 1 1 0 2 1 0 0 0 1 0 3 1 1]
 [1 0 1 0 0 1 1 0 1 1 0 0 1 0 0 0]
 [0 2 0 0 0 1 1 0 1 1 1 0 0 0 0 0]]

```

Now you have a matrix (practically a list of lists in Python) that represents the three documents (the three rows of the matrix) and the count of each term, token, or word in your lexicon make up the columns of the matrix. That was fast! With just 1 line of code, `vectorizer.fit_transform(corpus)`, we have gotten to the same result as with dozens of lines you needed to manually

tokenize, create a lexicon and count the terms. Note that these vectors have the length of 16, rather than 18 like the vectors you created manually. That's because scikit-learn tokenizes the sentences slightly differently (it only considers words of 2 letters or more as tokens) and drops the punctuation.

So, you have three vectors, one for each document. Now what? What can you do with them? Your document word-count vectors can do all the cool stuff any vector can do, so let's learn a bit more about vectors and vector spaces first.[\[114\]](#)

3.2.2 Vectorize your code

If you read about "vectorizing code" on the internet means something entirely different than "vectorizing text." Vectorizing text is about converting text into a meaningful vector representation of that text. Vectorizing code is about speeding up your code by taking advantage of powerful compiled libraries like numpy and using Python to do math as little as possible. The reason it's called "vectorizing" is because you can use vector algebra notation to eliminate the for loops in your code, the slowest part of many NLP pipelines. Instead of for loops iterating through all the elements of a vector or matrix to do math you just use numpy to do the for loop for you in compiled C code. And Pandas uses numpy under the hood for all its vector algebra, so you can mix and match a DataFrame with a numpy array or a Python float and it will all run really fast.

```
>>> v1 = np.array(list(range(5)))
>>> v2 = pd.Series(reversed(range(5)))
>>> slow_answer = sum([4.2 * (x1 * x2) for x1, x2 in zip(v1, v2)])
>>> slow_answer
42.0

>>> faster_answer = sum(4.2 * v1 * v2) #1
>>> faster_answer
42.0

>>> fastest_answer = 4.2 * v1.dot(v2) #2
>>> fastest_answer
42.0
```

Python's dynamic typing design makes all this magic possible. When you

multiply a float by an array or DataFrame, instead of raising an error because you're doing math on two different types, the interpreter will figure out what you're trying to do and "make it so," just like Sulu. And it will compute what you're looking for in the fastest possible way, using compiled C code rather than a Python for loop.



Tip

If you use vectorization to eliminate some of the for loops in your code, you can speed up your NLP pipeline by a 100x or more. This is 100x more models that you can try. The Berlin Social Science Center (WZB) has a great tutorial on vectorization.[\[115\]](#). And if you poke around elsewhere on the site you'll find perhaps the only trustworthy source of statistics and data on the affect NLP and AI is having on society.[\[116\]](#)

3.2.3 Vector spaces

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

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

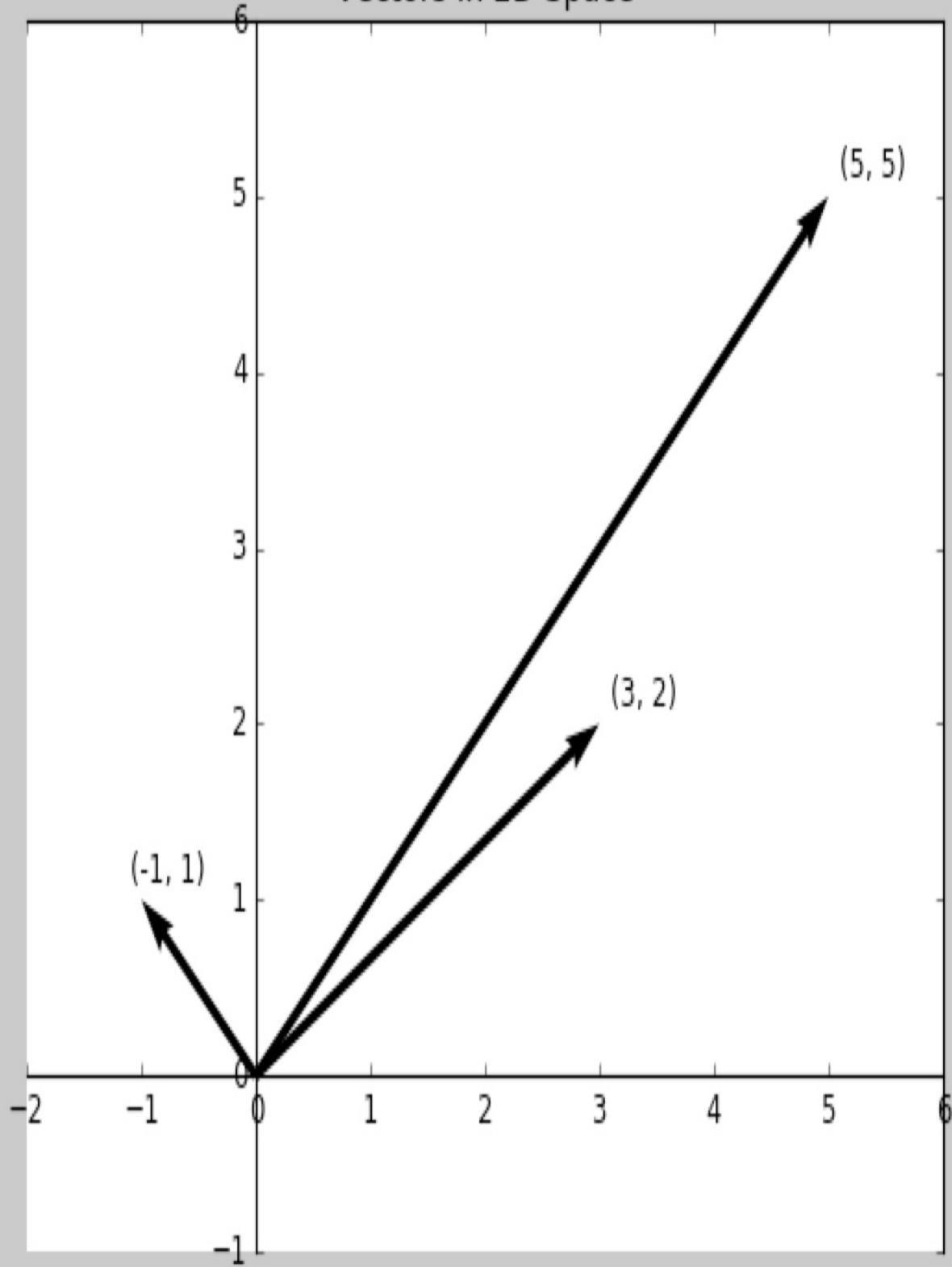
What about latitude and longitude on a map or globe? That geographic coordinate space is definitely two-dimensional because it's an ordered list of

two numbers: latitude and longitude. But each of the latitude-longitude pairs describes a point on an approximately spherical surface—the Earth's surface. The latitude-longitude vector space is not rectilinear, and Euclidean geometry doesn't exactly work in it. That means you have to be careful when you calculate things like distance or closeness between two points represented by a pair of 2D geographic coordinates, or points in any non-Euclidean space. Think about how you would calculate the distance between the latitude and longitude coordinates of Portland, OR and New York, NY.[\[117\]](#)

Figure 3.3 is one way to draw the 2D vectors $(5, 5)$, $(3, 2)$, and $(-1, 1)$. The head of a vector (represented by the pointy tip of an arrow) is used to identify a location in a vector space. So the vector heads in this diagram will be at those three pairs of coordinates. The tail of a position vector (represented by the "rear" of the arrow) is always at the origin, or $(0, 0)$.

Figure 3.1. 2D vectors

Vectors in 2D Space

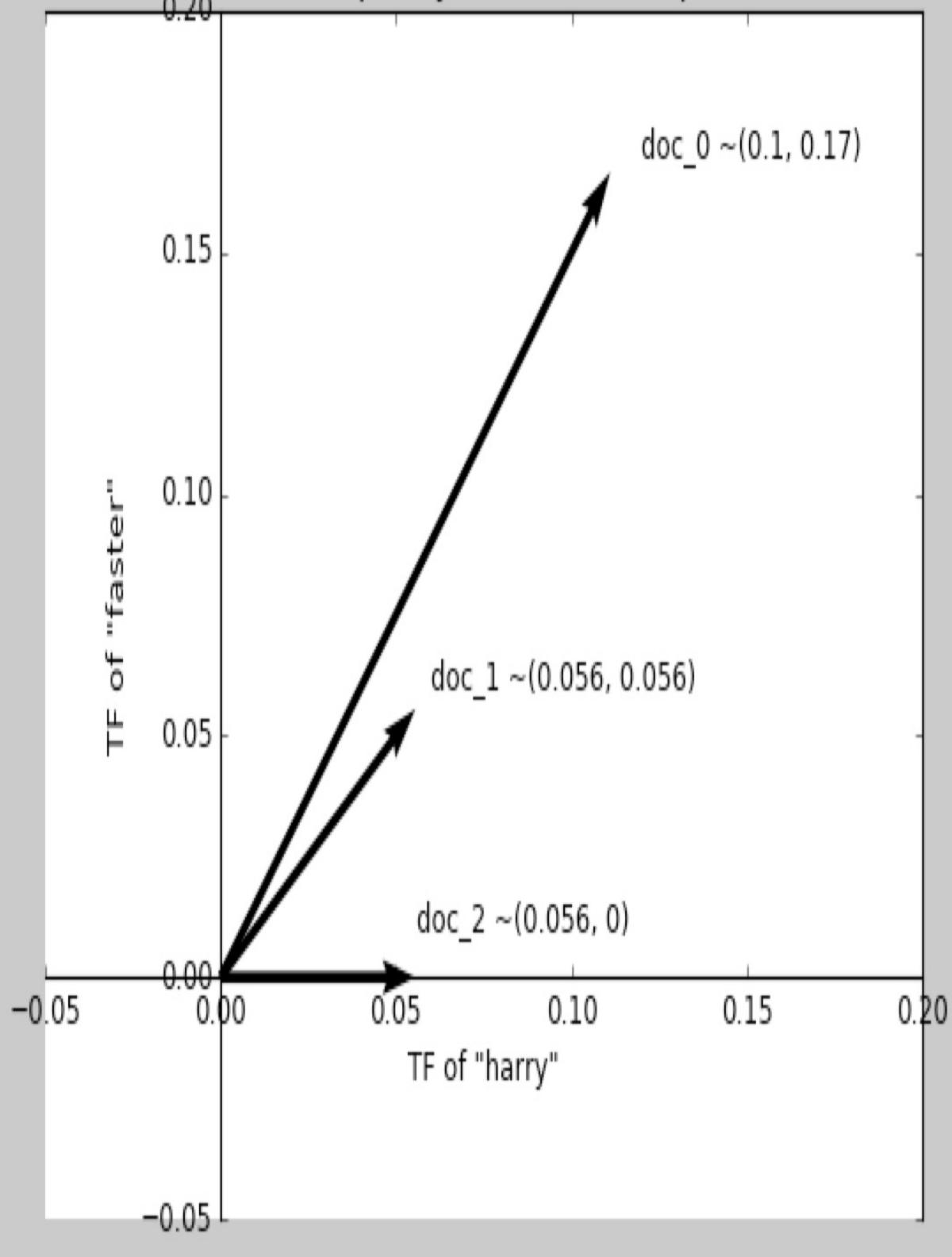


What about 3D vector spaces? Positions and velocities in the 3D physical world you live in can be represented by x, y, and z coordinates in a 3D vector. But you aren't limited to normal 3D space. You can have 5 dimensions, 10 dimensions, 5,000, whatever. The linear algebra all works out the same. You might need more computing power as the dimensionality grows. And you'll run into some "curse-of-dimensionality" issues, but you can wait to deal with that until the last chapter, chapter 13.[\[118\]](#)

For a natural language document vector space, the dimensionality of your vector space is the count of the number of distinct words that appear in the entire corpus. For TF (and TF-IDF to come), we call this dimensionality capital letter "K". This number of distinct words is also the vocabulary size of your corpus, so in an academic paper it'll usually be called " $|V|$ " You can then describe each document, within this K-dimensional vector space by a K-dimensional vector. K = 18 in your three-document corpus about Harry and Jill (or 16, if your tokenizer drops the punctuation). Because humans can't easily visualize spaces of more than three dimensions, let's set aside most of those dimensions and look at two for a moment, so you can have a visual representation of the vectors on this flat page you're reading. So in figure 3.4, K is reduced to two for a two-dimensional view of the 18-dimensional Harry and Jill vector space.

Figure 3.2. 2D term frequency vectors

Term Frequency Vectors in 2D Space

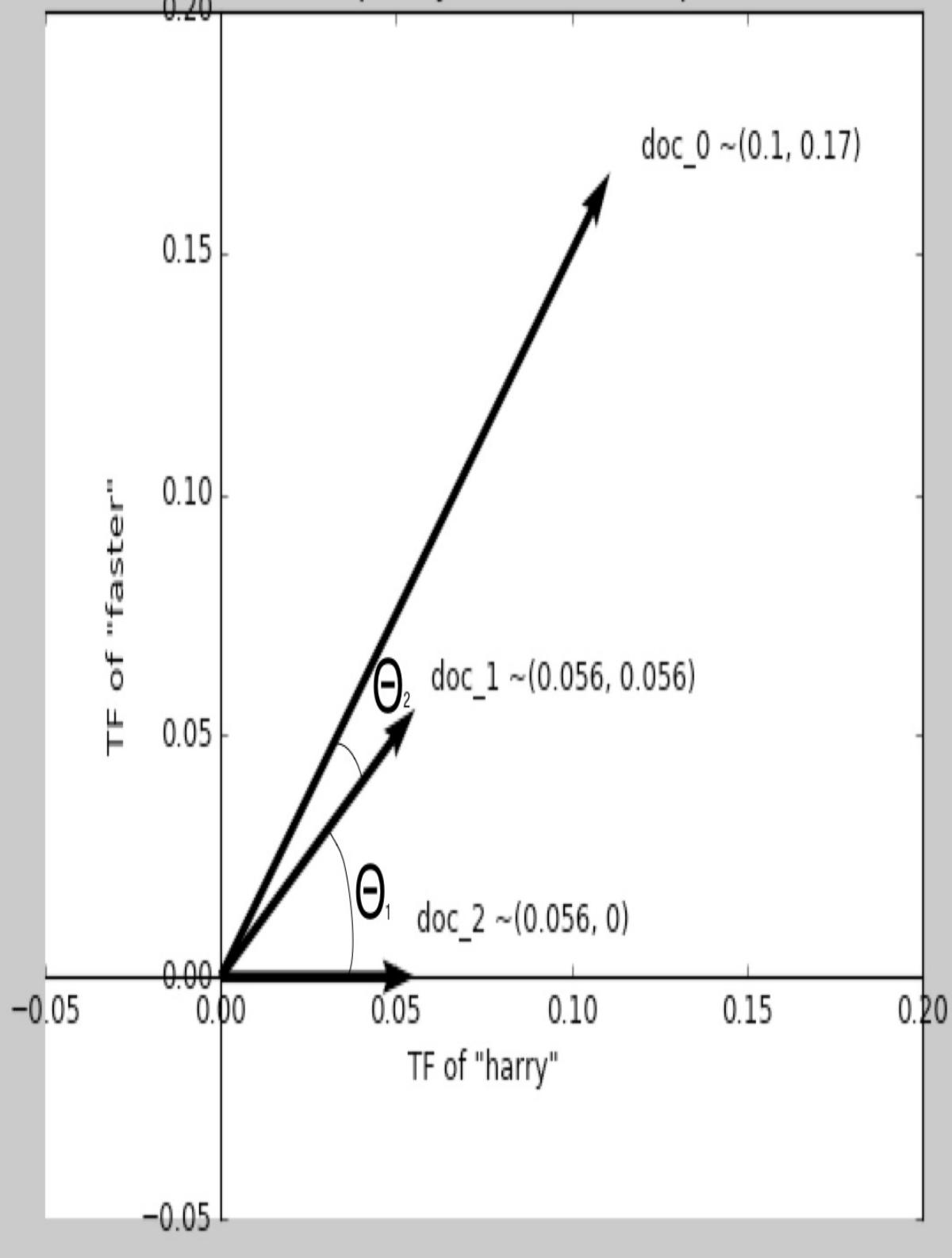


K-dimensional vectors work the same way, just in ways you can't easily visualize. Now that you have a representation of each document and know they share a common space, you have a path to compare them. You could measure the Euclidean distance between the vectors by subtracting them and computing the length of that distance between them, which is called the 2-norm distance. It's the distance a "crow" would have to fly (in a straight line) to get from a location identified by the tip (head) of one vector and the location of the tip of the other vector. Check out appendix C on linear algebra to see why this is a bad idea for word count (term frequency) vectors.

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

Figure 3.3. 2D thetas

Term Frequency Vectors in 2D Space



Cosine similarity, is merely the cosine of the angle between two vectors (theta), shown in figure 3.5, which can be calculated from the Euclidian dot product using [3.4](#). Cosine similarity is efficient to calculate because the dot product does not require evaluation of any trigonometric functions. In addition, cosine similarity has a convenient range for most machine learning problems: -1 to +1.

Equation 3.1

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}| |\mathbf{B}| * \cos(\theta)$$

In Python this would be:

```
A.dot(B) == (np.linalg.norm(A) * np.linalg.norm(B)) * np.cos(angl
```

If you solve this equation for `np.cos(angle_between_A_and_B)` (called "cosine similarity between vectors A and B") you can derive code to computer the cosine similarity:

Listing 3.2. Cosine similarity formula in Python

```
cos_similarity_between_A_and_B = np.cos(angle_between_A_and_B) \
    = A.dot(B) / (np.linalg.norm(A) * np.linalg.norm(B))
```

In linear algebra notation this becomes [3.5](#):

Equation 3.2 cosine similarity between two vectors

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{|\mathbf{A}| |\mathbf{B}|}$$

Or in pure Python without numpy:

Listing 3.3. Compute cosine similarity in python

```
>>> import math
>>> def cosine_sim(vec1, vec2):
...     vec1 = [val for val in vec1.values()] #1
```

```

...
    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 you need to take the dot product of two of your vectors in question—multiply the elements of each vector pairwise—and then sum those products up. You then divide by the norm (magnitude or length) of each vector. The vector norm is the same as its Euclidean distance from the head to the tail of the vector—the square root of the sum of the squares of its elements. This *normalized dot product*, like the output of the cosine function, will be a value between -1 and 1. It is the cosine of the angle between these two vectors. It gives you a value for how much the vectors point in the same direction.

A cosine similarity of **1** represents identical normalized vectors that point in exactly the same direction along all dimensions. The vectors may have different lengths or magnitudes, but they point in the same direction.

Remember you divided the dot product by the norm of each vector. So the closer a cosine similarity value is to 1, the closer the two vectors are in angle. For NLP document vectors that have a cosine similarity close to 1, you know that the documents are using similar words in similar proportion. So the documents whose document vectors are close to each other are likely talking about the same thing.

A cosine similarity of **0** represents two vectors that share no components. They are orthogonal, perpendicular in all dimensions. For NLP TF vectors, this situation occurs only if the two documents share no words in common. This doesn't necessarily mean they have different meanings or topics, just that they use completely different words.

A cosine similarity of **-1** represents two vectors that are anti-similar, completely opposite. They point in opposite directions. This can never happen for simple word count (term frequency) vectors or even normalized TF vectors (which we talk about later). Counts of words can never be

negative. So word count (term frequency) vectors will always be in the same "quadrant" of the vector space. None of the term frequency vectors can sneak around into one of the quadrants in the vector space. None of your term frequency vectors can have components (word frequencies) that are the negative of another term frequency vector, because term frequencies just can't be negative.

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

If you want to compute cosine similarity for regular numpy vectors, such as those returned by `CountVectorizer`, you can use `scikit-learn's built-in tools. Here is how you can calculate the cosine similarity between word vectors 1 and 2 that we computed in Listing 3.2:

```
>>> from sklearn.metrics.pairwise import cosine_similarity
>>> vec1 = count_vectors[1,:]
>>> vec2 = count_vectors[2,:]
>>> cosine_similarity(vec1, vec2)
array([[0.55901699]])
```

Note that because the vectors we got from `CountVectorizer` are slightly shorter, this distance is going to be different from cosine similarity between our DIY document vectors. As an exercise, you can check that the `sklearn` cosine similarity gives the same result for our `OrderedDict` vectors created with `Counter` class - see if you can figure it out!

[113] You can check out this package's full documentation on its webpage (<http://scikit-learn.org/>) - we'll be using it a lot in this book.

[114] If you would like more details about linear algebra and vectors take a look at Appendix C.

[115] "Vectorization and Parallelization" by WZB.eu (<https://datascience.blog.wzb.eu/2018/02/02/vectorization-and-parallelization-in-python-with-numpy-and-pandas/>).

[116] "Knowledge and Society in Times of Upheaval"
(<https://wzb.eu/en/node/60041>)

[117] You'd need to use a package like GeoPy (geopy.readthedocs.io) to get the math right.

[118] The curse of dimensionality is that vectors will get exponentially farther and farther away from one another, in Euclidean distance, as the dimensionality increases. A lot of simple operations become impractical above 10 or 20 dimensions, like sorting a large list of vectors based on their distance from a "query" or "reference" vector (approximate nearest neighbor search). To dig deeper, check out Wikipedia's "Curse of Dimensionality" article (https://en.wikipedia.org/wiki/Curse_of_dimensionality), explore hyperspace with one of this book's authors at bit.ly/explorehyperspace (<https://proai.org/slides-squash-hyperspace>), play with the Python annoy package (<https://github.com/spotify/annoy>), or search Google Scholar for "high dimensional approximate nearest neighbors" (<https://scholar.google.com/scholar?q=high+dimensional+approximate+nearest+neighbor>)

3.3 Bag of n-grams

You have already seen in the last chapter how to create n -grams from the tokens in your corpus. Now, it's time to use them to create a better representation of documents. Fortunately for you, you can use the same tools you are already familiar with, just tweak the parameters slightly.

First, let's add another sentence to our corpus, which will illustrate why bag-of-ngrams can sometimes be more useful than bag-of-words.

```
>>> new_sentence = "Is Harry hairy and faster than Jill?"  
>>> ngram_docs = copy.copy(docs)  
>>> ngram_docs.append(new_sentence)
```

If you compute the vector of word counts for this last sentence, using the same model we trained in Listing 3.2, you will see that it is exactly equal to the representation of the second sentence:

```
>>> new_sentence_vector = vectorizer_transform(new_sentence)
>>> print (new_sentence_vector.toarray())
[[1 0 1 0 0 1 1 0 1 1 0 0 1 0 0 0]]
```

To be sure, let's calculate the cosine similarity between the two document vectors:

```
>>> cosine_similarity(count_vectors[1,:], new_sentence)
[[1.]]
```

Let's now do the same vectorization process we did a few pages ago with CountVectorizer, but instead you'll "order" your CountVectorizer to count 2-grams instead of tokens:

```
>>> ngram_vectorizer = CountVectorizer(ngram_range=(1,2))
>>> ngram_vectors = ngram_vectorizer.fit_transform(corpus)
>>> print(ngram_vectors.toarray())
[[1 0 0 1 2 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0 2 1 1 1]
 [1 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0]
 [1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0]]
```

You can immediately notice that these vectors are significantly longer, as there are always more 2-grams than tokens. If you look closer, you can even notice that the representations of the second and fourth sentence are no longer the same. To be sure, let's compute the cosine similarity between them:

```
>>> cosine_similarity(ngram_vectors[1,:], ngram_vectors[2,:])
[[0.66666667]]
```

And now we can distinguish between the two sentences! It is worth noting that bag of n -grams approach has its own challenges. With large texts and corpora, the amount of n -grams increases exponentially, causing "curse-of-dimensionality" issues we mentioned before. However, as you saw in this section, there might be cases where you will want to use it instead of single token counting.

3.3.1 Analyzing this

Even though until now we only dealt with n -grams of word token, n -gram of characters can be useful too. For example, they can be used for language

detection, or authorship attribution (deciding who among the set of authors wrote the document analyzed). Let's solve a puzzle using character n -grams and the CountVectorizer class you just learned how to use.

We'll start by importing a small and interesting python package called this, and examining some of its constants:

```
>>> from this import s
>>> print (s)
Gur Mra bs Clguba, ol Gvz Crgref
Ornhgvshy vf orggre guna htyl.
Rkcyvpvg vf orggre guna vzcyvpvg.
Fvzcyr vf orggre guna pbzcyrk.
Pbzcyrk vf orggre guna pbzcyvpngrq.
Syng vf orggre guna arfgrq.
Fcnefr vf orggre guna qrafr.
Ernqnovyvgl pbhagf.
Fcrpvny pnfrf nera'g fcrpvny rabhtu gb oernx gur ehyrf.
Nygubhtu cenpgvpnyvgl orngf chevgl.
Reebef fubhyq arire cnff fvyragy1.
Hayrff rkcyvpvgyl fvyraprq.
Va gur snpr bs nzovthvgl, ershfr gur grzcgngvba gb thrff.
Gurer fubhyq or bar-- naq cersrenoyl bayl bar --boivbhf jnl gb qb
Nygubhtu gung jnl znl abg or boivbhf ng svefg hayrff lbh'er Qhgpu
Abj vf orggre guna arire.
Nygubhtu arire vf bsgra orggre guna *evtug* abj.
Vs gur vzcyrzragngvba vf uneq gb rkcytva, vg'f n onq vqrn.
Vs gur vzcyrzragngvba vf rnfl gb rkcytva, vg znl or n tbbq vqrn.
Anzrfcnprf ner bar ubaxvat terng vqrn -- yrg'f qb zber bs gubfr!
```

What are these strange words? In what language are they written? H.P. Lovecraft fans may think of the ancient language used to summon the dead deity Cthulhu.[\[119\]](#) But even to them, this message will be incomprehensible.

To figure out the meaning of our cryptic piece of text, you'll use the method you just learned - figuring out token frequency. Only this time, a little bird is telling you it might be worth to start with character tokens rather than word tokens! Luckily, CountVectorizer can serve you here as well. You can see the results of the code in Figure 3.4a

```
>>> char_vectorizer = CountVectorizer(ngram_range=(1,1), analyzer
>>> s_char_frequencies = char_vectorizer.fit_transform(s)
>>> generate_histogram(s_char_frequencies, s_char_vectorizer) #2
```

Hmmm. Not quite sure what you can do with these frequency counts. But then again, you haven't even seen the frequency counts for any other text yet. Let's choose some big document - for example, the Wikipedia article for Machine Learning, [\[120\]](#) and try to do the same analysis (check out the results in Figure 3.4b):

```
>>> DATA_DIR = ('https://gitlab.com/tangibleai/nlpia/'  
...                 '-/raw/master/src/nlpia/data')  
  
>>> url = DATA_DIR + '/machine_learning_full_article.txt'  
>>> ml_text = requests.get(url).content.decode()  
>>> ml_char_frequencies = char_vectorizer.fit_transform(ml_text)  
>>> generate_histogram(s_char_frequencies, s_char_vectorizer)
```

Now that looks interesting! If you look closer at the two frequency histograms, you might notice the similarities in the order of their highs and lows. It's as if the character frequency pattern is similar, but shifted.

To determine whether this is the real shift, let's use a technique often used in signal processing: computing the distance between the highest point of the signal, the "peak", and see if other peaks follow a similar distance. You'll use a couple of handy built-in python functions: `ord()` and `chr()`.

```
>>> peak_distance = ord('r') - ord('e')  
>>> peak_distance  
13  
>>> chr(ord('v') - peak_distance)  
'i'  
>>> chr(ord('n') - peak_distance)  
'a'
```

So, we can see that the most frequent letters in both distributions are shifted by the same `peak_distance`. That distance is preserved between the least frequent letters, too:

```
>>> chr(ord('w') - peak_distance)  
'j'
```

By this point, you have probably Googled our riddle and discovered that our message is actually encoded using `rot-13` cipher. This cipher substitutes every letter with a letter with the 13th letter after it in the alphabet. Let's use

python's codecs package to reveal what this is all about:

```
>>> import codecs  
>>> print(codecs.decode(s, 'rot-13'))  
The Zen of Python, by Tim Peters
```

Beautiful is better than ugly. Explicit is better than implicit. Simple is better than complex. Complex is better than complicated. Flat is better than nested. Sparse is better than dense. Readability counts. Special cases aren't special enough to break the rules. Although practicality beats purity. Errors should never pass silently. Unless explicitly silenced. In the face of ambiguity, refuse the temptation to guess. There should be one-- and preferably only one --obvious way to do it. Although that way may not be obvious at first unless you're Dutch. Now is better than never. Although never is often better than **right** now. If the implementation is hard to explain, it's a bad idea. If the implementation is easy to explain, it may be a good idea. Namespaces are one honking great idea— let's do more of those!

And you have revealed the Zen of Python! These words of wisdom were written by one of the Python tribe elders, Tim Peters, back in 1999 and since then have been placed in public domain, put to music,^[121] and even parodied.^[122] This book's authors are trying to act according to these principles whenever they can - and you should, too! And thanks to character *n*-grams, you were able to "translate" them from rot-13-encrypted English into the regular ones.

[119] If the reference is unfamiliar to you, check out the story *Call of Cthulhu* by H.P. Lovecraft:

<https://www.hplovecraft.com/writings/texts/fiction/cc.aspx>

[120] Retrieved on July 9th 2021 from here:

https://en.wikipedia.org/wiki/Machine_learning

[121] Check out this youtube video: <https://www.youtube.com/watch?v=i6G6dmVJy74>

[122] Try importing that package from pypi to discover Python antipatterns - but as the package creators themselves warn you,

3.4 Zipf's Law

Now on to our main topic— Sociology. Okay, not, but you'll make a quick detour into the world of counting people and words, and you'll learn 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). In the 1930s, the American linguist George Kingsley Zipf sought to formalize Estoup's observation, and this relationship eventually came to bear Zipf's name.

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

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

Specifically, *inverse proportionality* refers to a situation where an item in a ranked list will appear with a frequency tied explicitly to its rank in the list. The first item in the ranked list will appear twice as often as the second, and three times as often as the third, for example. 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, it may be worth investigating.

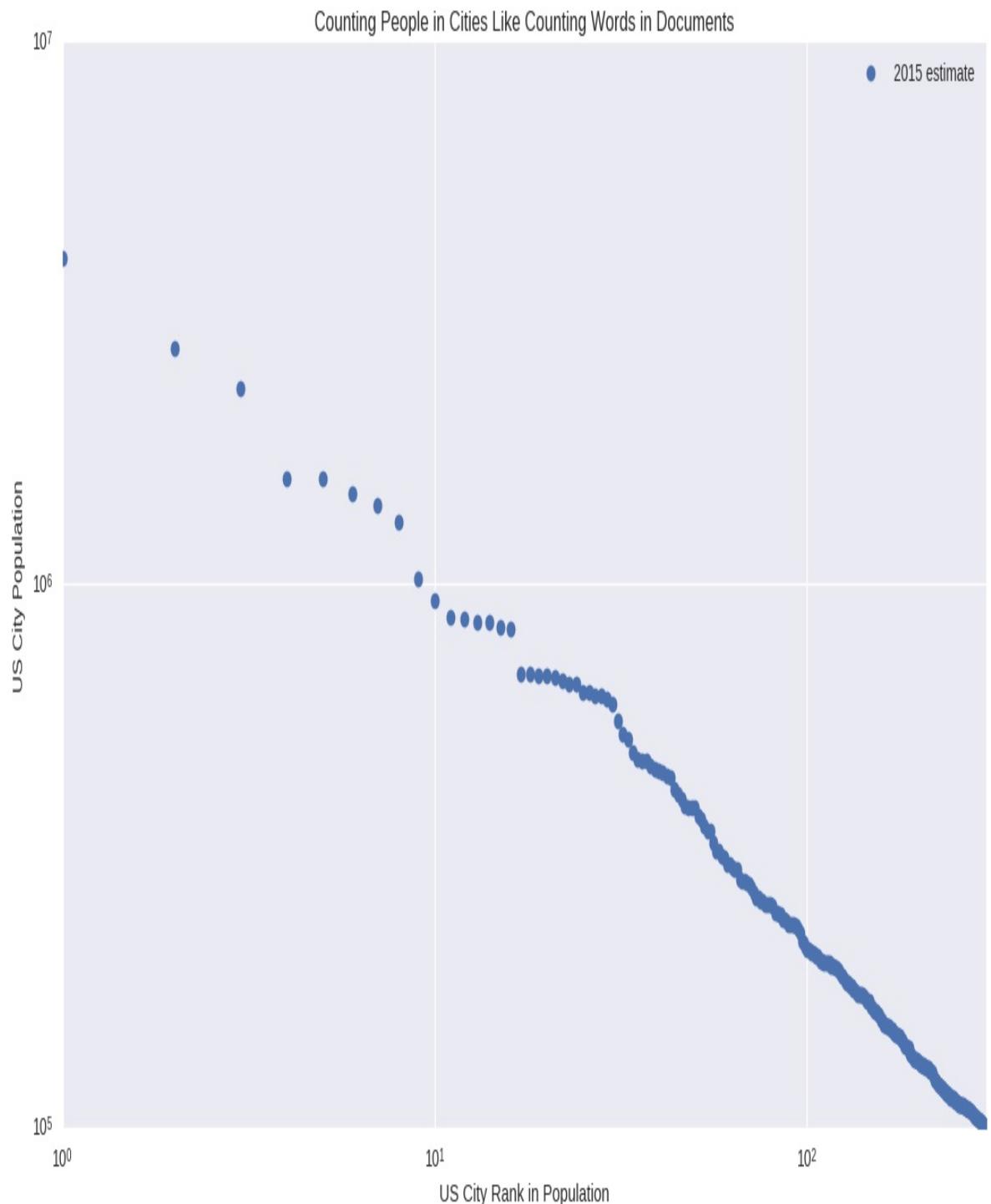
As an example of how far Zipf's Law stretches beyond the world of words, figure 3.6 charts the relationship between the population of US cities and the rank of that population. It turns out that Zipf's Law applies to counts of lots of things. Nature is full of systems that experience exponential growth and "network effects" like population dynamics, economic output, and resource distribution.^[123] It's interesting that something as simple as Zipf's Law could hold true across a wide range of natural and manmade phenomena. Nobel Laureate Paul Krugman, speaking about economic models and Zipf's Law,

put it succinctly:

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

Here is an updated version of Krugman's city population plot:[\[124\]](#)

Figure 3.4. City population distribution



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

The Brown Corpus was the first million-word electronic corpus of English, created in 1961 at Brown University. This corpus contains text from 500 sources, and the sources have been categorized by genre, such as news, editorial, and so on.[\[125\]](#)

-- NLTK Documentation

```
>>> nltk.download('brown') #1
>>> from nltk.corpus import brown
>>> brown.words()[:10] #2
['The',
 'Fulton',
 'County',
 'Grand',
 'Jury',
 'said',
 'Friday',
 'an',
 'investigation',
 'of']
>>> brown.tagged_words()[:5] #3
[('The', 'AT'),
 ('Fulton', 'NP-TL'),
 ('County', 'NN-TL'),
 ('Grand', 'JJ-TL'),
 ('Jury', 'NN-TL')]
>>> len(brown.words())
1161192
```

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

```
>>> from collections import Counter
>>> puncs = set((',', '.', '--', '-', '!', '?',
...      ':', ';', '(', ')', '[', ']'))
>>> word_list = (x.lower() for x in brown.words() if x not in puncs)
>>> token_counts = Counter(word_list)
>>> token_counts.most_common(10)
[('the', 69971),
 ('of', 36412),
 ('and', 28853),
 ('to', 26158),
 ('a', 23195),
 ('in', 21337),
 ('that', 10594),
 ('is', 10109),
```

```
('was', 9815),  
('he', 9548)]
```

A quick glance shows that the word frequencies in the Brown corpus follow the logarithmic relationship Zipf predicted. "The" (rank 1 in term frequency) occurs roughly twice as often as "of" (rank 2 in term frequency), and roughly three times as often as "and" (rank 3 in term frequency). If you don't believe us, use the example code

(https://gitlab.com/tangibleai/nlpia2/blob/main/src/nlpia2/ch03/ch03_zipf.py) in the nlpia package to see this yourself.

In short, if you rank the words of a corpus by the number of occurrences and list them in descending order, you'll 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. And it is four times as likely to appear as the fourth word on the list. So given a large corpus, you can use this breakdown to say statistically how likely a given word is to appear in any given document of that corpus.

[123] See the web page titled "There is More than a Power Law in Zipf" (<https://www.nature.com/articles/srep00812>).

[124] Population data downloaded from Wikipedia using Pandas. See the `nlpia.book.examples` code on GitHub
(https://gitlab.com/tangibleai/nlpia2/blob/main/src/nlpia2/ch03/ch03_zipf.py)

[125] For a complete list, see <http://icame.uib.no/brown/bcm-los.html>.

3.5 Inverse Document Frequency

Now back to your document vectors. Word counts and n -gram counts are useful, but pure word count, even when normalized by the length of the document, doesn't tell you much about the importance of that word in that document *relative* to the rest of the documents in the corpus. If you could suss out that information, you could start to describe documents within the corpus. Say you have a corpus of every book about artificial intelligence (AI) ever written. "Intelligence" would almost surely occur many times in every

book (document) you counted, but that doesn't provide any new information, it doesn't help distinguish between those documents. Whereas something like "neural network" or "conversational engine" might not be so prevalent across the entire corpus, but for the documents where it frequently occurred, you would know more about their nature. For this you need another tool.

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

Let's return to the Algorithmic Bias example from Wikipedia and grab another section (that deals with algorithmic racial and ethnic discrimination) and say it is the second document in your Bias corpus.

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.

Algorithms have been criticized as a method for obscuring racial prejudices in decision-making. Because of how certain races and ethnic groups were treated in the past, data can often contain hidden biases. For example, black people are likely to receive longer sentences than white people who committed the same crime. This could potentially mean that a system amplifies the original biases in the data.

In 2015, Google apologized when black users complained that an image-identification algorithm in its Photos application identified them as gorillas. In 2010, Nikon cameras were criticized when image-recognition algorithms consistently asked Asian users if they were blinking. Such examples are the product of bias in biometric data sets. Biometric data is drawn from aspects of the body, including racial features either observed or inferred, which can then be transferred into data points. Speech recognition technology can have different accuracies depending on the user's accent. This may be caused by the lack of training data for speakers of that accent.

Biometric data about race may also be inferred, rather than observed. For example, a 2012 study showed that names commonly associated with blacks

were more likely to yield search results implying arrest records, regardless of whether there is any police record of that individual's name. A 2015 study also found that Black and Asian people are assumed to have lesser functioning lungs due to racial and occupational exposure data not being incorporated into the prediction algorithm's model of lung function.

In 2019, a research study revealed that a healthcare algorithm sold by Optum favored white patients over sicker black patients. The algorithm predicts how much patients would cost the health-care system in the future. However, cost is not race-neutral, as black patients incurred about \$1,800 less in medical costs per year than white patients with the same number of chronic conditions, which led to the algorithm scoring white patients as equally at risk of future health problems as black patients who suffered from significantly more diseases.

A study conducted by researchers at UC Berkeley in November 2019 revealed that mortgage algorithms have been discriminatory towards Latino and African Americans which discriminated against minorities based on "creditworthiness" which is rooted in the U.S. fair-lending law which allows lenders to use measures of identification to determine if an individual is worthy of receiving loans. These particular algorithms were present in FinTech companies and were shown to discriminate against minorities.

-- Wikipedia *Algorithmic Bias: Racial and ethnic discrimination*
(https://en.wikipedia.org/wiki/Algorithmic_bias#Racial_and_ethnic_discrimination)

First let's get the total word count for each document in your corpus:

```
>>> DATA_DIR = ('https://gitlab.com/tangibleai/nlpia/'  
...                 '-/raw/master/src/nlpia/data')  
  
>>> url = DATA_DIR + '/bias_discrimination.txt'  
>>> bias_discrimination = requests.get(url).content.decode()  
>>> intro_tokens = [token.text for token in nlp(bias_intro.lower())]  
>>> disc_tokens = [token.text for token in nlp(bias_discrimination)]  
>>> intro_total = len(intro_tokens)  
>>> intro_total  
479  
>>> disc_total = len(disc_tokens)  
>>> disc_total  
451
```

Now with a couple of tokenized documents about bias in hand, let's look at the term frequency of the term "bias" in each document. You'll store the TFs you find in two dictionaries, one for each document.

```
>>> intro_tf = {}
>>> disc_tf = {}
>>> intro_counts = Counter(intro_tokens)
>>> intro_tf['bias'] = intro_counts['bias'] / intro_total
>>> disc_counts = Counter(disc_tokens)
>>> disc_tf['bias'] = disc_counts['bias'] / disc_total
>>> 'Term Frequency of "bias" in intro is:{:.4f}'.format(intro_tf)
Term Frequency of "bias" in intro is:0.0167
>>> 'Term Frequency of "bias" in discrimination chapter is: {:.4f}
...     .format(disc_tf['bias']))
'Term Frequency of "bias" in discrimination chapter is: 0.0022'
```

Okay, you have a number eight times as large as the other. Is the intro section eight times as much about bias? 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
>>> disc_tf['and'] = disc_counts['and'] / disc_total
>>> print('Term Frequency of "and" in intro is: {:.4f}'\
...     .format(intro_tf['and']))
Term Frequency of "and" in intro is: 0.0292
>>> print('Term Frequency of "and" in discrimination chapter is:
...     .format(disc_tf['and']))
Term Frequency of "and" in discrimination chapter is: 0.0303
```

Great! You know both of these documents are about "and" just as much as they are about "bias" - actually, the discrimination chapter is more about "and" than about "bias"! Oh, wait. That's not helpful, huh? Just as in your first example, where the system seemed to think "the" was the most important word in the document about your fast friend Harry, in this example "and" is considered highly relevant. Even at first glance, you can tell this isn't revelatory.

A good way to think of a term's inverse document frequency is this: How strange is it that this token is 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. Your first step toward topic analysis!

A term's IDF 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 "bias" in your current example, the answer is the same for both:

```
2 total documents / 2 documents contain "and" = 2/2 = 1  
2 total documents / 2 documents contain "bias" = 2/2 = 1
```

Not very interesting. So let's look at another word "black".

2 total documents / 1 document contains "black" = 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, disc_tokens]:  
...     if 'and' in doc:  
...         num_docs_containing_and += 1 #1
```

And let's grab the TF of "black" in the two documents:

```
>>> intro_tf['black'] = intro_counts['black'] / intro_total  
>>> disc_tf['black'] = disc_counts['black'] / disc_total
```

And finally, the IDF for all three. You'll store the IDFs in dictionaries per document like you did with TF:

```
>>> num_docs = 2  
>>> intro_idf = {}  
>>> disc_idf = {}  
>>> intro_idf['and'] = num_docs / num_docs_containing_and  
>>> disc_idf['and'] = num_docs / num_docs_containing_and  
>>> intro_idf['bias'] = num_docs / num_docs_containing_bias  
>>> disc_idf['bias'] = num_docs / num_docs_containing_bias  
>>> intro_idf['black'] = num_docs / num_docs_containing_black  
>>> disc_idf['black'] = num_docs / num_docs_containing_black
```

And then for the intro document you find:

```
>>> intro_tfidf = {}  
>>> intro_tfidf['and'] = intro_tf['and'] * intro_idf['and']  
>>> intro_tfidf['bias'] = intro_tf['bias'] * intro_idf['bias']  
>>> intro_tfidf['black'] = intro_tf['black'] * intro_idf['black']
```

And then for the history document:

```
>>> disc_tfidf = {}
>>> disc_tfidf['and'] = disc_tf['and'] * disc_idf['and']
>>> disc_tfidf['bias'] = disc_tf['bias'] * disc_idf['bias']
>>> disc_tfidf['black'] = disc_tf['black'] * disc_idf['black']
```

3.5.1 Return of Zipf

You're almost there. Let's say, though, you have a corpus of 1 million documents (maybe you're baby-Google), and someone searches for the word "cat", and in your 1 million documents you have exactly 1 document that contains the word "cat". The raw IDF of this is:

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

Let's imagine you have 10 documents with the word "dog" in them. Your IDF for "dog" is:

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

That's a big difference. Your friend Zipf would say that's **too** big, because it's likely to happen a lot. Zipf's Law showed that when you compare the frequencies of two words, like "cat" and "dog", even if they occur a similar number of times the more frequent word will have an exponentially higher frequency than the less frequent one. So Zipf's Law suggests that you scale all your word frequencies (and document frequencies) with the `log()` function, the inverse of `exp()`. This ensures that words with similar counts, such as "cat" and "dog", aren't vastly different in frequency. And this distribution of word frequencies will ensure that your TF-IDF scores are more uniformly distributed. So you should redefine IDF to be the log of the original probability of that word occurring in one of your documents. You'll also want to take the log of the term frequency as well.[\[126\]](#)

The base of log function is not important, since you just want to make the frequency distribution uniform, not to scale it within a particular numerical range.[\[127\]](#) If you use a base 10 log function, you'll get:

search: cat

Equation 3.3

$$\text{idf} = \log(1,000,000 / 1) = 6 \quad (3)$$

search: dog

Equation 3.4

$$\text{idf} = \log(1,000,000 / 10) = 5 \quad (4)$$

So now you're weighting the TF results of each more appropriately to their occurrences in language, in general.

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

Equation 3.5

$$\text{tf}(t, d) = \frac{\text{count}(t)}{\text{count}(d)} \quad (5)$$

Equation 3.6

$$\text{idf}(t, D) = \log\left(\frac{\text{number of documents}}{\text{number of documents containing } t}\right) \quad (6)$$

Equation 3.7

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) * \text{idf}(t, D) \quad (7)$$

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, you have a number. Something your 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 it assigns a numeric value to the importance of that word in the given document, given its usage across the entire corpus.

In some classes, all the calculations will be done in log space so that multiplications become additions and division becomes subtraction:

```
>>> log_tf = log(term_occurrences_in_doc) - \
...     log(num_terms_in_doc) #1
>>> log_log_idf = log(log(total_num_docs) - \
...     log(num_docs_containing_term)) #2
>>> log_tf_idf = log_tf + log_log_idf #3
```

This single number, the TF-IDF is the humble foundation of a simple search engine. As you've stepped from the realm of text firmly into the realm of numbers, it's time for some math. You won't likely ever have to implement the preceding formulas for computing TF-IDF. Linear algebra isn't necessary for full understanding of the tools used in natural language processing, but a general familiarity with how the formulas work can make their use more

intuitive.

3.5.2 Relevance ranking

As you saw earlier, you can easily compare two vectors and get their similarity, but you 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 your vectors will more thoroughly reflect the meaning, or topic, of the document. Back to your Harry example:

```
>>> document_tfidf_vectors = []
>>> for doc in docs:
...     vec = copy.copy(zero_vector) #1
...     tokens = [token.text for token in nlp(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 setup, you have K-dimensional vector representation of each document in the corpus. And now on to the hunt! Or search, in your case. From the previous section, you might remember how we defined similarity between vectors. Two vectors are considered similar if their cosine similarity is high, so you can find two similar vectors near each other if they maximize the cosine similarity.

Now you have all you need to do a basic TF-IDF based search. You 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 find the documents whose vectors have the highest cosine similarities to the query and return those as

the search results.

If you take your three documents about Harry, and make the query "How long does it take to get to the store?":

```
>>> query = "How long does it take to get to the store?"  
>>> query_vec = copy.copy(zero_vector) #1  
  
>>> tokens = [token.text for token in nlp(query.lower())]  
>>> token_counts = Counter(tokens)  
  
>>> for key, value in token_counts.items():  
...     docs_containing_key = 0  
...     for _doc in docs:  
...         if key in _doc.lower():  
...             docs_containing_key += 1  
...     if docs_containing_key == 0:  
...         continue  
...     tf = value / len(tokens)  
...     idf = len(docs) / docs_containing_key  
...     query_vec[key] = tf * idf  
>>> cosine_sim(query_vec, document_tfidf_vectors[0])  
0.5235048549676834  
>>> cosine_sim(query_vec, document_tfidf_vectors[1])  
0.0  
>>> cosine_sim(query_vec, document_tfidf_vectors[2])  
0.0
```

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

Actually, Google's search engine is safe from competition from us. You have to do an "index scan" of your 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*.^[128] You 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^[129] package and its source code.^[130]



Tip

In the preceding code, you dropped the keys that were not found in your pipeline's lexicon (vocabulary) to avoid a divide-by-zero error. But a better approach is to +1 the denominator of every IDF calculation, which ensures no denominators are zero. In fact this approach is so common it has a name, *additive smoothing* or "Laplace smoothing" [\[131\]](#)— will usually improve the search results for TF-IDF keyword-based searches.

3.5.3 Another vectorizer

Now that was a lot of code for things that have long since been automated. The scikit-learn package you used at the beginning of this chapter has a tool for TF-IDF too. Just as CountVectorizer you saw previously, it does tokenization, omits punctuation, and computes the tf-idf scores all in one.

Here's how you can use sklearn to build a TF-IDF matrix. You'll find the syntax similar to CountVectorizer.

Listing 3.4. Computing TF-IDF matrix using scikit-learn

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> corpus = docs
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> model = vectorizer.fit_transform(corpus) #1
>>> print(model.todense().round(2)) #2
[[ 0.16  0.    0.48  0.21  0.21  0.    0.25  0.21  0.    0.    0.    0.21  0.
  0.21  0.21]
 [ 0.37  0.    0.37  0.    0.    0.37  0.29  0.    0.37  0.37  0.    0.    0.4
  0.    0.    ]
 [ 0.    0.75  0.    0.    0.    0.29  0.22  0.    0.29  0.29  0.38  0.    0.
  0.    0.    ]]
```

With scikit-learn, in four lines you created a matrix of your three documents and the inverse document frequency for each term in the lexicon. It's very similar to the matrix you got from CountVectorizer, only this time it contains TF-IDF of each term, token, or word in your lexicon make up the columns of the matrix. On large texts this or some other pre-optimized TF-IDF model will save you scads of work.

3.5.4 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. [3.1](#) lists some of the ways you can normalize and smooth your term frequency weights.

Table 3.1. Alternative TF-IDF normalization approaches (Molino 2017) [\[a\]](#)

| Scheme | Definition |
|------------|--|
| None | $w_{ij} = f_{ij}$ |
| TD-IDF | $w_{ij} = \log(f_{ij}) \times \log\left(\frac{N}{n_j}\right)$ |
| TF-ICF | $w_{ij} = \log(f_{ij}) \times \log\left(\frac{N}{f_j}\right)$ |
| Okapi BM25 | $w_{ij} = \frac{\frac{f_{ij}}{f_j}}{0.5 + 1.5 \times \frac{f_j}{\sum_j} + f_{ij}} \log \frac{N - n_j + 0.5}{f_{ij} + 0.5}$ |
| ATC | $stem : [w_{ij} = \frac{\left(0.5 + 0.5 \times \frac{f_{ij}}{\max_f}\right) \log\left(\frac{N}{n_j}\right)}{\sqrt{\sum_{i=1}^N \left[\left(0.5 + 0.5 \times \frac{f_{ij}}{\max_f}\right) \log\left(\frac{N}{n_j}\right)\right]^2}}]$ |
| LTU | $w_{ij} = \frac{(\log(f_{ij}) + 1.0) \log\left(\frac{N}{n_j}\right)}{0.8 + 0.2 \times f_j \times \frac{j}{\sum_j}}$ |
| MI | $w_{ij} = \log \frac{P(t_{ij} \text{ vert } c_j)}{P(t_{ij})P(c_j)}$ |
| PosMI | $w_{ij} = \max(0, MI)$ |
| | |

| | |
|------------------|--|
| T-Test | $w_{ij} = \frac{P(t_{ij} \text{ vert } c_j) - P(t_{ij})P(c_j)}{\sqrt{P(t_{ij})P(c_j)}}$ |
| chi ² | See section 4.3.5 of <i>From Distributional to Semantic Similarity</i> (https://www.era.lib.ed.ac.uk/bitstream/handle/1842/563/IP030023.pdf) by James Richard Curran |
| Lin98a | $W_{ij} = \frac{f_{ij} \times f}{f_j \times f_j}$ |
| Lin98b | $w_{ij} = -1 \times \log \frac{n_j}{N}$ |
| Gref94 | $w_{ij} = \frac{\log f_{ij} + 1}{\log n_j + 1}$ |

[a] *Word Embeddings Past, Present and Future* by Piero Molino at AI with the

Search engines (information retrieval systems) match keywords (term) between queries and documents in a corpus. If you’re building a search engine and want to provide documents that are likely to match what your users are looking for, you should spend some time investigating the alternatives described by Piero Molino in figure 3.7.

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

3.5.5 Okapi BM25

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

document length itself.

```
q_idf * dot(q_tf, d_tf[i]) * 1.5 / (dot(q_tf, d_tf[i])) + .25 + .7
```

You can optimize your pipeline by choosing the weighting scheme that gives your users the most relevant results. But if your corpus isn't too large, you might consider forging ahead with us into even more useful and accurate representations of the meaning of words and documents.

[126] Gerard Salton and Chris Buckley first demonstrated the usefulness of log scaling for information retrieval in their paper Term Weighting Approaches in Automatic Text Retrieval

(<https://ecommons.cornell.edu/bitstream/handle/1813/6721/87-881.pdf>).

[127] Later we show you how to normalize the TF-IDF vectors after all the TF-IDF values have been calculated using this log scaling.

[128] See the web page titled "Inverted index - Wikipedia"

(https://en.wikipedia.org/wiki/Inverted_index).

[129] See the web page titled "Whoosh : PyPI"

(<https://pypi.python.org/pypi/Whoosh>).

[130] See the web page titled "GitHub - Mplsbeb/whoosh: A fast pure-Python search engine" (<https://github.com/Mplsbeb/whoosh>).

[131] See the web page titled "Additive smoothing - Wikipedia"

(https://en.wikipedia.org/wiki/Additive_smoothing).

3.6 Using TF-IDF for your bot

In this chapter, you learned how TF-IDF can be used to represent natural language documents with vectors, find similarities between them, and perform keyword search. But if you want to build a chatbot, how can you use those capabilities to make your first intelligent assistant?

Actually, many chatbots rely heavily on a search engine. And some chatbots

use their search engine as their only algorithm for generating responses. You just need to take one additional step to turn your simple search index (TF-IDF) into a chatbot. To make this book as practical as possible, every chapter will show you how to make your bot smarter using the skills you picked up in that chapter.

In this chapter, you’re going to make your chatbot answer data science questions. The trick is simple: you’re store your training data in pairs of questions and appropriate responses. Then you can use TF-IDF to search for a question most similar to the user input text. Instead of returning the most similar statement in your database, you return the response associated with that statement. And with that, you’re chatting!

Let’s do it step by step. First, let’s load our data. You’ll use the corpus of data science questions that Hobson was asked by his mentees in the last few years. They are located in the qary repository:

```
>>> DS_FAQ_URL = ('https://gitlab.com/tangibleai/qary/-/raw/master'
                   'src/qary/data/faq/faq-python-data-science-cleaning.csv')
>>> qa_dataset = pd.read_csv(DS_FAQ_URL)
```

Next, let’s create TF-IDF vectors for the questions in our dataset. You’ll use the `scikit-learn` `TfidfVectorizer` class you’ve seen in the previous section.

We’re now ready to implement the question-answering itself. Your bot will reply to the user’s question by using the same vectorizer you trained on the dataset, and finding the most similar questions.

And your first question-answering chatbot is ready! Let’s try to ask it a couple of data science questions:

3.7 What’s next

Now that you can convert natural language text to numbers, you can begin to manipulate them and compute with them. Numbers firmly in hand, in the next chapter you’ll refine those numbers to try to represent the **meaning** or **topic** of natural language text instead of just its words. In subsequent chapters, we show you how to implement a semantic search engine that finds

documents that "mean" something similar to the words in your query rather than just documents that use those exact words from your query. Semantic search is much better than anything TF-IDF weighting and stemming and lemmatization can ever hope to 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 your 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. TF-IDFs are just the first stage in your pipeline, a basic set of features you'll extract from text. In the next chapter, you will compute topic vectors from your TF-IDF vectors. Topic vectors are an even better representation of the meaning of a document than 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 deep learning embeddings of the meaning of words and documents in later chapters.

3.8 Review

1. What makes a Counter vector (`CountVectorizer` output) different from a `collections.Counter` object?
2. Can you use `TfidfVectorizer` on a large corpus (more than 1M documents) with a huge vocabulary (more than 1M tokens)? What problems do you expect to encounter?
3. Think of an example of corpus or task where term frequency (TF) will perform better than TF-IDF.
4. We mentioned that bag of character n-grams can be used for language recognition tasks. How would an algorithm that uses character n-grams to distinguish one language from another work?
5. What disadvantages of TF-IDF have you seen throughout the chapter? Can you come up with additional ones that weren't mentioned?
6. How would you use TF-IDF as a base to create a better search engine?

3.9 Summary

- Any web-scale search engine with millisecond response times has the power of a TF-IDF term document matrix hidden under the hood.
- Zipf's law can help you predict the frequencies of all sorts of things including words, characters, and people.
- Term frequencies must be weighted by their inverse document frequency to ensure the most important, most meaningful words are given the heft they deserve.
- Bag-of-words / Bag-of-ngrams and TF-IDF are the most basic algorithms to represent natural language documents with a vector of real numbers.
- Euclidean distance and similarity between pairs of high dimensional vectors doesn't adequately represent their similarity for most NLP applications.
- Cosine distance, the amount of "overlap" between vectors, can be calculated efficiently just multiplying the elements of normalized vectors together and summing up those products.
- Cosine distance is the go-to similarity score for most natural language vector representations.

4 Finding meaning in word counts (semantic analysis)

This chapter covers

- Analyzing semantics (meaning) to create topic vectors
- Semantic search using the semantic similarity between topic vectors
- Scalable semantic analysis and semantic search for large corpora
- Using semantic components (topics) as features in your NLP pipeline
- Navigating high-dimensional vector spaces

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

The TF-IDF vectors (term frequency – inverse document frequency vectors) from chapter 3 helped you estimate the importance of words in a chunk of text. You used TF-IDF vectors and matrices to tell you how important each word is to the overall meaning of a bit of text in a document collection. These TF-IDF "importance" scores worked not only for words, but also for short sequences of words, *n*-grams. They are great for searching text if you know the exact words or *n*-grams you're looking for. But they also have certain limitations. Often, you need a representation that takes not just counts of words, but also their *meaning*.

Researchers have discovered several ways to represent the meaning of words using their co-occurrence with other words. You will learn about some of them, like *Latent Semantic Analysis*(LSA) and *Latent Dirichlet Allocation*, in this chapter. These methods create *semantic* or *topic* vectors to represent words and documents. [\[132\]](#) You will use your weighted frequency scores from TF-IDF vectors, or the bag-of-words (BOW) vectors that you learned to create in the previous chapter. These scores and the correlations between them, will help you compute the topic "scores" that make up the dimensions

of your topic vectors.

Topic vectors will help you do a lot of interesting things. They make it possible to search for documents based on their meaning—*semantic search*. Most of the time, semantic search returns search results that are much better than keyword search. Sometimes semantic search returns documents that are exactly what the user is searching for, even when they can't think of the right words to put in the query.

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

And lastly, you will be able to compare any two statements or documents and tell how "close" they are in *meaning* to each other.



Tip

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

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

4.1 From word counts to topic scores

You know how to count the frequency of words, and to score the importance of words in a TF-IDF vector or matrix. But that's not enough. Let's look at what problems that might create, and how to approach representing the

meaning of your text rather than just individual term frequencies.

4.1.1 The limitations of TF-IDF vectors and lemmatization

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

In chapter 2, you normalized word endings so that words that differed only in their last few characters were collected together under a single token. You used normalization approaches such as stemming and lemmatization to create small collections of words with similar spellings, and often similar meanings. You labeled each of these small collections of words, with their lemma or stem, and then you processed these new tokens instead of the original words.

This lemmatization approach kept similarly *spelled* [\[133\]](#) words together in your analysis, but not necessarily words with similar meanings. And it definitely failed to pair up most synonyms. Synonyms usually differ in more ways than just the word endings that lemmatization and stemming deal with. Even worse, lemmatization and stemming sometimes erroneously lump together antonyms, words with opposite meaning.

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

For example, the TF-IDF vector for this chapter in *NLPIA*, the chapter that you're reading right now, may not be at all close to similar-meaning passages in university textbooks about latent semantic indexing. But that's exactly what this chapter is about, only we use modern and colloquial terms in this

chapter. Professors and researchers use more consistent, rigorous language in their textbooks and lectures. Plus, the terminology that professors used a decade ago has likely evolved with the rapid advances of the past few years. For example, terms such "latent semantic indexing" were more popular than the term "latent semantic analysis" that researchers now use.[\[134\]](#)

So, different words with similar meaning pose a problem for TF-IDF. But so do words that look similar, but mean very different things. Even formal English text written by an English professor can't avoid the fact that most English words have multiple meanings, a challenge for any new learner, including machine learners. This concept of words with multiple meanings is called *polysemy*.

Here are some ways in which polysemy can affect the semantics of a word or statement.

- Homonyms—Words with the same spelling and pronunciation, but different meanings (For example: *The band was playing old Beatles' songs. Her hair band was very beautiful.*)
- Homographs—Words spelled the same, but with different pronunciations and meanings.(For example: *I object to this decision. I don't recognize this object.*)
- Zeugma—Use of two meanings of a word simultaneously in the same sentence (For example: *Mr. Pickwick took his hat and his leave.*)

You can see how all of these phenomena will lower TF-IDF's performance, by making the TF-IDF vectors of sentences with similar words but different meanings being more similar to each other than they should be. To deal with these challenges, we need a more powerful tool.

4.1.2 Topic vectors

When you do math on TF-IDF vectors, such as addition and subtraction, these sums and differences only tell you about the frequency of word uses in the documents whose vectors you combined or differenced. That math doesn't tell you much about the "meaning" behind those words. You can compute word-to-word TF-IDF vectors (word co-occurrence or correlation

vectors) by just multiplying your TF-IDF matrix by itself. But "vector reasoning" with these sparse, high-dimensional vectors doesn't work well. You when you add or subtract these vectors from each other, they don't represent an existing concept or word or topic well.

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

Essentially, what you'll be doing when creating these new vectors is defining a new space. When you represent words and documents by TF-IDF or bag-of-words vectors, you are operating in a space defined by the words, or terms occurring in your document. There is a dimension for each term - that's why you easily reach several thousand dimensions. And every term is "orthogonal" to every other term - when you multiply the vector signifying one word with a vector representing another one, you always get a zero, even if these words are synonyms.

The process of topic modelling is finding a space with fewer dimensions, so that words that are close semantically are aligned to similar dimensions. We will call these dimensions *topics*, and the vectors in the new space *topic vectors*. You can have as many topics as you like. Your topic space can have just one dimension, or thousands of dimensions.

You can add and subtract the topic vectors you'll compute in this chapter just like any other vector. Only this time the sums and differences mean a lot more than they did with TF-IDF vectors. The distance or *similarity* between topic vectors is useful for things like finding documents about similar subjects or semantic search.

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

Coming up with a numerical representation of the semantics (meaning) of words and sentences can be tricky. This is especially true for "fuzzy" languages like English, which has multiple dialects and many different interpretations of the same words.

Keeping these challenges in mind, can you imagine how you might squash a TF-IDF vector with one million dimensions (terms) down to a vector with 10 or 100 dimensions (topics)? This is like identifying the right mix of primary colors to try to reproduce the paint color in your apartment so you can cover over those nail holes in your wall.

You'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. You might even weight them for how important they are to the topic, how much you'd like each word to contribute to the "mix." And you could have negative weights for words that reduce the likelihood that the text is about that topic.

4.1.3 Thought experiment

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

Let's say you're processing some sentences about pets in Central Park in New York City (NYC). Let's create three topics: one about pets, one about animals, and another about cities. Call these topics "petness", "animalness", and "cityness." So your "petness" topic about pets will score words like "cat" and "dog" significantly, but probably ignore words like "NYC" and "apple." The "cityness" topic will ignore words like "cat" and "dog" but might give a little weight to "apple", just because of the "Big Apple" association.

If you "trained" your topic model like this, without using a computer, just your common sense, you might come up with some weights like those in Listing 4.1.

Listing 4.1. Sample weights for your topics

```

>>> import numpy as np

>>> topic = {}
>>> tfidf = dict(list(zip('cat dog apple lion NYC love'.split(),
...     np.random.rand(6))))    #1
>>> topic['petness'] = (.3 * tfidf['cat'] +\
...             .3 * tfidf['dog'] +\
...             0 * tfidf['apple'] +\
...             0 * tfidf['lion'] -\
...             .2 * tfidf['NYC'] +\
...             .2 * tfidf['love'])    #2
>>> topic['animalness'] = (.1 * tfidf['cat'] +\
...             .1 * tfidf['dog'] -\
...             .1 * tfidf['apple'] +\
...             .5 * tfidf['lion'] +\
...             .1 * tfidf['NYC'] -\
...             .1 * tfidf['love'])
>>> topic['cityness'] = ( 0 * tfidf['cat'] -\
...             .1 * tfidf['dog'] +\
...             .2 * tfidf['apple'] -\
...             .1 * tfidf['lion'] +\
...             .5 * tfidf['NYC'] +\
...             .1 * tfidf['love'])

```

In this thought experiment, we added up the word frequencies that might be indicators of each of your topics. We weighted the word frequencies (TF-IDF values) by how likely the word is associated with a topic. Note that these weights can be negative as well for words that might be talking about something that is in some sense the opposite of your topic.

Note this is not a real algorithm, or example implementation, just a thought experiment. You're just trying to figure out how you can teach a machine to think like you do. You arbitrarily chose to decompose your words and documents into only three topics ("petness", "animalness", and "cityness"). And your vocabulary is limited, it has only six words in it.

The next step is to think through how a human might decide mathematically which topics and words are connected, and what weights those connections should have. Once you decided on three topics to model, you then had to then decide how much to weight each word for those topics. You blended words in proportion to each other to make your topic "color mix." The topic modeling transformation (color mixing recipe) is a 3 x 6 matrix of

proportions (weights) connecting three topics to six words. You multiplied that matrix by an imaginary 6×1 TF-IDF vector to get a 3×1 topic vector for that document.

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

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

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

As you read the rest of the weighted sums in Listing 4.1, try to guess how we came up with these weights for these three topics and six words. You may have a different "corpus" in your head than the one we used in our heads. So you may have a different opinion about the "appropriate" weights for these words. How might you change them? What could you use as an objective measure of these proportions (weights)? We'll answer that question in the next section.



Note

We chose a signed weighting of words to produce the topic vectors. This

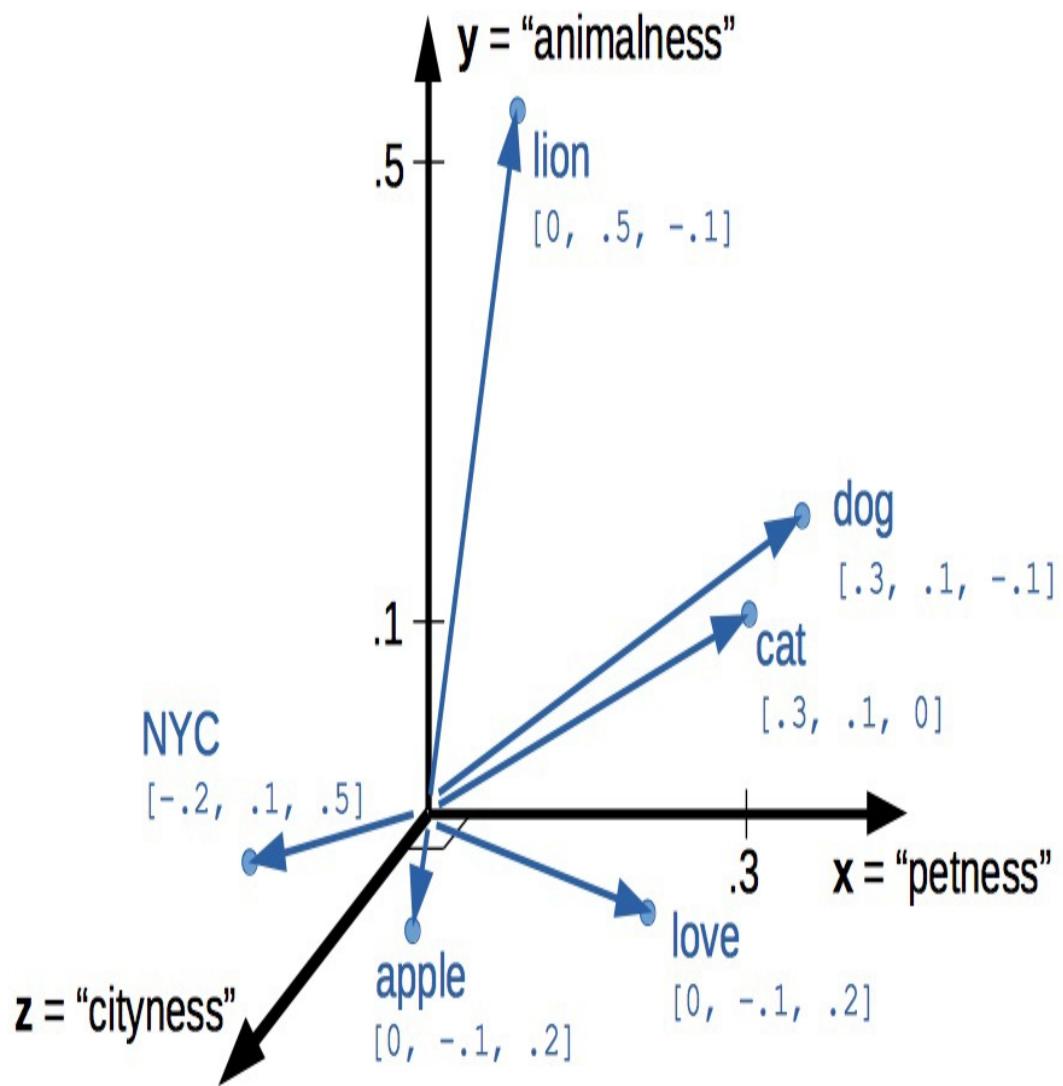
allows you to use negative weights for words that are the "opposite" of a topic. And because you're doing this manually by hand, we chose to normalize your topic vectors by the easy-to-compute L¹-norm (meaning the sum of absolute values of the vector dimensions equals 1). Nonetheless, the real LSA you'll use later in this chapter normalizes topic vectors by the more useful L²-norm. We'll cover the different norms and distances later in this chapter.

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

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

These six word-topic vectors shown in Figure 4.1, one for each word, represent the meanings of your six words as 3D vectors.

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



Earlier, the vectors for each topic, with weights for each word, gave you 6-D vectors representing the linear combination of words in your three topics. Now, you hand-crafted a way to represent a document by its topics. If you just count up occurrences of these six words and multiply them by your weights, you get the 3D topic vector for any document. And 3D vectors are fun because they're easy for humans to visualize. You can plot them and share insights about your corpus or a particular document in graphical form.

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

The documents in your corpus might use many more words, but this particular topic vector model will only be influenced by the use of these six words. You could extend this approach to as many words as you had the patience (or an algorithm) for. As long as your model only needed to separate documents according to three different dimensions or topics, your vocabulary could keep growing as much as you like. In the thought experiment, you compressed six dimensions (TF-IDF normalized frequencies) into three dimensions (topics).

This subjective, labor-intensive approach to semantic analysis relies on human intuition and common sense to break documents down into topics. Common sense is hard to code into an algorithm.^[135] And obviously this isn't suitable for a machine learning pipeline. Plus it doesn't scale well to more topics and words.

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

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

4.1.4 Algorithms for scoring topics

You still need an algorithmic way to determine these topic vectors, or to derive them from vectors you already have - like TF-IDF or bag-of-words

(BOW) 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 you can estimate what a word or morpheme [\[136\]](#) signifies. In 1957 he gave you a clue about how to compute the topics for words. Firth wrote:

You shall know a word by the company it keeps.

-- J. R. Firth 1957

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

In the next sections, you'll see 2 algorithms for creating these topic vectors. The first one, *Latent Semantic Analysis* (LSA), is applied to your TF-IDF matrix to gather up words into topics. It works on bag-of-words vectors, too, but TF-IDF vectors give slightly better results. LSA optimizes these topics to maintain diversity in the topic dimensions; when you use these new topics instead of the original words, you still capture much of the meaning (semantics) of the documents. The number of topics you need for your model to capture the meaning of your documents is far less than the number of words in the vocabulary of your TF-IDF vectors. So LSA is often referred to as a dimension reduction technique. LSA reduces the number of dimensions you need to capture the meaning of your documents. [\[137\]](#)

The other algorithm we'll cover is called *Latent Dirichlet Allocation*, often shortened to LDA. Because we use LDA to signify Latent Discriminant Analysis classifier in this book, we will shorten Latent Dirichlet Allocation to LDiA instead.

LDiA takes the math of LSA in a different direction. It uses a nonlinear statistical algorithm to group words together. As a result, it generally takes much longer to train than linear approaches like LSA. Often this makes LDiA less practical for many real-world applications, and it should rarely be the first approach you try. Nonetheless, the statistics of the topics it creates

sometimes more closely mirror human intuition about words and topics. So LDiA topics will often be easier for you to explain to your boss. It is also more useful for some single-document problems such as document summarization.

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

[132] We use term "topic vector" in this chapter about topic analysis and we use the term "word vector" in chapter 6 about Word2vec. Formal NLP texts such as the NLP bible by Jurafsky and Martin (<https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf#chapter.15>) use "topic vector." Others, like the authors of Semantic Vector Encoding and Similarity Search (<https://arxiv.org/pdf/1706.00957.pdf>), use the term "semantic vector."

[133] Both stemming and lemmatization remove or alter the word endings and prefixes, the last few characters of a word. Edit-distance calculations are better for identifying similarly spelled (or misspelled) words

[134] I love Google Ngram Viewer for visualizing trends like this one: (<http://mng.bz/ZoyA>).

[135] Doug Lenat at Stanford is trying to do just that, code common sense into an algorithm. See the Wired Magazine article "Doug Lenat's Artificial Intelligence Common Sense Engine" (<https://www.wired.com/2016/03/doug-lenat-artificial-intelligence-common-sense-engine>).

[136] A *morpheme* is the smallest meaningful parts of a word. See Wikipedia article "Morpheme" (<https://en.wikipedia.org/wiki/Morpheme>).

[137] The wikipedia page for topic models has a video that shows the intuition behind LSA.
https://upload.wikimedia.org/wikipedia/commons/7/70/Topic_model_scheme.mp4

4.2 The challenge: detecting toxicity

To see the power of topic modeling, we'll try to solve a real problem: recognizing toxicity in Wikipedia comments. This is a common NLP task that content and social media platforms face nowadays. Throughout this chapter, we'll work on a dataset of Wikipedia discussion comments, [\[138\]](#) which we'll want to classify into two categories - toxic and non-toxic. First, let's load our dataset and take a look at it:

Listing 4.2. The toxic comment dataset

```
>>> import pandas as pd
>>> pd.options.display.width = 120      #1
>>>
>>> DATA_DIR = ('https://gitlab.com/tangibleai/nlpia/-/raw/master
...           'src/nlpia/data')

>>> url= DATA_DIR + '/toxic_comment_small.csv'
>>>
>>> comments = pd.read_csv(url)
>>> index = ['comment{}{}'.format(i, '''*j) for (i,j) in zip(rang
>>> comments = pd.DataFrame(comments.values, columns=comments.col
>>> mask = comments.toxic.astype(bool).values
>>> comments['toxic'] = comments.toxic.astype(int)
>>> len(comments)
5000
>>> comments.toxic.sum()
650
>>> comments.head(6)
text    tox
comment0  you have yet to identify where my edits violat...
comment1  "\n as i have already said,wp:rfc or wp:ani. (...
comment2  your vote on wikiquote simple english when it ...
comment3  your stalking of my edits i've opened a thread...
comment4! straight from the smear site itself. the perso...
comment5  no, i can't see it either - and i've gone back...
```

So you have 5,000 comments, and 650 of them are labeled with the binary class label "toxic."

Before you dive into all the fancy dimensionality reduction stuff, let's try to solve our classification problem using vector representations for the messages that you are already familiar with - TF-IDF. But what *model* will you choose to classify the messages? To decide, let's look at the TF-IDF vectors first.

Listing 4.3. Creating TF-IDF vectors for the SMS dataset

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> import spacy
>>> nlp = spacy.load("en_core_web_sm")
>>>
>>> def spacy_tokenize(sentence):
...     return [token.text for token in nlp(sentence.lower())]
>>>
>>> tfidf_model = TfidfVectorizer(tokenizer=spacy_tokenize)
>>> tfidf_docs = tfidf_model.fit_transform(\n...     raw_documents=comments.text).toarray()
>>> tfidf_docs.shape
(5000, 19169)
```

The spaCy tokenizer gave you 19,169 words in your vocabulary. You have almost 4 times as many words as you have messages. And you have almost 30 times as many words as toxic comments. So your model will not have a lot of information about the words that will indicate whether a comment is toxic or not.

You have already met at least one classifier in this book - Naive Bayes in chapter 2. Usually, a Naive Bayes classifier will not work well when your vocabulary is much larger than the number of labeled examples in your dataset. So we need something different this time.

4.2.1 Latent Discriminant Analysis classifier

In this chapter, we're going to introduce a classifier that is based on an algorithm called Latent Discriminant Analysis (LDA). LDA is one of the most straightforward and fast classification models you'll find, and it requires fewer samples than the fancier algorithms.

The input to LDA will be a labeled data - so we need not just the vectors representing the messages, but their class too. In this case, we have two classes - toxic comments and non-toxic comments. LDA algorithm uses some math that beyond the scope of this book, but in case of two classes, its implementation is pretty intuitive.

In essence, this is what LDA algorithm does when faced with a two-class

problem:

1. It finds a line, or axis, in your vector space, such that if you project all the vectors (data points) in the space on that axis, the two classes would be as separated as possible.
2. It projects all the vectors on that line.
3. It predicts the probability of each vector to belong to one of two classes, according to a *cutoff* point between the two classes.

Surprisingly, in the majority of cases, the line that maximizes class separation is very close to the line that connects the two *centroids* [139] of the clusters representing each class.

Let's perform manually this approximation of LDA, and see how it does on our dataset.

```
>>> mask = comments.toxic.astype(bool).values      #1
>>> toxic_centroid = tfidf_docs[mask].mean(axis=0)  #2
>>> nontoxic_centroid = tfidf_docs[~mask].mean(axis=0)  #3

>>> centroid_axis = toxic_centroid - nontoxic_centroid
>>> toxicity_score = tfidf_docs.dot(centroid_axis)
>>> toxicity_score.round(3)
array([-0.008, -0.022, -0.014, ..., -0.025, -0.001, -0.022])
```

This raw `toxicity_score` is the distance along the line from the nontoxic centroid to the toxic centroid. You calculated that score by projecting each TF-IDF vector onto that line between the centroids using the dot product. And you did those 5,000 dot products all at once in a "vectorized" numpy operation. This can speed things up 100 times compared to a Python for loop.

You have just one step left in our classification. You need to transform our score into the actual class prediction. Ideally, you'd like your score to range between 0 and 1, like a probability. Once you have the scores normalized, you can deduce the classification from the score based on a cutoff - here, we went with a simple 0.5 You can use `sklearn MinMaxScaler` to perform the normalization:

```
>>> from sklearn.preprocessing import MinMaxScaler
```

```

>>> comments['manual_score'] = MinMaxScaler().fit_transform(\n...     toxicity_score.reshape(-1,1))\n>>> comments['manual_predict'] = (comments.manual_score > .5).ast\n>>> comments['toxic manual_predict manual_score'].split()].round(2\n    toxic  manual_predict  manual_score\ncomment0      0            0        0.41\ncomment1      0            0        0.27\ncomment2      0            0        0.35\ncomment3      0            0        0.47\ncomment4!     1            0        0.48\ncomment5      0            0        0.31

```

That looks pretty good. Almost all of the first six messages were classified correctly. Let's see how it did on the rest of the training set.

```

>>> (1. - (comments.toxic - comments.manual_predict).abs().sum()\n0.895

```

Not bad! 89.5% of the messages were classified correctly with this simple "approximate" version of LDA. How will the "full" LDA do? We'll use the implementation of LDA from scikitlearn.

```

>>> from sklearn.discriminant_analysis import LinearDiscriminantA\n>>> lda_tfidf = LDA(n_components=1)\n>>> lda_tfidf = lda_tfidf.fit(tfidf_docs, comments['toxic'])\n>>> comments['tfidf_predict'] = lda_tfidf.predict(tfidf_docs)\n>>> round(float(lda_tfidf.score(tfidf_docs, comments['toxic']))),\n0.999

```

99.9%! Almost perfect accuracy. Does it mean you don't need to have topic modeling at all? Maybe you already achieved everything you need.

We know that you won't be deceived by our trick questions, and have already figured out the trap. The reason for this perfect 99.9% result is that we haven't separated out a test set. This A+ score is on test "questions" that the classifier has already "seen."



Tip

Note the class methods you used in order to train and make predictions with our model. Every model in `sklearn` has those methods: `fit()`, `predict()`

(and its cousin `predict_proba()` that gives you the probability score). That makes it easier to swap and replace models with each other as you try them when solving your machine learning problems.

Let's see how our classifier does in more realistic situation. You'll split your comment dataset into 2 parts - training set and testing set. (As you can imagine, there is a function in `sklearn` just for that!) And you'll see how the classifier performs on the messages it wasn't trained on.

Listing 4.4. LDA model performance with train-test split

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(tfidf_doc
...     comments.toxic.values, test_size=0.5, random_state=271828
>>> lda_tfidf = LDA(n_components=1)
>>> lda = lda_tfidf.fit(X_train, y_train)      #1
>>> round(float(lda.score(X_train, y_train)), 3)
0.999
>>> round(float(lda.score(X_test, y_test)), 3)
0.554
```

The training set accuracy for your TF-IDF based model is almost perfect. But the test set accuracy is 0.55 - a bit better than flipping a coin. And test set accuracy is the only accuracy that counts. This is exactly what topic modeling will help you. It will allow you to generalize your models from a small training set so it still works well on messages using different combinations of words (but similar topics).



Tip

Note the `random_state` parameter for the `train_test_split`. The algorithm for `train_test_split()` are stochastic. So each time you run it you will get different results and different accuracy values. If you want to make your pipeline repeatable, look for the `seed` argument for these models and dataset splitters. You can set the seed to the same value with each run to get reproducible results.

Let's look a bit deeper at how our LDA model did, using a tool called *confusion matrix*. It will tell you the comments that it labeled as toxic that

weren't toxic (false positives), and the ones that were labeled as non-toxic that should have been labeled toxic (false negatives). Here's how you do it with an `sklearn` function:

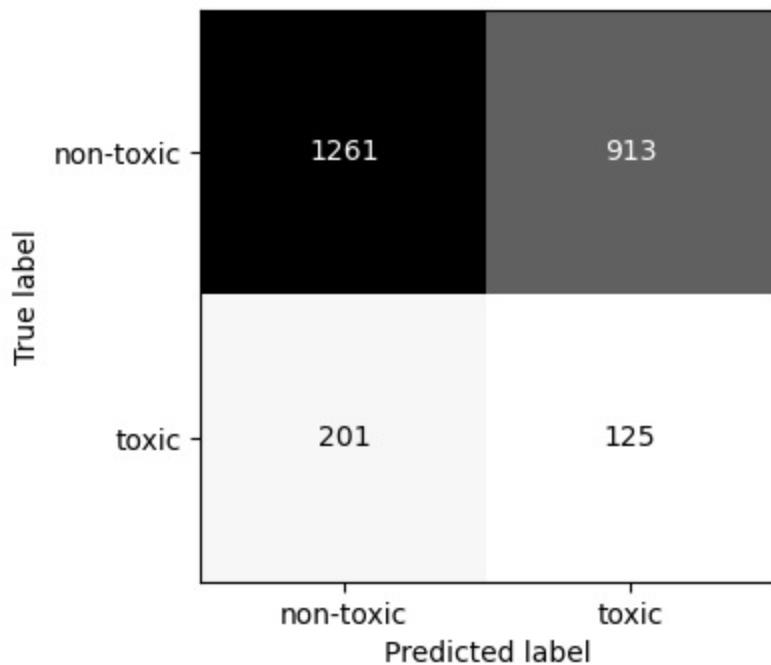
```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_test, lda.predict(X_test))
array([[1261,  913],
       [ 201,  125]], dtype=int64)
```

Hmmm. It's not exactly clear what's going on here. Fortunately, `sklearn` have taken into account that you might need a more visual way to present your confusion matrix to people, and included a function just for that. Let's try it out:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.metrics import plot_confusion_matrix
>>> plot_confusion_matrix(lda, X_test, y_test, cmap="Greys",
...                         display_labels=['non-toxic', 'toxic'], colorba
>>> plt.show()
```

You can see the resulting `matplotlib` plot on Fig. 4.3. Now, that's a bit clearer. From this plot, you can see what's problematic with your model's performance.

Figure 4.2. Confusion matrix of TF-IDF based classifier



First of all, out of 326 comments in the test set that were actually toxic, the model was able to identify correctly only 125 - that's 38.3%. This measure (how many of the instances of the class we're interested in the model was able to identify), is called *recall*, or *sensitivity*. On the other hand, out of 1038 comments the model labeled as toxic, only 125 are truly toxic comments. So the "positive" label is only correct in 12% of cases. This measure is called *precision*.[\[140\]](#)

You can already see how precision and recall give us more information than model accuracy. For example, imagine that instead of using machine learning models, you decided to use to use a deterministic rule and just label all the comments as non-toxic. As about 13% of comments in our dataset are actually toxic, this model will have accuracy of 0.87 - much better than the last LDA model you trained! However, its recall is going to be 0 - it doesn't help you at all in our task, which is to identify toxic messages.

You might also realize that there is a tradeoff between these two measures. What if you went with another deterministic rule and labeled all the comments as toxic? In this case, your recall would be perfect, as you would correctly classify all the toxic comments. However, the precision will suffer,

as most of the comments labeled as toxic will actually be perfectly OK.

Depending on your use case, you might decide to prioritize either precision or recall over the other. But in a lot of cases, you would want both of them to be reasonably good.

In this case, you're likely to use the F_1 score - a harmonic mean of precision and recall. Higher precision and higher recall both lead to a higher F_1 score, making it easier to benchmark your models with just one metric.^[141]

You can learn more about analyzing your classifier's performance in Appendix D. For now, we will just note this model's F_1 score before we continue on.

4.2.2 Going beyond linear

LDA is going to serve you well in many circumstances. However, it still has some assumptions that will cause the classifier to underperform when these assumptions are not fulfilled. For example, LDA assumes that the feature covariance matrices for all of your classes are the same. That's a pretty strong assumption! As a result of it, LDA can only learn linear boundaries between classes.

If you need to relax this assumption, you can use a more general case of LDA called *Quadratic Discriminant Analysis*, or QDA. QDA allows different covariance matrices for different classes, and estimates each covariance matrix separately. That's why it can learn quadratic, or curved, boundaries.

^[142] That makes it more flexible, and helps it to perform better in some cases.

^[138] The larger version of this dataset was a basis for a Kaggle competition in 2017(<https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge>), and was released by Jigsaw under CC0 license.

^[139] A centroid of a cluster is a point whose coordinates are the average of the coordinates of all the points in that cluster.

^[140] To gain some more intuition about precision and recall, Wikipedia's

article (https://en.wikipedia.org/wiki/Precision_and_recall) has some good visuals.

[141] You can read more about the reasons *not* to use F_1 score in some cases, and about alternative metrics in the Wikipedia article:
<https://en.wikipedia.org/wiki/F-score>

[142] You can see a visual example of the two estimator's in `scikit-learn's documentation: https://scikit-learn.org/dev/modules/lda_qda.html

4.3 Reducing dimensions

Before we dive into LSA, let's take a moment to understand what, conceptually, it does to our data. The idea behind LSA's approach to topic modeling is *dimensionality reduction*. As its name suggests, dimensionality reduction is a process in which we find a lower-dimensional representation of data that retains as much information as possible.

Let's examine this definition and understand what it means. To give you an intuition, let's step away from NLP for a moment and switch to more visual examples. First, what's a lower-dimension representation of data? Think about taking a 3-D object (like your sofa) and representing it in 2-D space. For example, if you shine a light behind your sofa in a dark room, its shadow on the wall is its two-dimensional representation.

Why would we want such a representation? There might be many reasons. Maybe we don't have capacity to store or transmit the full data as it is. Or maybe we want to visualize our data to understand it better. You already saw the power of visualizing your data points and clustering them when we talked about LDA. But our brain can't really work with more than 2 or 3 dimensions - and when we're dealing with real-world data, especially natural language data, our datasets might have hundreds or even thousands of dimensions. Dimensionality reduction tools like PCA are very useful when we want to simplify and visually map our dataset.

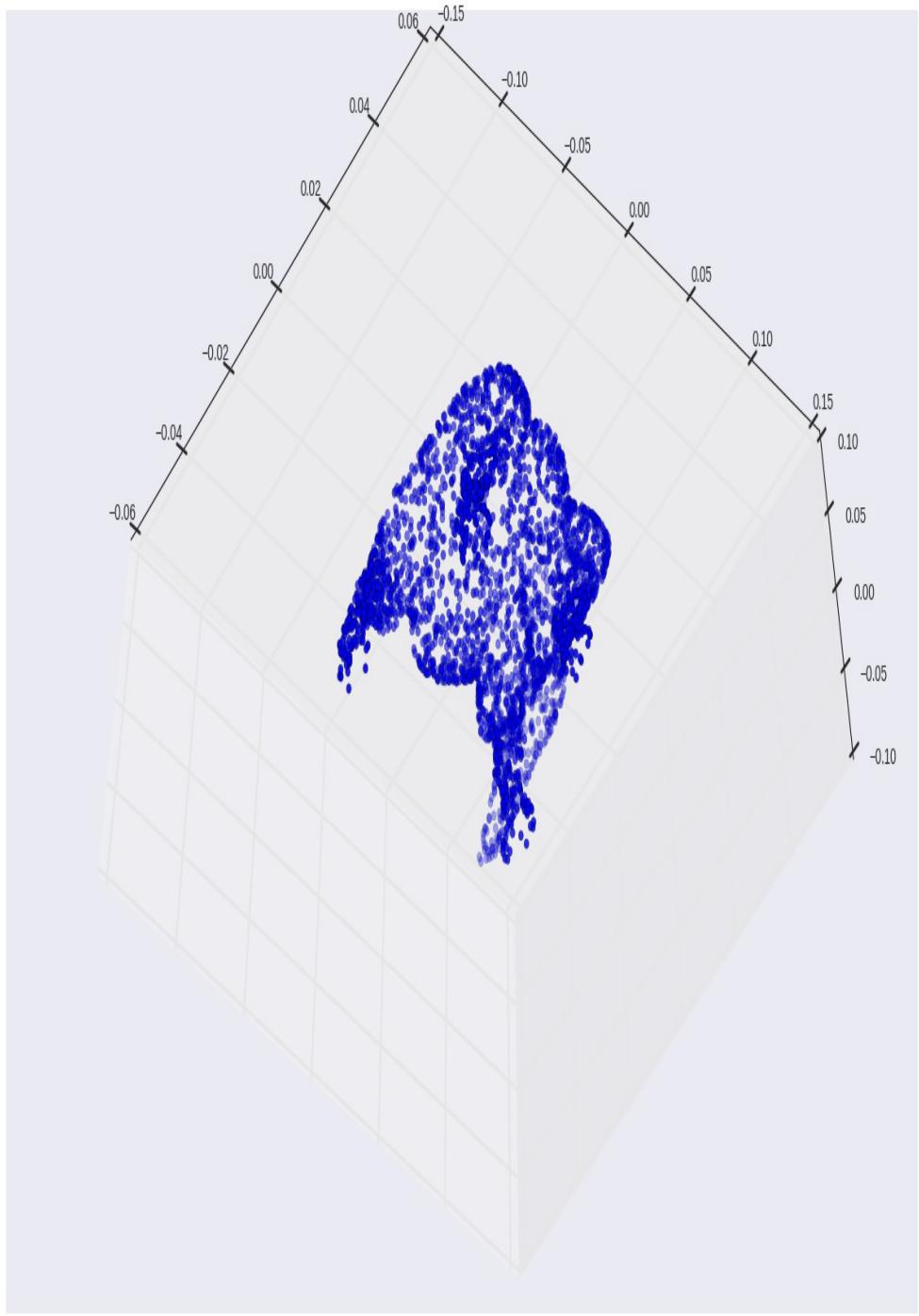
Another important reason is curse of dimensionality we briefly mentioned in chapter 3. Sparse, multidimensional data is harder to work with, and

classifiers trained on it are more prone to overfitting. A rule of thumb that's often used by data scientists is that there should be at least 5 records for every dimension. We've already seen that even for small text datasets, TF-IDF matrices can quickly push into 10 or 20 thousand dimensions. And that's true for many other types of data, too.

From the "sofa shadow" example, you can see that we can build infinitely many lower-dimensional representations of the same "original" dataset. But some representations are better than others. What does "better" mean in this case? When talking about visual data, you can intuitively understand that a representation that allows us to recognize the object is better than the one that doesn't. For example, let's take a point cloud that was taken from a 3D scan of a real object, and project it onto a two dimensional plane.

You can see the result in Figure 4.3. Can you guess what the 3D object was from that representation?

Figure 4.3. Looking up from below the "belly" at the point cloud for a real object



To continue our "shadows" analogy, think about the midday sun shining above the heads of a group of people. Every person's shadow would be a round patch. Would we be able to use those patches to tell who is tall and who is short, or which people have long hair? Probably not.

Now you understand that good dimensionality reduction has to do with being able to *distinguish* between different objects and data points in the new representation. And that not all features, or dimensions, of your data are equally important for that process of distinguishing. So there will be features which you can easily discard without losing much information. But for some features, losing them will significantly hurt your ability to understand your data. And because you are dealing with linear algebra here, you don't only have the option of leaving out or including a dimension - you can also combine several dimensions into a smaller dimension set that will represent our data in a more concise way. Let's see how we do that.

4.3.1 Enter Principal Component Analysis

You now know that to find your data's representation in fewer dimensions, you need to find a combination of dimensions that will preserve your ability to distinguish between data points. This will let you, for example, to separate them into meaningful clusters. To continue the shadow example, a good "shadow representation" allows you to see where is the head and where are the legs of your shadow. It does it by preserving the difference in height between these objects, rather than "squishing them" into one spot like the "midday sun representation" does. On the other hand, our body's "thickness" is roughly uniform from top to bottom - so when you see our "flat" shadow representation, that discards that dimension, you don't lose as much information as in the case of discarding our height.

In mathematics, this difference is represented by *variance*. And when you think about it makes sense that features with *more* variance - wider and more frequent deviation from the mean - are more helpful for you to tell the difference between data points.

But you can go beyond looking at each feature by itself. What matters also is

how the features relate between each other. Here, the visual analogies may start to fail you, because the three dimensions we operate in are orthogonal to each other, and thus completely unrelated. But let's think back about our topic vectors you saw in the previous part: "animalness", "petness", "cityness". If you examine every two features among this triade, it becomes obvious that some features are more strongly connected than others. Most words that have a "petness" quality to them, also have some "animalness" one. This property of a pair of features, or dimensions, is called *covariance*. It is strongly connected to *correlation*, which is just covariance normalized by the variance of each feature in the tandem. The higher the covariance between features, the more connected they are - and therefore, there is more redundancy between the two of them, as you can deduce one from the other. It also means that you can find a single dimension that preserves most of the variance contained in these two dimensions.

To summarize, to reduce the number of dimensions describing our data without losing information, you need to find a representation which *maximizes* the variance along each of its new axes, while reducing the dependence between the dimensions and getting rid of those who have high covariance. This is exactly what *Principal Component Analysis*, or PCA, does. It finds a set of dimensions along which the variance is maximized. These dimensions are *orthonormal* (like x,y and z axes in the physical world) and are called *principal components* - hence the name of the method. PCA also allows you to see how much variance each dimension "is responsible for", so that you can choose the optimal number of principal components that preserve the "essence" of your data set. PCA then takes your data and projects it into a new set of coordinates.

Before we dive into how PCA does that, let's see the magic in action. In the following listing, you will use the PCA method of scikit-learn to take the same 3D point cloud you've seen on the last page, and find a set of two dimensions that will maximize the variance of this point cloud.

Listing 4.5. PCA Magic

```
>>> import pandas as pd  
>>> pd.set_option('display.max_columns', 6)  
>>> from sklearn.decomposition import PCA
```

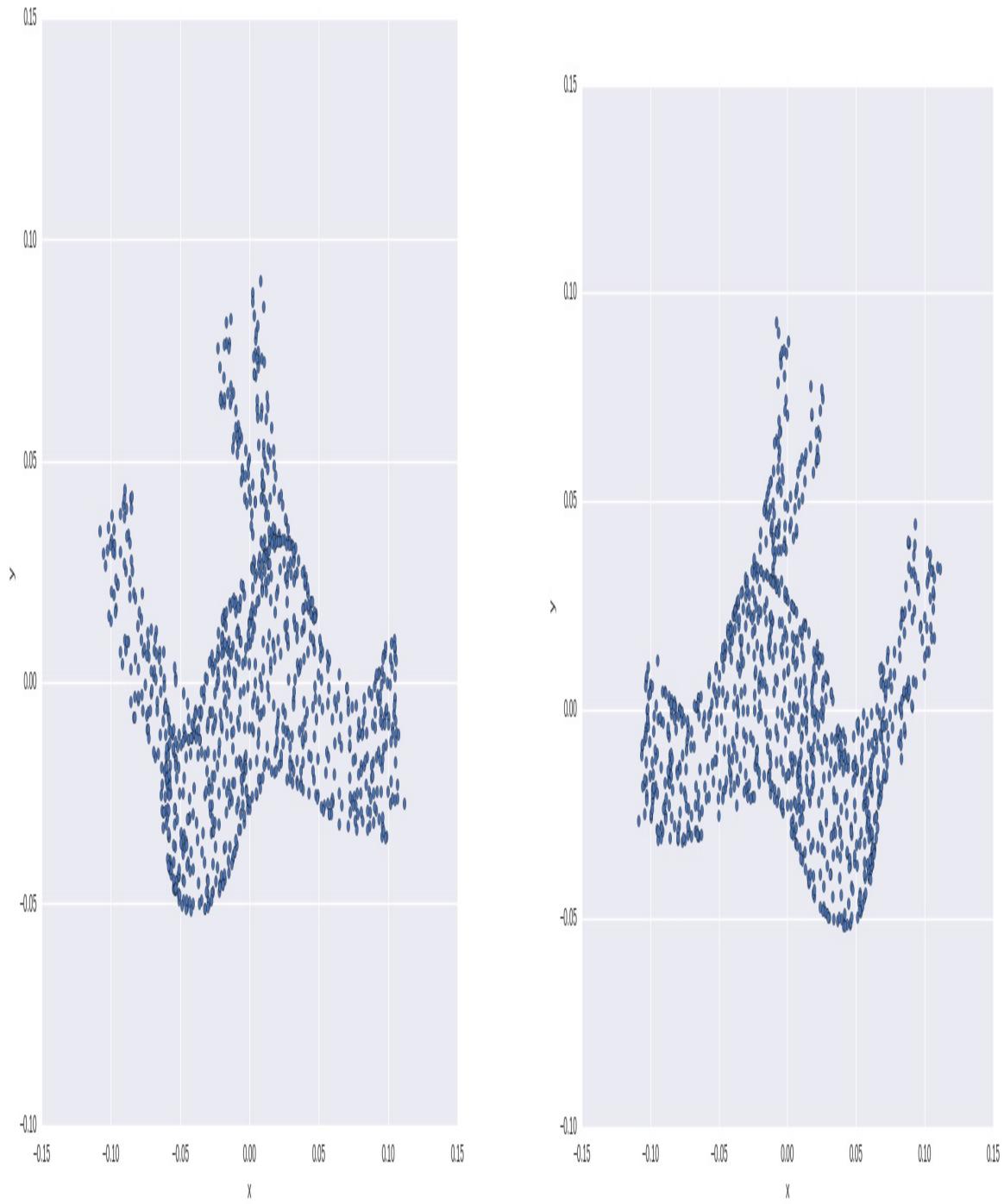
```
>>> import seaborn
>>> from matplotlib import pyplot as plt

>>> DATA_DIR = ('https://gitlab.com/tangibleai/nlpia/'
...             '-/raw/master/src/nlpia/data')

>>> df = pd.read_csv(DATA_DIR + '/pointcloud.csv.gz', index_col=0
>>> pca = PCA(n_components=2)
>>> df2d = pd.DataFrame(pca.fit_transform(df), columns=list('xy'))
>>> df2d.plot(kind='scatter', x='x', y='y')
>>> plt.show()
```

The result of running this code may look like a picture on the right or the left of figure 4.4, but it will never tip or twist to a new angle. That's because PCA always finds the two dimensions that will maximize the variance, and in the code we align these dimensions with x and y axes. However 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.

Figure 4.4. Head-to-head horse point clouds upside down



Now that we've seen PCA in the works,[\[143\]](#) let's take a look at how it finds those principal components that allow us to work with our data in fewer dimensions without losing much information.

4.3.2 Singular Value Decomposition

At the heart of PCA is a mathematical procedure called Singular Value Decomposition, or SVD.^[144] SVD is an algorithm for decomposing any matrix into three "factors", three matrices that can be multiplied together to recreate the original matrix. This is analogous to finding exactly three integer factors for a large integer. But your factors aren't scalar integers, they are 2D real matrices with special properties.

Let's say we have our dataset, consisting of m n-dimensional points, represented by a matrix W . In its full version, this is what SVD of W would look like in math notation (assuming $m > n$):

$$W_{m \times n} = U_{m \times m} S_{m \times n} V_{n \times n}^T$$

The matrices U , S and V have special properties. U and V matrices are *orthogonal*, meaning that if you multiply them by their transposed versions, you'll get a unit matrix. And S is *diagonal*, meaning that it has non-zero values only on its diagonal.

Note the equality sign in this formula. It means that if you multiply U , S and V , you'll get *exactly* W , our original dataset. But you can see that the smallest dimension of our matrices is still n . Didn't we want to reduce the number of dimensions? That's why in this chapter, you'll be using the version of SVD called *reduced*, or *truncated* SVD.^[145] That means that you'll only looking for the top p dimensions that you're interested in.

At this point you could say "Wait, but couldn't we do the full SVD and just take the dimensions that preserve maximum variance?" And you'll be completely right, we could do it this way! However, there are other benefits to using truncated SVD. In particular, there are several algorithms that allow computing truncated SVD decomposition of the matrix pretty fast, especially when the matrice is sparse. *Sparse matrices* are matrices that have the same value (usually zero or NaN) in most of its cells. NLP bag-of-words and TF-IDF matrices are almost always sparse, because most documents don't contain many of the words in your vocabulary.

This is what truncated SVD looks like:

$W_{m \times n} \sim U_{m \times p} S_{p \times p} V_{p \times n}^T$

In this formula, m and n are the number of rows and columns in the original matrix, while p is the number of dimensions you want to keep. For example, in the horse example, p would be equal to two if we want to display the horse in a two-dimensional space. In the next chapter, when you'll use SVD for LSA, it will signify the number of topics you want to use while analyzing your documents. Of course, p needs to be lesser than both m and n .

Note the "approximately equal" sign in this case - because we're losing dimensions, we can't expect to get exactly the same matrix when we multiply our factors! There's always some loss of information. What we're gaining, though, is a new way to represent our data with fewer dimensions than the original representation. With our horse point cloud, we are now able to convey its "horsy" essence without needing to print voluminous 3-D plots. And when PCA is used in real life, it can simplify hundred- or thousand-dimensional data into short vectors that are easier to analyze, cluster and visualize.

So, what are the matrices U, S and V useful for? For now, we'll give you a simple intuition of their roles. In the next chapter, we'll dive deeper into these matrices' application when we talk about LSA.

Let's start with V^T - or rather, with its transposed version V . V matrix's columns are sometimes called *principal directions*, and sometimes *principal components*. As `scikit-learn` library, which you utilize in this chapter, uses the latter convention, we're going to stick to it as well.

You can think of V as a "transformer" tool, that is used to map your data from the "old" space (its representation in matrix W 's "world") to the new, lower dimensional one. Imagine our we added a few more points to our 3D horse point cloud and now want to understand where those new point would be in our 2D representation, without needing to recalculate the transformation for all the points. To map every new point q to its location on a 2D plot, all you need to do is to multiply it by V :

$$\hat{q} = q \cdot V$$

What is, then the meaning of ' $U \cdot S$ '? With some algebra wizardry, you can see that it is actually your data mapped into the new space! Basically, it your data points in new, lesser-dimensional representation.

[143] To understand dimensionality reduction more in depth, check out this great 4-part post series by Hussein Abdullatif: <http://mng.bz/RlRv>

[144] There are actually two main ways to perform PCA; you can dig into the Wikipedia article for PCA

(https://en.wikipedia.org/wiki/Principal_component_analysis#Singular_value) and see what the other method is and how the two basically yield an almost identical result.

[145] To learn more about *Full* SVD and its other applications, you can read the Wikipedia article:

https://en.wikipedia.org/wiki/Singular_value_decomposition

4.4 Latent Semantic Analysis

Finally, we can stop "horsing around" and get back to topic modeling! Let's see how everything you've learned about dimensionality reduction, PCA and SVD will start making sense when we talk about finding topics and concepts in our text data.

Let's start from the dataset itself. You'll use the same comment corpus you used for the LDA classifier in section 4.1, and transform it into a matrix using TF-IDF. You might remember that the result is called a term-document matrix. This name is useful, because it gives you an intuition on what the rows and the columns of the matrix contain: the rows would be terms, your vocabulary words; and the columns will be documents.

Let's re-run listings 4.1 and 4.2 to get to our TF-IDF matrix again. Before diving into LSA, we examined the matrix shape:

```
>>> tfidf_docs.shape  
(5000, 19169)
```

So what do you have here? A 19,169-dimensional dataset, whose "space" is

defined by the terms in the corpus vocabulary. It's quite a hassle to work with a single vector representation of comments in this space, because there are almost 20,000 numbers to work with in each vector - longer than the message itself! It's also hard to see if the messages, or sentences inside them, are similar conceptually - for example, expressions like "leave this page" and "go away" will have very low similarity score, despite their meanings being very close to each other. So it's much harder to cluster and classify documents in the way it's represented in TF-IDF matrix.

Also note that only 650 of your 5,000 messages (13%) are labeled as toxic. So you have an unbalanced training set with about 8:1 normal comments to toxic comments (personal attacks, obscenity, racial slurs, etc.). And you have a large vocabulary - the number of your vocabulary tokens (25172) is greater than the 4,837 messages (samples) you have to go on. So you have many more unique words in your vocabulary (or lexicon) than you have comments, and even more when you compare it to the number of toxic messages. That's a recipe for overfitting.^[146] Only a few unique words out of your large vocabulary will be labeled as "toxic" words in your dataset.

Overfitting means that you will "key" off of only few words in your vocabulary. So your toxicity filter will be dependent on those toxic words being somewhere in the toxic messages it filters out. Trolls could easily get around your filter if they just used synonyms for those toxic words. If your vocabulary doesn't include the new synonyms, then your filter will mis-classify those cleverly constructed comments as non-toxic.

And this overfitting problem is an inherent problem in NLP. It's hard to find a labeled natural language dataset that includes all the ways that people might say something that should be labeled that way. We couldn't find an "ideal" set of comments that included all the different ways people say toxic and nontoxic things. And only a few corporations have the resources to create such a dataset. So all the rest of us need to have "countermeasures" for overfitting. You have to use algorithms that "generalize" well on just a few examples.

The primary countermeasure to overfitting is to map this data into a new, lower-dimensional space. What will define this new space are weighted combinations of words, or *topics*, that your corpus talks about in a variety of

ways. Representing your messages using topics, rather than specific term frequency, will make your NLP pipeline more "general", and allow our spam filter to work on a wider range of messages. That's exactly what LSA does - it finds the new topic "dimensions", along which variance is maximized, using SVD method we discovered in the previous section.

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

But you don't have to give the topics a name to make use of them. Just as you didn't analyze all the 1000s of dimensions in your stemmed bag-of-words vectors or TF-IDF vectors from previous chapters, you don't have to know what all your topics "mean." You can still do vector math with these new topic vectors, just like you did with TF-IDF vectors. You can add and subtract them and estimate the similarity between documents based on their "topic representation", rather than "term frequency representation". And these similarity estimates will be more accurate, because your new representation actually takes into account the meaning of tokens and their co-occurrence with other tokens.

4.4.1 Diving into semantic analysis

But enough talking about LSA - let's do some coding! This time, we're going to use another `scikit-learn` tool named `TruncatedSVD` that performs - what a surprise - the truncated SVD method that we examined in the previous chapter. We could use the `PCA` model you saw in the previous section, but we'll go with this more direct approach - it will allow us to understand better what's happening "under the hood". In addition `TruncatedSVD` is meant to deal with sparse matrices, so it will perform better on most TF-IDF and BOW

matrices.

We will start with decreasing the number of dimensions from 9232 to 16 - we'll explain later how we chose that number.

Listing 4.6. LSA using TruncatedSVD

```
>>> from sklearn.decomposition import TruncatedSVD  
>>>  
>>> svd = TruncatedSVD(n_components=16, n_iter=100)      #1  
>>> columns = ['topic{}'.format(i) for i in range(svd.n_components)]  
>>> svd_topic_vectors = svd.fit_transform(tfidf_docs)       #2  
>>> 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  
comment0    0.121  -0.055   0.036   ...    -0.038    0.089   0.011  
comment1    0.215   0.141  -0.006   ...     0.079   -0.016  -0.070  
comment2    0.342  -0.200   0.044   ...    -0.138    0.023   0.069  
comment3    0.130  -0.074   0.034   ...    -0.060    0.014   0.073  
comment4!   0.166  -0.081   0.040   ...    -0.008    0.063  -0.020  
comment5    0.256  -0.122  -0.055   ...     0.093   -0.083  -0.074
```

What you have just produced using `fit-transform` method is your document vectors in the new representation. Instead of representing your comments with 19,169 frequency counts, you represented it with just 16. This matrix is also called *document-topic* matrix. By looking at the columns, you can see how much every topic is "expressed" in every comment.



Note

How do the methods we use relate to the matrix decomposition process we described? You might have realized that what the `fit_transform` method returns is exactly $U \cdot S$ - your tf-idf vectors projected into the new space. And your V matrix is saved inside the `TruncatedSVD` object in the `components_` variable.

If you want to explore your topics, you can find out how much of each word they "contain" by examining the weights of each word, or groups of words, across every topic.

First let's assign words to all the dimensions in your transformation. You need to get them in the right order because your `TfidfVectorizer` stores the vocabulary as a dictionary that maps each term to an index number (column number).

```
>>> list(tfidf_model.vocabulary_.items())[:5] #1
[('you', 18890),
 ('have', 8093),
 ('yet', 18868),
 ('to', 17083),
 ('identify', 8721)]
>>> column_nums, terms = zip(*sorted(zip(tfidf.vocabulary_.values
...     tfidf.vocabulary_.keys())))
#2
>>> terms[:5]
('\\n', '\\n ', '\\n \\n', '\\n \\n ', '\\n ')
```

Now you can create a nice Pandas DataFrame containing the weights, with labels for all the columns and rows in the right place. But it looks like our first few terms are just different combinations of newlines - that's not very useful!

Whoever gave you the dataset should have done a better job of cleaning them out. Let's look at a few random terms from your vocabulary using the helpful Pandas method `DataFrame.sample()`

```
>>> topic_term_matrix = pd.DataFrame(svd.components_, columns=ter
... index=['topic{}'.format(i) for i in range(16)])
>>> pd.options.display.max_columns = 8
>>> topic_term_matrix.sample(5, axis='columns', random_state=2718
    littered  unblock.(t•c  orchestra  flanking  civilised
topic0  0.000268      0.000143      0.000630      0.000061      0.000119
topic1  0.000297     -0.000211     -0.000830     -0.000088     -0.000168
topic2 -0.000367      0.000157     -0.001457     -0.000150     -0.000133
topic3  0.000147     -0.000458      0.000804      0.000127      0.000181
```

None of these words looks like "inherently toxic". Let's look at some words that we would intuitively expect to appear in "toxic" comments, and see how much weight those words have in different topics.

```
>>> pd.options.display.max_columns = 8
>>> toxic_terms = topic_term_matrix['pathetic crazy stupid idiot
>>> toxic_terms
    pathetic  crazy  stupid  idiot  lazy  hate  die  kill
```

```

topic0      0.3    0.1    0.7    0.6    0.1    0.4    0.2    0.2
topic1     -0.2    0.0   -0.1   -0.3   -0.1   -0.4   -0.1    0.1
topic2      0.7    0.1    1.1    1.7   -0.0    0.9    0.6    0.8
topic3     -0.3   -0.0   -0.0    0.0    0.1   -0.0    0.0    0.2
topic4      0.7    0.2    1.2    1.4    0.3    1.7    0.6    0.0
topic5     -0.4   -0.1   -0.3   -1.3   -0.1    0.5   -0.2   -0.2
topic6      0.0    0.1    0.8    1.7   -0.1    0.2    0.8   -0.1
...
>>> toxic_terms.T.sum()
topic0      2.4
topic1     -1.2
topic2      5.0
topic3     -0.2
topic4      5.9
topic5     -1.8
topic6      3.4
topic7     -0.7
topic8      1.0
topic9     -0.1
topic10     -6.6
...

```

Topics 2 and 4 appear to be more likely to contain toxic sentiment. And topic 10 seems to be an "anti-toxic" topic. So words associated with toxicity can have a positive impact on some topics and a negative impact on others. There's no single obvious toxic topic number.

And what `transform` method does is just multiply whatever you pass to it with V matrix, which is saved in `components_`. You can check out the code of `TruncatedSVD` to see it with your own eyes! [\[147\]](#) link at the top left of the screen.]

4.4.2 TruncatedSVD or PCA?

You might be asking yourself now - why did we use scikit-learn's PCA class in the horse example, but `TruncatedSVD` for topic analysis for our comment dataset? Didn't we say that PCA is based on the SVD algorithm?

And you will be right - if you look into the implementation of PCA and `TruncatedSVD` in `sklearn`, you'll see that most of the code is similar between the two. They both use the same algorithms for SVD decomposition of matrices. However, there are several differences that might make each model

preferable for some use cases or others.

The biggest difference is that TruncatedSVD does not center the matrix before the decomposition, while PCA does. What this means is that if you center your data before performing TruncatedSVD by subtracting columnwise mean from the matrix, like this:

```
>>> tfidf_docs = tfidf_docs - tfidf_docs.mean()
```

You'll get the same results for both methods. Try this yourself by comparing the results of TruncatedSVD on centered data and of PCA, and see what you get!

The fact that the data is being centered is important for some properties of Principal Component Analysis, [\[148\]](#) which, you might remember, has a lot of applications outside NLP. However, for TF-IDF matrices, that are mostly sparse, centering doesn't always make sense. In most cases, centering makes a sparse matrix dense, which causes the model run slower and take much more memory. PCA is often used to deal with dense matrices and can compute a precise, full-matrix SVD for small matrices. In contrast, TruncatedSVD already assumes that the input matrix is sparse, and uses the faster approximated, randomized methods. So it deals with your TF-IDF data much more efficiently than PCA.

4.4.3 How well LSA performs for toxicity detection?

You've spent enough time peering into the topics - let's see how our model performs with lower-dimensional representation of the comments! You'll use the same code we ran in listing 4.3, but will apply it on the new 16-dimensional vectors. This time, the classification will go much faster:

```
>>> X_train_16d, X_test_16d, y_train_16d, y_test_16d = train_tes
>>> lda_lsa = LinearDiscriminantAnalysis(n_components=1)
>>> lda_lsa = lda_lsa.fit(X_train_16d, y_train_16d)
>>> round(float(lda_lsa.score(X_train_16d, y_train_16d)), 3)
0.881
>>> round(float(lda_lsa.score(X_test_16d, y_test_16d)), 3)
0.88
```

Wow, what a difference! The classifier's accuracy on the training set dropped from 99.9% for TF-IDF vectors to 88.1% But the test set accuracy jumped by 33%! That's quite an improvement.

Let's check the F1 score:

```
>>> from sklearn.metrics import f1_score  
>>> f1_score(y_test_16d, lda_lsa.predict(X_test_16d)).round(3)  
0.342
```

We've almost doubled our F1 score, compared to TF-IDF vectors classification! Not bad.

Unless you have a perfect memory, by now you must be pretty annoyed by scrolling or paging back to the performance of the previous model. And when you'll be doing real-life natural language processing, you'll probably be trying much more models than in our toy example. That's why data scientists record their model parameters and performance in a *hyperparameter table*.

Let's make one of our own. First, recall the classification performance we got when we run an LDA classifier on TF-IDF vectors, and save it into our table.

```
>>> hparam_table = pd.DataFrame()  
>>> tfidf_performance = {'classifier': 'LDA',  
...                      'features': 'tf-idf (spacy tokenizer)',  
...                      'train_accuracy': 0.99 ,  
...                      'test_accuracy': 0.554,  
...                      'test_precision': 0.383 ,  
...                      'test_recall': 0.12,  
...                      'test_f1': 0.183}  
>>> hparam_table = hparam_table.append(  
...      tfidf_performance, ignore_index=True) #1
```

Actually, because you're going to extract these scores for a few models, it might make sense to create a function that does this:

Listing 4.7. A function that creates a record in hyperparameter table.

```
>>> def hparam_rec(model, X_train, y_train, X_test, y_test, model_name)  
...     return {'classifier': model_name,  
...             'features': features,  
...             'train_accuracy': float(model.score(X_train, y_train)),
```

```

...
    'test_accuracy': float(model.score(X_test, y_test)),
...
    'test_precision': precision_score(y_test, model.predict
...
    'test_recall': recall_score(y_test, model.predict(X_tes
...
    'test_f1': f1_score(y_test, model.predict(X_test)) }
>>> lsa_performance = hparam_rec(lda_lsa, X_train_16d, y_train_16
...
        X_test_16d,y_test_16d, 'LDA', 'LSA (16 components)'))
>>> hparam_table = hparam_table.append(lsa_performance)
>>> hparam_table.T #1
          0           1
classifier          LDA          LDA
features      tf-idf (spacy tokenizer)  LSA (16d)
train_accuracy          0.99       0.8808
test_accuracy           0.554       0.88
test_precision           0.383       0.6
test_recall              0.12      0.239264
test_f1                 0.183     0.342105

```

You can go even further and wrap most of your analysis in a nice function, so that you don't have to copy-paste again:

```

>>> def evaluate_model(X,y, classifier, classifier_name, features
...   X_train, X_test, y_train, y_test = train_test_split(X, y, te
...   classifier = classifier.fit(X_train, y_train)
...   return hparam_rec(classifier, X_train, y_train, X_test,y_t
...                     classifier_name, features)

```

4.4.4 Other ways to reduce dimensions

SVD is by far the most popular way to reduce dimensions of a dataset, making LSA your first choice when thinking about topic modeling. However, there are several other dimensionality reduction techniques you can also use to achieve the same goal. Not all of them are even used in NLP, but it's good to be aware of them. We'll mention two methods here - *random projection* and *non-negative matrix factorization* (NMF).

Random projection is a method to project a high-dimensional data on lower-dimensional space, so that the distances between data points are preserved. Its stochastic nature makes it easier to run it on parallel machines. It also allows the algorithm to use less memory as it doesn't need to hold all the the data in the memory at the same time the way PCA does. And because its computational complexity lower, random projections can be occasionally used on datasets with very high dimensions, when decomposition speed is an

important factor.

Similarly, NMF is another matrix factorization method that is similar to SVD, but assumes that the data points and the components are all non-negative. It's more commonly used in image processing and computer vision, but can occasionally come handy in NLP and topic modeling too.

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

[146] See the web page titled "Overfitting - Wikipedia" (<https://en.wikipedia.org/wiki/Overfitting>).

[147] You can access the code of any scikit-learn function by clicking the [source] link at the top left of the screen.

[148] You can dig into the maths of PCA here:
https://en.wikipedia.org/wiki/Principal_component_analysis

4.5 Latent Dirichlet allocation (LDiA)

You've spent most of this chapter talking about latent semantic analysis and various ways to accomplish it using scikit-learn. LSA should be your first choice for most topic modeling, semantic search, or content-based recommendation engines. [149] Its math is straightforward and efficient, and it produces a linear transformation that can be applied to new batches of natural language without training and with little loss in accuracy. But we'll show you another algorithm, *Latent Dirichlet Allocation* (or LDiA, to distinguish it from LDA you've met before), than can give you slightly better results in some situations.

LDiA does a lot of the things you did to create your topic models with LSA (and SVD under the hood), but unlike LSA, LDiA assumes a Dirichlet distribution of word frequencies. It's more precise about the statistics of allocating words to topics than the linear math of LSA.

LDiA creates a semantic vector space model (like your topic vectors) using

an approach similar to how your brain worked during the thought experiment earlier in the chapter. In your thought experiment, you manually allocated words to topics based on how often they occurred together in the same document. The topic mix for a document can then be determined by the word mixtures in each topic by which topic those words were assigned to. This makes an LDiA topic model much easier to understand, because the words assigned to topics and topics assigned to documents tend to make more sense than for LSA.

LDiA assumes that each document is a mixture (linear combination) of some arbitrary number of topics that you select when you begin training the LDiA model. LDiA also assumes that each topic can be represented by a distribution of words (term frequencies). The probability or weight for each of these topics within a document, as well as the probability of a word being assigned to a topic, is assumed to start with a Dirichlet probability distribution (the *prior* if you remember your statistics). This is where the algorithm gets its name.

4.5.1 The LDiA idea

The LDiA approach was developed in 2000 by geneticists in the UK to help them "infer population structure" from sequences of genes.^[150] Stanford Researchers (including Andrew Ng) popularized the approach for NLP in 2003.^[151] But don't be intimidated by the big names that came up with this approach. We explain the key points of it in a few lines of Python shortly. You only need to understand it enough to get a feel for what it's doing (an intuition), so you know what you can use it for in your pipeline.

Blei and Ng came up with the idea by flipping your thought experiment on its head. They imagined how a machine that could do nothing more than roll dice (generate random numbers) could write the documents in a corpus you want to analyze. And because you're only working with bags of words, they cut out the part about sequencing those words together to make sense, to write a real document. They just modeled the statistics for the mix of words that would become a part of a particular the BOW for each document.

They imagined a machine that only had two 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. Your document "character sheet" needs only two rolls of the dice. But the dice are large and there are several of them, with complicated rules about how they are combined to produce the desired probabilities for the different values you want. You want particular probability distributions for the number of words and number of topics so that it matches the distribution of these values in real documents analyzed by humans for their topics and words.

The two rolls of the dice represent:

1. Number of words to generate for the document (Poisson distribution)
2. Number of topics to mix together for the document (Dirichlet distribution)

After it has these two numbers, the hard part begins, choosing the words for a document. The imaginary BOW generating machine iterates over those topics and randomly chooses words appropriate to that topic until it hits the number of words that it had decided the document should contain in step 1. 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, your "bot" just looks up the probabilities for the words for each topic from a matrix of term-topic probabilities. If you don't remember what that matrix looks like, glance back at the simple example earlier in this chapter.

So all this machine needs is a single parameter for that Poisson distribution (in the dice roll from step 1) that tells it what the "average" document length should be, and a couple more parameters to define that Dirichlet distribution that sets up the number of topics. Then your document generation algorithm needs a term-topic matrix of all the words and topics it likes to use, its vocabulary. And it needs a mix of topics that it likes to "talk" about.

Let's flip the document generation (writing) problem back around to your original problem of estimating the topics and words from an existing document. You need to measure, or compute, those parameters about words

and topics for the first two steps. Then you 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 by analyzing the statistics of the documents in a corpus. For example, for step 1, they could calculate the mean number of words (or n -grams) in all the bags of words for the documents in their corpus, something like this:

```
>>> total_corpus_len = 0
>>> for document_text in comments.text:
...     total_corpus_len += len(spacy_tokenize(document_text))
>>> mean_document_len = total_corpus_len / len(sms)
>>> round(mean_document_len, 2)
21.35
```

Or, in a one-liner:

```
>>> sum([len(spacy_tokenize(t)) for t in comments.text]) * 1. / 1
21.35
```

Keep in mind, you should calculate this statistic directly from your BOWs. You need to make sure you're counting the tokenized and vectorized words in your documents. And make sure you've applied any stop word filtering, or other normalizations before you count up your unique terms. That way your count includes all the words in your BOW vector vocabulary (all the n -grams you're counting), but only those words that your BOWs use (not stop words, for example). This LDiA algorithm relies on a bag-of-words vector space model, unlike LSA that took TF-IDF matrix as input.

The second parameter you need to specify for an LDiA model, the number of topics, is a bit trickier. The number of topics in a particular set of documents can't be measured directly until after you've assigned words to those topics. Like k -means and KNN and other clustering algorithms, you must tell it the k ahead of time. You can guess the number of topics (analogous to the k in k-means, the number of "clusters") and then check to see if that works for your set of documents. Once you've told LDiA how many topics to look for, it will find the mix of words to put in each topic to optimize its objective function. [\[152\]](#)

You can optimize this "hyperparameter" (k , the number of topics) [\[153\]](#) by

adjusting it until it works for your application. You can automate this optimization if you can measure something about the quality of your LDiA language model for representing the meaning of your documents. One "cost function" you could use for this optimization is how well (or poorly) that LDiA model performs in some classification or regression problem, like sentiment analysis, document keyword tagging, or topic analysis. You just need some labeled documents to test your topic model or classifier on.

4.5.2 LDiA topic model for comments

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 tries to keep things spread apart that were spread apart to start with, LDiA tries to keep things close together that started out close together.

This may sound like it's the same thing, but it's not. The math optimizes for different things. Your optimizer has a different objective function so it will reach a different objective. To keep close high-dimensional vectors close together in the lower-dimensional space, LDiA has to twist and contort the space (and the vectors) in nonlinear ways. This is a hard thing to visualize until you do it on something 3D and take "projections" of the resultant vectors in 2D.

Let's see how that works for a dataset of a few thousand comments, labeled for spaminess. First compute the TF-IDF vectors and then some topics vectors for each SMS message (document). We assume the use of only 16 topics (components) to classify the spaminess of messages, as before. Keeping the number of topics (dimensions) low can help reduce overfitting.
[\[154\]](#)

LDiA works with raw BOW count vectors rather than normalized TF-IDF vectors. You've already done this process in chapter 3:

```
>>> from sklearn.feature_extraction.text import CountVectorizer  
>>>  
>>> counter = CountVectorizer(tokenizer=spacy_tokenize)  
>>> bow_docs = pd.DataFrame(counter.fit_transform(raw_documents=c  
...           .toarray(), index=index)
```

```
>>> column_nums, terms = zip(*sorted(zip(counter.vocabulary_.valu
...     counter.vocabulary_.keys())))
>>> bow_docs.columns = terms
```

Let's double-check that your counts make sense for that first comment labeled "comment0":

```
>>> comments.loc['comment0'].text
'you have yet to identify where my edits violated policy.
 4 july 2005 02:58 (utc)'
>>> bow_docs.loc['comment0'][bow_docs.loc['comment0'] > 0].head()
   1
(  1
)  1
.  1
02:58  1
Name: comment0, dtype: int64
```

We'll apply Latent Dirichlet Allocation to the count vector matrix in the same way we applied LSA to TF-IDF matrix:

```
>>> from sklearn.decomposition import LatentDirichletAllocation as
>>> ldia = LDiA(n_components=16, learning_method='batch')
>>> ldia = ldia.fit(bow_docs)      #1
>>> ldia.components_.shape
(16, 19169)
```

So your model has allocated your 19,169 words (terms) to 16 topics (components). Let's take a look at the first few words and how they're allocated. Keep in mind that your counts and topics will be different from ours. LDiA is a stochastic algorithm that relies on the random number generator to make some of the statistical decisions it has to make about allocating words to topics. So each time you run `sklearn.LatentDirichletAllocation` (or any LDiA algorithm), you will get different results unless you set the random seed to a fixed value.

```
>>> pd.set_option('display.width', 75)
>>> term_topic_matrix = pd.DataFrame(ldia.components_, index=term
...     columns=columns)
>>> term_topic_matrix.round(2).head(3)
                topic0  topic1  ...  topic14  topic15
a            21.853  0.063  ...    0.063  922.515
aaaaaaaaaaaaahhhhhhhhhhhh  0.063  0.063  ...    0.063  0.063
```

| | | | | | |
|-------|-------|-------|-----|-------|-------|
| aalst | 0.063 | 0.063 | ... | 0.063 | 0.063 |
| aap | 0.063 | 0.063 | ... | 2.062 | 0.062 |

It looks like the values in LDiA topic vectors have much higher spread than LSA topic vectors - there are a lot of near-zero values, but also some really big ones. Let's do the same trick you did when performing topic modeling with LSA. We can look at typical "toxic" words and see how pronounced they are in every topic.

```
>>> toxic_terms= components.loc['pathetic crazy stupid lazy idiot'
>>> toxic_terms
      topic0  topic1  topic2  ...  topic13  topic14  topic15
pathetic    1.06    0.06   32.35  ...     0.06    0.06    9.47
crazy       0.06    0.06    3.82  ...     1.17    0.06    0.06
stupid      0.98    0.06    4.58  ...     8.29    0.06   35.80
lazy        0.06    0.06    1.34  ...     0.06    0.06   3.97
idiot       0.06    0.06    6.31  ...     0.06    1.11   9.91
hate        0.06    0.06    0.06  ...     0.06   480.06    0.06
die         0.06    0.06   26.17  ...     0.06    0.06    0.06
kill        0.06    4.06    0.06  ...     0.06    0.06    0.06
```

That looks very different from the LSA representation of our toxic terms! Looks like some terms can have high topic-term weights in some topics, but not others. `topic0` and `topic1` seem pretty "indifferent" to toxic terms, while `topic 2` and `topic 15` have quite large topic-terms weight for at least 4 or 5 of the toxic terms. And `topic14` has a very high weight for the term `hate`!

Let's see what other terms scored high in this topic. As you saw earlier, because we didn't do any preprocessing to our dataset, a lot of terms are not very interesting. Let's focus on terms that are words, and are longer than 3 letters - that would eliminate a lot of the stop words.

```
>>> non_trivial_terms = [term for term in components.index
                         if term.isalpha() and len(term)>3]
components.topic14.loc[non_trivial_terms].sort_values(ascending=F
hate          480.062500
killed        14.032799
explosion     7.062500
witch         7.033359
june          6.676174
wicked        5.062500
dead           3.920518
years          3.596520
```

| | |
|---------|----------|
| wake | 3.062500 |
| arrived | 3.062500 |

It looks like a lot of the words in the topic have semantic relationship between them. Words like "killed" and "hate", or "wicked" and "witch", seem to belong in the "toxic" domain. You can see that the allocation of words to topics can be rationalized or reasoned about, even with this quick look.

Before you fit your classifier, you need to compute these LDiA topic vectors for all your documents (comments). And let's see how they are different from the topic vectors produced by LSA 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
comment0    0.0    0.0    0.00  ...    0.00    0.0    0.0
comment1    0.0    0.0    0.28  ...    0.00    0.0    0.0
comment2    0.0    0.0    0.00  ...    0.00    0.0    0.0
comment3    0.0    0.0    0.00  ...    0.95    0.0    0.0
comment4!   0.0    0.0    0.07  ...    0.00    0.0    0.0
```

You can see that these topics are more cleanly separated. There are a lot of zeros in your allocation of topics to messages. This is one of the things that makes LDiA topics easier to explain to coworkers when making business decisions based on your NLP pipeline results.

So LDiA topics work well for humans, but what about machines? How will your LDA classifier fare with these topics?

4.5.3 Detecting toxicity with LDiA

Let's see how good these LDiA topics are at predicting something useful, such as comment toxicity. You'll use your LDiA topic vectors to train an LDA model again (like you did twice - with your TF-IDF vectors and LSA topic vectors). And because of the handy function you defined in listing 4.5, you only need a couple of lines of code to evaluate your model:

```
>>> model_ldia16 = LinearDiscriminantAnalysis()
>>> ldia16_performance=evaluate_model(ldia16_topic_vectors,
```

```

    comments.toxic, model_ldia16, 'LDA', 'LDIA (16 components)'
>>> hparam_table = hparam_table.append(ldia16_performance, ignore_index=True)
>>> hparam_table.T

```

| | 0 | 1 | 2 |
|----------------|--------------------------|-----------|------------|
| classifier | LDA | LDA | LDA |
| features | tf-idf (spacy tokenizer) | LSA (16d) | LDIA (16d) |
| train_accuracy | 0.99 | 0.8808 | 0.8688 |
| test_accuracy | 0.554 | 0.88 | 0.8616 |
| test_precision | 0.383 | 0.6 | 0.388889 |
| test_recall | 0.12 | 0.239264 | 0.107362 |
| test_f1 | 0.183 | 0.342105 | 0.168269 |

It looks that the classification performance on 16-topic LDIA vectors is worse than on the raw TF-IDF vectors, without topic modeling. Does it mean the LDIA is useless in this case? Let's not give up on it too soon and try to increase the number of topics.

4.5.4 A fairer comparison: 32 LDIA topics

Let's try one more time with more dimensions, more topics. Perhaps LDIA isn't as efficient as LSA so it needs more topics to allocate words to. Let's try 32 topics (components).

```

>>> ldia32 = LDIA(n_components=32, learning_method='batch')
>>> ldia32 = ldia32.fit(bow_docs)
>>> model_ldia32 = LinearDiscriminantAnalysis()
>>> ldia32_performance = evaluate_model(ldia32_topic_vectors,
...                                         comments.toxic, model_ldia32, 'LDA', 'LDIA (32d)')
>>> hparam_table = hparam_table.append(ldia32_performance,
...                                         ignore_index = True)
>>> hparam_table.T

```

| | 0 | 1 | 2 |
|----------------|--------------------------|-----------|------------|
| classifier | LDA | LDA | LDA |
| features | tf-idf (spacy tokenizer) | LSA (16d) | LDIA (16d) |
| train_accuracy | 0.99 | 0.8808 | 0.8688 |
| test_accuracy | 0.554 | 0.88 | 0.8616 |
| test_precision | 0.383 | 0.6 | 0.388889 |
| test_recall | 0.12 | 0.239264 | 0.107362 |
| test_f1 | 0.183 | 0.342105 | 0.168269 |

That's nice! Increasing the dimensions for LDIA almost doubled both the precision and the recall of the models, and our F1 score looks much better. The larger number of topics allows LDIA to be more precise about topics,

and, at least for this dataset, produce topics that linearly separate better. But the performance of this vector representations still is not quite as good as that of LSA. So LSA is keeping your comment topic vectors spread out more efficiently, allowing for a wider gap between comments 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 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

You saw earlier how you can browse the source code of all '`sklearn`' from the documentation pages. But there is even a more straightforward method to do it from your Python console. You can find the source code path in the `__file__` attribute on any Python module, such as `sklearn.__file__`. And in `ipython` (`jupyter` console), you can view the source code for any function, class, or object with `??`, like `LDA??`:

```
>>> import sklearn
>>> sklearn.__file__
'/Users/hobs/anaconda3/envs/conda_env_nlpia/lib/python3.6/site-pa
earn/__init__.py'
>>> from sklearn.discriminant_analysis\
...     import LinearDiscriminantAnalysis as LDA
>>> LDA??
Init signature: LDA(solver='svd', shrinkage=None, priors=None, n_
=None, store_covariance=False, tol=0.0001)
Source:
class LinearDiscriminantAnalysis(BaseEstimator, LinearClassifierM
                                TransformerMixin):
    """Linear Discriminant Analysis

    A classifier with a linear decision boundary, generated by fi
    class conditional densities to the data and using Bayes' rule

    The model fits a Gaussian density to each class, assuming tha
```

classes share the same covariance matrix."""

...

This won't work on functions and classes that are extensions, whose source code is hidden within a compiled C++ module.

[149] A 2015 comparison of content-based movie recommendation algorithms by Sonia Bergamaschi and Laura Po found LSA to be approximately twice as accurate as LDIA. See "Comparing LDA and LSA Topic Models for Content-Based Movie Recommendation Systems" by Sonia Bergamaschi and Laura Po (https://www.dbgroup.unimo.it/~po/pubs/LNBI_2015.pdf).

[150] "Jonathan K. Pritchard, Matthew Stephens, Peter Donnelly, Inference of Population Structure Using Multilocus Genotype Data"
<http://www.genetics.org/content/155/2/945>

[151] See the PDF titled "Latent Dirichlet Allocation" by David M. Blei, Andrew Y. Ng, and Michael I. Jordan
(<http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf>).

[152] You can learn more about the particulars of the LDIA objective function here in the original paper "Online Learning for Latent Dirichlet Allocation" by Matthew D. Hoffman, David M. Blei, and Francis Bach
(<https://www.di.ens.fr/%7Efba/mdlhnips2010.pdf>).

[153] The symbol used by Blei and Ng for this parameter was *theta* rather than *k*

[154] See Appendix D if you want to learn more about why overfitting is a bad thing and how *generalization* can help.

4.6 Distance and similarity

We need to revisit those similarity scores we talked about in chapters 2 and 3 to make sure your new topic vector space works with them. Remember that you can use similarity scores (and distances) to tell how similar or far apart two documents are based on the similarity (or distance) of the vectors you

used to represent them.

You can use similarity scores (and distances) to see how well your LSA topic model agrees with the higher-dimensional TF-IDF model of chapter 3. You'll see how good your model is at retaining those distances after having eliminated a lot of the information contained in the much higher-dimensional bags of words. You can check how far away from each other the topic vectors are and whether that's a good representation of the distance between the documents' subject matter. You want to check that documents that mean similar things are close to each other in your new topic vector space.

LSA preserves large distances, but it does not always preserve close distances (the fine "structure" of the relationships between your documents). The underlying SVD algorithm is focused on maximizing the variance between all your documents in the new topic vector space.

Distances between feature vectors (word vectors, topic vectors, document context vectors, and so on) drive the performance of an NLP pipeline, or any machine learning pipeline. So what are your options for measuring distance in high-dimensional space? And which ones should you chose for a particular NLP problem? Some of these commonly used examples may be familiar from geometry class or linear algebra, but many others are probably new to you:

- Euclidean or Cartesian distance, or root mean square error (RMSE): 2-norm or L_2
- Squared Euclidean distance, sum of squares distance (SSD): L_2^2
- Cosine or angular or projected distance: normalized dot product
- Minkowski distance: p-norm or L_p
- Fractional distance, fractional norm: p-norm or L_p for $0 < p < 1$
- City block, Manhattan, or taxicab distance, sum of absolute distance (SAD): 1-norm or L_1
- Jaccard distance, inverse set similarity
- Mahalanobis distance
- Levenshtein or edit distance

The variety of ways to calculate distance is a testament to how important it is.

In addition to the pairwise distance implementations in Scikit-learn, many others are used in mathematics specialties such as topology, statistics, and engineering.^[155] For reference, here are all the distances you can find in the `sklearn.metrics.pairwise` module:^[156]

Listing 4.8. Pairwise distances available in sklearn

```
'cityblock', 'cosine', 'euclidean', 'l1', 'l2', 'manhattan', 'bra  
'canberra', 'chebyshev', 'correlation', 'dice', 'hamming', 'jacca  
'kulinskis', 'mahalanobis', 'matching', 'minkowski', 'rogerstanim  
'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'squeu  
'yule'
```

Distance measures are often computed from similarity measures (scores) and vice versa such that distances are inversely proportional to similarity scores. Similarity scores are designed to range between 0 and 1. Typical conversion formulas look like this:

```
>>> similarity = 1. / (1. + distance)  
>>> distance = (1. / similarity) - 1.
```

But for distances and similarity scores that range between 0 and 1, like probabilities, it's more common to use a formula like this:

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

And cosine distances have their own convention for the range of values they use. The angular distance between two vectors is often computed as a fraction of the maximum possible angular separation between two vectors, which is 180 degrees or pi radians.^[157] As a result cosine similarity and distance are the reciprocal of each other:

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

Why do we spend so much time talking about distances? In the last section of this book, we'll be talking about semantic search. The idea behind semantic search is to find documents that have the highest *semantic similarity* with

your search query - or the lowest *semantic distance*. In our semantic search application, we'll be using cosine similarity - but as you can see in the last two pages, there are multiple ways to measure how similar documents are.

[155] See Math.NET Numerics for more distance metrics (<https://numerics.mathdotnet.com/Distance.html>).

[156] See the documentation for sklearn.metrics.pairwise (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distance.html).

[157] See the web page titled "Cosine similarity - Wikipedia" (https://en.wikipedia.org/wiki/Cosine_similarity).

4.7 Steering with feedback

All the previous approaches to semantic analysis failed to take into account information about the similarity between documents. We created topics that were optimal for a generic set of rules. Our unsupervised learning of these feature (topic) extraction models didn't have any data about how "close" the topic vectors should be to each other. We didn't allow any "feedback" about where the topic vectors ended up, or how they were related to each other.

Steering or "learned distance metrics"^[158] are the latest advancement in dimension reduction and feature extraction. By adjusting the distance scores reported to clustering and embedding algorithms, you can "steer" 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're interested in.

In the previous sections about LSA, you ignored all the meta information about your documents. For example, with the comments you ignored the sender of the message. This is a good indication of topic similarity and could be used to inform your topic vector transformation (LSA).

At Talentpair we experimented with matching resumes to job descriptions using the cosine distance between topic vectors for each document. This worked OK. But we learned pretty quickly that we got much better results

when we started "steering" our topic vectors based on feedback from candidates and account managers responsible for helping them find a job. Vectors for "good pairings" were steered closer together than all the other pairings.

One way to do this is to calculate the mean difference between your two centroids (like you did for LDA) and add some portion of this "bias" to all the resume or job description vectors. Doing so should take out the average topic vector difference between resumes and job descriptions. Topics such as beer on tap at lunch might appear in a job description but never in a resume. Similarly bizarre hobbies, such as underwater sculpture, might appear in some resumes but never a job description. Steering your topic vectors can help you focus them on the topics you're interested in modeling.

[158] See the web page titled "eccv spgraph"
(<http://users.cecs.anu.edu.au/~sgould/papers/eccv14-spgraph.pdf>).

4.8 Topic vector power

With topic vectors, you can do things like compare the meaning of words, documents, statements, and corpora. You can find "clusters" of similar documents and statements. You're no longer comparing the distance between documents based merely on their word usage. You're no longer limited to keyword search and relevance ranking based entirely on word choice or vocabulary. You can now find documents that are relevant to your query, not just a good match for the word statistics themselves.

This is called "semantic search", not to be confused with the "semantic web."
[159] Semantic search is what strong search engines do when they give you documents that don't contain many of the words in your query, but are exactly what you were looking for. These advanced search engines use LSA topic vectors to tell the difference between a Python package in "The Cheese Shop" and a python in a Florida pet shop aquarium, while still recognizing its similarity to a "Ruby gem."^[160]

Semantic search gives you a tool for finding and generating meaningful text. But our brains are not good at dealing with high-dimensional objects, vectors,

hyperplanes, hyperspheres, and hypercubes. Our intuitions as developers and machine learning engineers breaks down above three dimensions.

For example, to do a query on a 2D vector, like your lat/lon location on Google Maps, you can quickly find all the coffee shops nearby without much searching. You can just scan (with your eyes or with code) near your location and spiral outward with your search. Alternatively, you can create bigger and bigger bounding boxes with your code, checking for longitudes and latitudes within some range on each, that's just for comparison operations and that should find you everything nearby.

However, dividing up a high dimensional vector space (hyperspace) with hyperplanes and hypercubes as the boundaries for your search is impractical, and in many cases, impossible.

As Geoffry Hinton says, "To deal with hyperplanes in a 14-dimensional space, visualize a 3D space and say 14 to yourself 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 your 3D world into hyperspace, enough to catch a glimpse of that 3D world from the outside. Like in *Flatland*, you used a lot of 2D visualizations in this chapter to help you explore the shadows that words in hyperspace leave in your 3D world. If you're anxious to check them out, skip ahead to the section showing "scatter matrices" of word vectors. You might also want to glance back at the 3D bag-of-words vector in the previous chapter and try to imagine what those points would look like if you added just one more word to your vocabulary to create a 4-D world of language meaning.

If you're taking a moment to think deeply about four dimensions, keep in mind that the explosion in complexity you're trying to wrap your head around is even greater than the complexity growth from 2D to 3D and exponentially greater than the growth in complexity from a 1D world of numbers to a 2D world of triangles, squares, and circles.

4.8.1 Semantic search

When you search for a document based on a word or partial word it contains, that's called *full text search*. This is what search engines do. They break a document into chunks (usually words) that can be indexed with an *inverted index* like you'd find at the back of a textbook. It takes a lot of bookkeeping and guesswork to deal with spelling errors and typos, but it works pretty well. [\[161\]](#)

Semantic search is full text search that takes into account the meaning of the words in your query and the documents you're searching. In this chapter, you've learned two ways—LSA and LDiA—to compute topic vectors that capture the semantics (meaning) of words and documents in a vector. One of the reasons that latent semantic analysis was first called latent semantic *indexing* was because it promised to power semantic search with an index of numerical values, like BOW and TF-IDF tables. Semantic search was the next big thing in information retrieval.

But unlike BOW and TF-IDF tables, tables of semantic vectors can't be easily discretized and indexed using traditional inverted index techniques. Traditional indexing approaches work with binary word occurrence vectors, discrete vectors (BOW vectors), sparse continuous vectors (TF-IDF vectors), and low-dimensional continuous vectors (3D GIS data). But high-dimensional continuous vectors, such as topic vectors from LSA or LDiA, are a challenge. [\[162\]](#) Inverted indexes work for discrete vectors or binary vectors, like tables of binary or integer word-document vectors, because the index only needs to maintain an entry for each nonzero discrete dimension. Either that value of that dimension is present or not present in the referenced vector or document. Because TF-IDF vectors are sparse, mostly zero, you don't need an entry in your index for most dimensions for most documents. [\[163\]](#)

LSA (and LDiA) produce topic vectors that are high-dimensional, continuous, and dense (zeros are rare). And the semantic analysis algorithm does not produce an efficient index for scalable search. In fact, the curse of dimensionality that you talked about in the previous section makes an exact index impossible. The "indexing" part of latent semantic indexing was a hope, not a reality, so the LSI term is a misnomer. Perhaps that is why LSA has become the more popular way to describe semantic analysis algorithms that produce topic vectors.

One solution to the challenge of high-dimensional vectors is to index them with a *locality sensitive hash* (LSH). A locality sensitive hash is like a zip code (postal code) that designates a region of hyperspace so that it can easily be found again later. And like a regular hash, it is discrete and depends only on the values in the vector. But even this doesn't work perfectly once you exceed about 12 dimensions. In figure 4.6, each row represents a topic vector size (dimensionality), starting with 2 dimensions and working up to 16 dimensions, like the vectors you used earlier for the SMS spam problem.

Figure 4.5. Semantic search accuracy deteriorates at around 12-D

| Dimensions | 100th Cosine Distance | Top 1 Correct | Top 2 Correct | Top 10 Correct | Top 100 Correct |
|------------|-----------------------------|------------------|------------------|-------------------|--------------------|
| 2 | .00 | TRUE | TRUE | TRUE | TRUE |
| 3 | .00 | TRUE | TRUE | TRUE | TRUE |
| 4 | .00 | TRUE | TRUE | TRUE | TRUE |
| 5 | .01 | TRUE | TRUE | TRUE | TRUE |
| 6 | .02 | TRUE | TRUE | TRUE | TRUE |
| 7 | .02 | TRUE | TRUE | TRUE | FALSE |
| 8 | .03 | TRUE | TRUE | TRUE | FALSE |
| 9 | .04 | TRUE | TRUE | TRUE | FALSE |
| 10 | .05 | TRUE | TRUE | FALSE | FALSE |
| 11 | .07 | TRUE | TRUE | TRUE | FALSE |
| 12 | .06 | TRUE | TRUE | FALSE | FALSE |
| 13 | .09 | TRUE | TRUE | FALSE | FALSE |
| 14 | .14 | TRUE | FALSE | FALSE | FALSE |
| 15 | .14 | TRUE | TRUE | FALSE | FALSE |
| 16 | .09 | TRUE | TRUE | FALSE | FALSE |

The table shows how good your search results would be if you used locality sensitive hashing to index a large number of semantic vectors. Once your vector had more than 16 dimensions, you'd have a hard time returning 2 search results that were any good.

So how can you do semantic search on 100-D vectors without an index? You now know how to convert the query string into a topic vector using LSA. And you know how to compare two vectors for similarity using the cosine similarity score (the scalar product, inner product, or dot product) to find the closest match. To find precise semantic matches, you need to find all the closest document topic vectors to a particular query (search) topic vector. But if you have n documents, you have to do n comparisons with your query topic vector. That's a lot of dot products.

You can vectorize the operation in numpy using matrix multiplication, but that doesn't reduce the number of operations, it only makes them 100 times faster. [\[164\]](#) Fundamentally, exact semantic search still requires $O(N)$ multiplications and additions for each query. So it scales only linearly with the size of your corpus. That wouldn't work for a large corpus, such as Google Search or even Wikipedia semantic search.

The key is to settle for "good enough" rather than striving for a perfect index or LSH algorithm for our high-dimensional vectors. There are now several open source implementations of some efficient and accurate *approximate nearest neighbors* algorithms that use LSH to efficiently implement semantic search. A couple of the easiest to use and install are

- Spotify's Annoy package [\[165\]](#)
- Gensim's `gensim.models.KeyedVector` class. [\[166\]](#)

Technically these indexing or hashing solutions cannot guarantee that you will find all the best matches for your semantic search query. But they can get you a good list of close matches almost as fast as with a conventional reverse index on a TF-IDF vector or bag-of-words vector, if you're willing to give up a little precision. [\[167\]](#)

[\[159\]](#) 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.

[\[160\]](#) Ruby is a programming language whose packages are called gems.

[161] A full text index in a database like PostgreSQL is usually based on trigrams of characters, to deal with spelling errors and text that doesn't parse into words.

[162] Clustering high-dimensional data is equivalent to discretizing or indexing high-dimensional data with bounding boxes and is described in the Wikipedia article "Clustering high dimensional data" (https://en.wikipedia.org/wiki/Clustering_high-dimensional_data).

[163] See the web page titled "Inverted index - Wikipedia" (https://en.wikipedia.org/wiki/Inverted_index).

[164] Vectorizing your Python code, especially doubly-nested `for` loops for pairwise distance calculations can speed your code by almost 100-fold. See Hacker Noon article "Vectorizing the Loops with Numpy" (<https://hackernoon.com speeding-up-your-code-2-vectorizing-the-loops-with-numpy-e380e939bed3>).

[165] Spotify's researchers compared their annoy performance to that of several alternative algorithms and implementations on their github repo (<https://github.com/spotify/annoy>).

[166] The approach used in `gensim` for hundreds of dimensions in word vectors will work fine for any semantic or topic vector. See `gensim`'s "KeyedVectors" documentation (<https://radimrehurek.com/gensim/models/keyedvectors.html>).

[167] If you want to learn about faster ways to find a high-dimensional vector's nearest neighbors, check out appendix F, or just use the Spotify annoy package to index your topic vectors.

4.9 Equipping your bot with semantic search

Let's use your newly-acquired knowledge in topic modeling to improve the bot you started to build in the previous chapter. We'll focus on the same task - question answering.

Our code is actually going to be pretty similar to your code in chapter 3. We will still use vector representations to find the most similar question in our dataset. But this time, our representations are going to be closer to representing the meaning of those questions.

First, let's load the question and answer data just like we did in the last chapter

```
>>> REPO_URL = 'https://gitlab.com/tangibleai/qary/-/raw/master'  
>>> FAQ_DIR = 'src/qary/data/faq'  
>>> FAQ_FILENAME = 'short-faqs.csv'  
>>> DS_FAQ_URL = '/'.join([REPO_URL, FAQ_DIR, FAQ_FILENAME])  
  
>>> df = pd.read_csv(DS_FAQ_URL)
```

The next step is to represent both the questions and our query as vectors. This is where we need to add just a few lines to make our representations semantic. Because our question dataset is small, we won't need to apply LSH or any other indexing algorithm.

```
>>> vectorizer = TfidfVectorizer()  
>>> vectorizer.fit(df['question'])  
>>> tfidf_vectors = vectorizer.transform(df['question'])  
>>> svd = TruncatedSVD(n_components=16, n_iterations=100)  
>>> tfidf_vectors_16d = svd.fit_transform(tfidf_vectors)  
>>>  
>>> def bot_reply(question):  
...     question_tfidf = vectorizer.transform([question]).toarray()  
...     question_16d = svd.transform(question_tfidf)  
...     idx = question_16d.dot(tfidf_vectors_16d.T).argmax()  
...     print(  
...         f"Your question:\n{question}\n\n"  
...         f"Most similar FAQ question:\n{df['question'][idx]}\n"  
...         f"Answer to that FAQ question:\n{df['answer'][idx]}")  
...
```

Let's do a sanity check of our model and make sure it still can answer easy questions:

```
>>> bot_reply("What's overfitting a model?")  
Your question:  
    What's overfitting a model?  
Most similar FAQ question:  
    What is overfitting?
```

Answer to that FAQ question:

When your test set accuracy is significantly lower than your tr

Now, let's give our model a tougher nut to crack - like the question our previous model wasn't good in dealing with. Can it do better?

```
>>> bot_reply("How do I decrease overfitting for Logistic Regress  
Your question:
```

How do I decrease overfitting for Logistic Regression?

Most similar FAQ question:

How to reduce overfitting and improve test set accuracy for a L

Answer to that FAQ question:

Decrease the C value, this increases the regularization strengt

Wow! Looks like the new version of our bot was able to "realize" that 'decrease' and 'reduce' have similar meanings. Not only that, it was also able to "understand" that 'Logistic Regression' and "LogisticRegression" are very close - such simple step was almost impossible for our TF-IDF model.

Looks like we're getting closer to building a truly robust question answering system. We'll see in the next chapter how we can do even better than topic modeling!

4.10 What's Next?

In the next chapters, you'll learn how to fine tune this concept of topic vectors so that the vectors associated with words are more precise and useful. To do this we first start learning about neural nets. This will improve your pipeline's ability to extract meaning from short texts or even solitary words.

4.11 Review

- What preprocessing techniques would you use to prepare your text for more efficient topic modeling with LDiA? What about LSA?
- Can you think of a dataset/problem where TF-IDF performs better than LSA? What about the opposite?
- We mentioned filtering stopwords as a prep process for LDiA. When would this filtering be beneficial?
- The main challenge of semantic search is that the dense LSA topic

vectors are not inverse-indexable. Can you explain why it's so?

4.12 Summary

- You can derive the meaning of your words and documents by analyzing co-occurrence of terms in your dataset.
- SVD can be used for semantic analysis to decompose and transform TF-IDF and BOW vectors into topic vectors.
- Hyperparameter table can be used to compare the performances of different pipelines and models.
- Use LDiA when you need to conduct an explainable topic analysis.
- No matter how you create your topic vectors, they can be used for semantic search to find documents based on their meaning.

5 Word brain (neural networks)

This chapter covers

- Building a base layer for your neural networks
- Understanding backpropagation to train neural networks
- Implementing a basic neural network in Python
- Implementing a scalable neural network in PyTorch
- Stacking network layers for better data representation
- Tuning up your neural network for better performance

When you read the title of this chapter, "word brain", the neurons in your brain started firing, reminding you where you'd heard something like that before. And now that you read the word "heard", your neurons might be connecting the words in the title to the part of your brain that processes the *sound* of words. And maybe, the neurons in your audio cortex are starting to connect the phrase "word brain" to common phrases that rhyme with it, such as "bird brain."

Even if my brain didn't predict your brain very well, you're about to build a small brain yourself. And the "word brain" you are about to build will be a lot better than both of our human brains, at least for some particularly hard NLP tasks. You're going to build a tiny brain that can process a single word and predict something about what it means. And a neural net can do this when the word it is processing is a person's name and it doesn't seem to *mean* anything at all to a human.

Don't worry if all of this talk about brains and predictions and words has you confused. You are going to start simple, with just a single artificial neuron, built in Python. And you'll use PyTorch to handle all the complicated math required to connect your neuron up to other neurons and create an artificial neural network. Once you understand neural networks, you'll begin to understand *deep learning*, and be able to use it in the real world for fun, positive social impact, and ... if you insist, profit.

5.1 Why neural networks?

When you use a deep neural network for machine learning it is called *deep learning*. In the past few years deep learning has smashed through the accuracy and intelligence ceiling on many tough NLP problems:

- question answering
- reading comprehension
- summarization
- *natural language inference* (NLI)

And recently deep learning (deep neural networks) enabled previously unimaginable applications:

- long, engaging conversations
- companionship
- writing software

That last one, writing software, is particularly interesting, because NLP neural networks are being used to write software ... wait for it ... for NLP. This means that AI and NLP algorithms are getting closer to the day when they will be able to self-replicate and self-improve. This has renewed hope interest in neural networks as path towards *Artificial General Intelligence* (AGI) - or at least *more* generally intelligent machines. And NLP is already being used to directly generate software that is advancing the intelligence of those NLP algorithms. And that virtuous cycle is creating models so complex and powerful that humans have a hard time understanding them and explaining how they work. An OpenAI article shows a clear inflection point in the complexity of models that happened in 2012, when Geoffrey Hinton's improvement to neural network architectures caught on. Since 2012, the amount of compute used in the largest AI training runs has been increasing exponentially with a 3.4-month doubling time. [\[168\]](#) Neural networks make all this possible because they:

- Are better at generalizing from a few examples
- Can automatically engineer features from raw data
- Can be trained easily on any unlabeled text

Neural networks do the feature engineering for you, and they do it optimally. They extract generally useful features and representations of your data according to whatever problem you set up in your pipeline. And modern neural networks work especially well even for information rich data such as natural language text.

5.1.1 Neural networks for words

With neural networks you don't have to guess whether the proper nouns or average word length or hand crafted word sentiment scores are going to be what your model needs. You can avoid the temptation to use readability scores, or sentiment analyzers to reduce the dimensionality of your data. You don't even have to squash your vectors with blind (unsupervised) dimension reduction approaches such as stop word filtering, stemming, lemmatizing, LDA, PCA, TSNE, or clustering. A neural network *mini-brain* can do this for you, and it will do it optimally, based on the statistics of the relationship between words and your target.



Warning

Don't use stemmers, lemmatizers or other keyword-based preprocessing in your deep learning pipeline unless you're absolutely sure it is helping your model perform better for your application.

If you're doing stemming, lemmatization, or keyword based analyses you probably want to try your pipeline without those filters. It doesn't matter whether you use NLTK, Stanford Core NLP, or even SpaCy, hand-crafted linguistics algorithms like lemmatizers are probably not helping. These algorithms are limited by the hand-labeled vocabulary and hand-crafted regular expressions that define the algorithm.

Here are some preprocessing algorithms that will likely trip up your neural nets:

- Porter stemmer
- Penn Treebank lemmatizer
- Flesch-Kincaid readability analyzer

- VADER sentiment analyzer

In the hyperconnected modern world of machine learning and deep learning, natural languages evolve too rapidly and these algorithms can't keep up. Stemmers and lemmatizers are overfit to a bygone era. The words "hyperconnected" and "overfit" were nonexistent 50 years ago. Lemmatizers, stemmers, and sentiment analyzers often do the wrong thing with unanticipated words such as these.[\[169\]](#)

Deep learning is a game changer for NLP. In the past brilliant linguists like Julie Beth Lovins needed to hand-craft algorithms to extract stems, lemmas, and keywords from text.[\[170\]](#) (Her one-pass stemmer and lemmatizer algorithms were later made famous by Martin Porter and others)[\[171\]](#) Deep neural networks now make all that laborious work unnecessary. They directly access the meaning of words based on their statistics, without requiring brittle algorithms like stemmers and lemmatizers.

Even powerful feature engineering approaches like the Latent Semantic Analysis (LSA) of chapter 4 can't match the NLU capabilities of neural nets. And the automatic learning of decision thresholds with decision trees, random forests, and boosted trees don't provide the depth of language understanding of neural nets. Conventional machine learning algorithms made full text search and universally accessible knowledge a reality. But deep learning with neural networks makes artificial intelligence and intelligent assistants possible. You no longer needed an information retrieval expert or librarian to find what you were looking for, you have a virtual librarian to assist you. Deep Learning now powers your thinking in ways you wouldn't have imagined a few years ago.

What is it about deep layers of neurons that has propelled NLP to such prominence in our lives? Why is it that we are now so dependent on neural machine translation (NMT) recommendation engines, middle button suggestions (There's a subreddit where people post comments made entirely of middle button suggestions from their smartphones?[\[172\]](#)), and auto-reply nudges? If you've tried digital detox, you may have experienced this sensation of not being fully yourself without NLP helping you behind the scenes. And NLP neural nets for have given us hope that Artificial General

Intelligence (AGI) is within reach. They promise to allow machines to learn in the same way we often do, by just reading a lot of text.

The power of NLP that you learned to employ in the previous chapters is about to get a lot more powerful. You'll want to understand how deep, layered networks of artificial neurons work in order to ensure that your algorithms benefit society instead of destroy it. (Stuart Russell's *Human Compatible AI* explains the dangers and promise of AI and AGI, with some insightful NLP examples.) To wield this power for good, you need to get a feeling for how neural networks work all the way down deep at the individual neuron.

You'll also want to understand *why* they work so well for many NLP problems...and why they fail miserably on others.

We want to save you from the "AI winter" that discouraged researchers in the past. If you employ neural networks incorrectly you could get frost bitten by an overfit NLP pipeline that works well on your test data, but proves disastrous in the real world. As you get to understand how neural networks work, you will begin to see how you can build more *robust NLP* neural networks. Neural networks for NLP problems are notoriously brittle and vulnerable to adversarial attacks such as poisoning. (You can learn more about how to measure a model's robustness and improve it from Robin Jia's PhD thesis.^[173]) But first you must build an intuition for how a single neuron works.



Tip

Here are two excellent NL texts about processing NL text with neural networks. And you can even use these texts to train a deep learning pipeline to understand the terminology of NLP.

- *A Primer on Neural Network Models for Natural Language Processing* by Yoav Goldberg (<https://u.cs.biu.ac.il/~yogo/nlp.pdf>)
- *CS224d: Deep Learning for Natural Language Processing* by Richard Socher (<https://web.stanford.edu/class/cs224d/lectures/>)

You might also want to check *Deep learning for Natural Language Processing* by Stephan Raaijmakers on Manning.

(<https://www.manning.com/books/deep-learning-for-natural-language-processing>)

5.1.2 Neurons as feature engineers

One of the main limitations of linear regression, logistic regression, and naive Bayes models is that they all require that you to engineer features one by one. You must find the best numerical representation of your text among all the possible ways to represent text as numbers. Then you have to parameterize a function that takes in these engineered feature representations and outputs your predictions. Only then can the optimizer start searching for the parameter values that best predict the output variable.



Note

In some cases you will want to manually engineer threshold features for your NLP pipeline. This can be especially useful if you need an explainable model that you can discuss with your team and relate to real world phenomena. To create a simpler model with few engineered features, without neural networks, requires you to examine residual plots for each and every feature. When you see a discontinuity or nonlinearity in the residuals at a particular value of the feature, that's a good threshold value to add to your pipeline. In some cases you can even find an association between your engineered thresholds and real world phenomena.

For example the TF-IDF vector representation you used in chapter 3 works well for information retrieval and full text search. However TF-IDF vectors often don't generalize well for semantic search or NLU in the real world where words are used in ambiguous ways or misspelled. And the PCA or LSA transformation of chapter 4 may not find the right topic vector representation for your particular problem. They are good for visualization but not optimal for NLU applications. Multi-layer neural networks promise to do this feature engineering for you and do it in a way that's in some sense optimal. Neural networks search a much broader space of possible feature engineering

functions.

Dealing with the polynomial feature explosion

Another example of some feature engineering that neural networks can optimize for you is polynomial feature extraction. (Think back to the last time you used `sklearn.preprocessing.PolynomialFeatures`) During feature engineering, you may assume think the relationship between inputs and outputs is quadratic or cubic, then you must square pr cube all your features. And if you don't know which interactions might be critical to solving your problem, you have to multiply all your features by each other.

You know the depth and breadth of this rabbit hole. The number of possible fourth order polynomial features is virtually limitless. You might try to reduce the dimensions of your TF-IDF vectors from 10s of thousands to 100s of dimensions using PCA or LSA. But throwing in fourth order polynomial features would exponentially expand your dimensionality beyond even the dimensionality of TF-IDF vectors.

And even with millions of possible polynomial features, there are still millions more threshold features. Random forests of decision trees and boosted decision trees have advanced to the point that they do a decent job of feature engineering automatically. So finding the right threshold features is essentially a solved problem. But these feature representations are difficult to explain and sometimes don't generalize well to the real world. This is where neural nets can help.

The Holy Grail of feature engineering is finding representations that say something about the physics of the real world. If your features are explainable according to real world phenomena, you can begin to build confidence that it is more than just predictive. It may be a truly causal model that says something about the world that is true in general and not just for your dataset.

Peter Woit explains how the explosion of possible models in modern physics are mostly *Not Even Wrong*.^[174] These *not even wrong* models are what you create when you use `sklearn.preprocessing.PolynomialFeatures`. And

that is a real problem. Very few of the millions of these extracted polynomial features are even physically possible In other words the vast majority of polynomial features are just noise.[\[175\]](#)



Important

For any machine learning pipeline, make sure your polynomial features never include the multiplication of more than 2 physical quantities. For example $x_1 * (w_2) 3$ is a legitimate fourth order polynomial features to try. However, polynomial features that multiple more than two quantities together, such as $x_1 * x_2 * (w_3) 2$ are not physically realizable and should be weeded out of your pipeline.

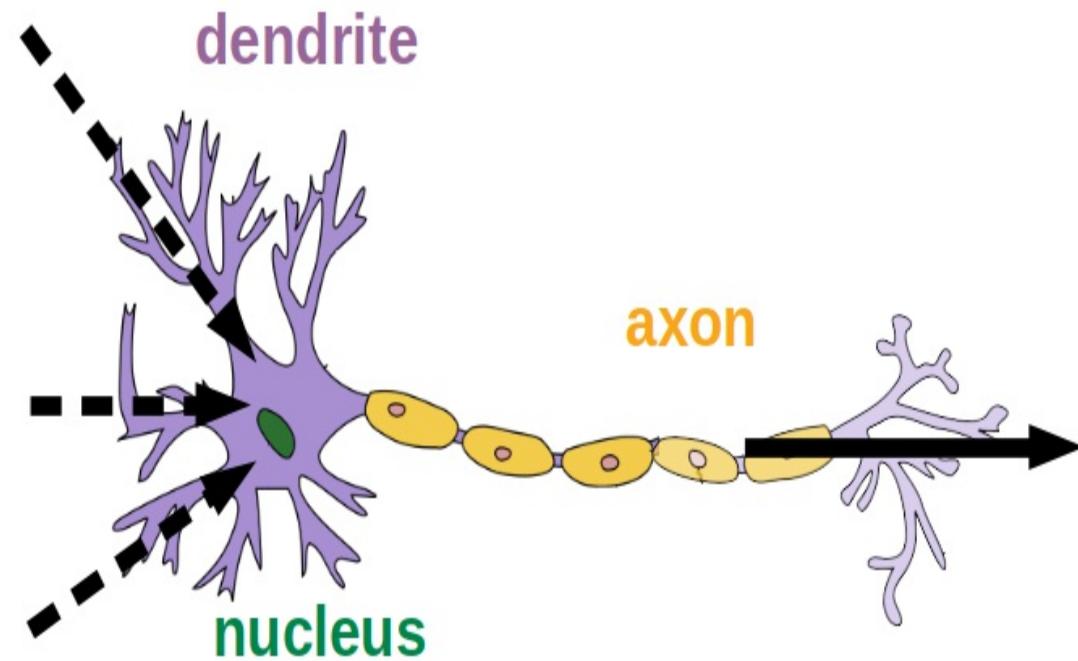
We hope that by now you're inspired by the possibilities that neural networks offer. Let's start our journey into the world of neural networks building single neurons that look a lot like logistic regressions. Ultimately you will be able to combine and stack these neurons in layers that optimize the feature engineering for you.

5.1.3 Biological neurons

Frank Rosenblatt came up with the first artificial neural network based on his understanding of how biological neurons in our brains work. He called it a perceptron because he was using it to help machines perceive their environment using sensor data as input.[\[176\]](#) He hoped they would revolutionize machine learning by eliminating the need to hand-craft filters to extract features from data. He also wanted to automate the process of finding the right combination of functions for any problem.

He wanted to make it possible for engineers to build AI systems without having to design specialized models for each problem. At the time, engineers used linear regressions, polynomial regressions, logistic regressions and decision trees to help robots make decisions. Rosenblatt's perceptron was a new kind of machine learning algorithm that could approximate any function, not just a line, a logistic function, or a polynomial.[\[177\]](#) He based it on how biological neurons work.

Figure 5.1. Biological neuron cell



Rosenblatt was building on a long history of successful logistic regression models. He was modifying the optimization algorithm slightly to better mimic what neuroscientists were learning about how biological neurons adjust their response to the environment over time.

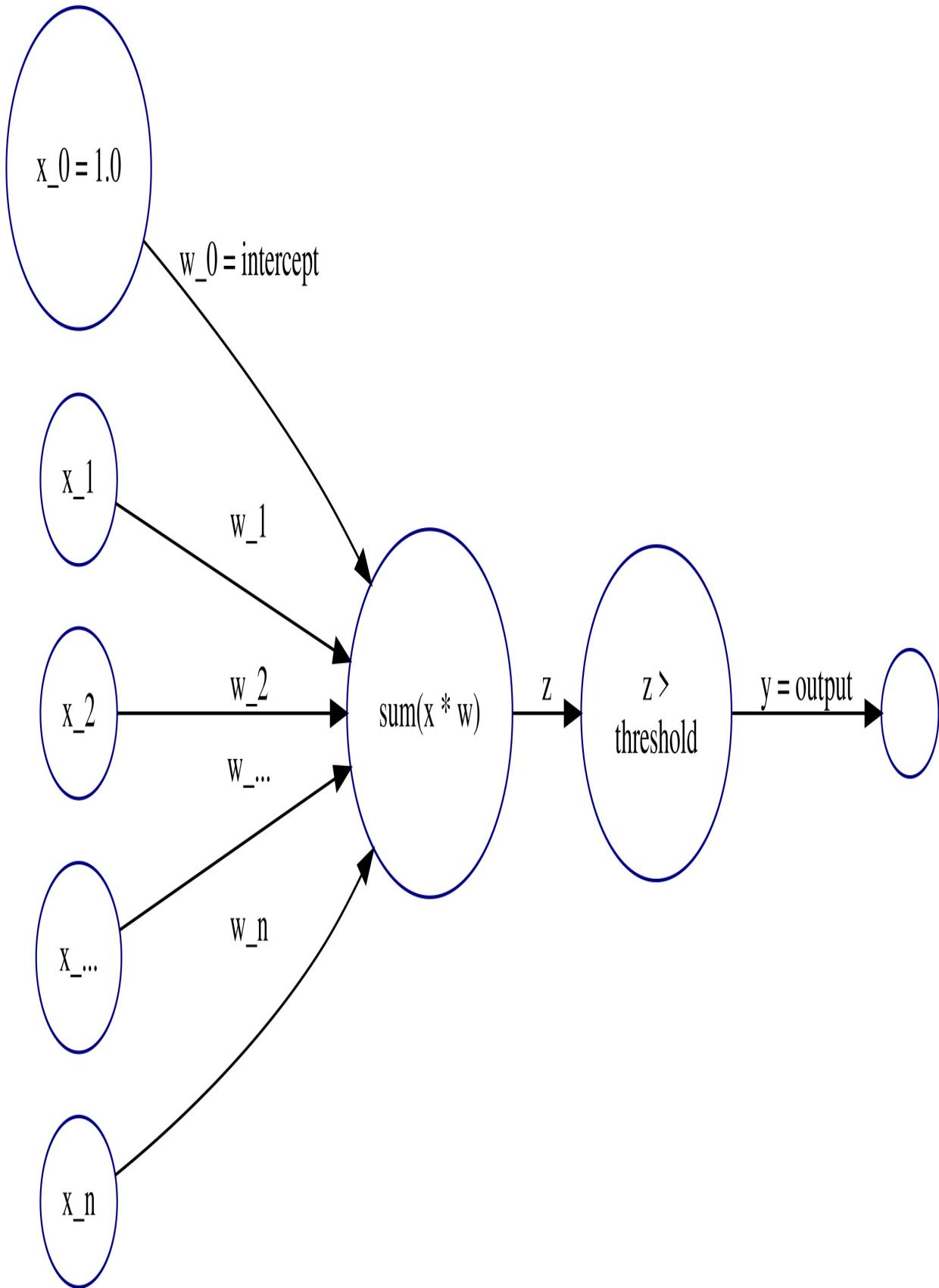
Electrical signals flow into a biological neuron in your brain through the *dendrites* (see figure 5.1) and into the nucleus. The nucleus accumulates electric charge and it builds up over time. When the accumulated charge in the nucleus reaches the activation level of that particular neuron it *fires* an electrical signal out through the *axon*. However, neurons are not all created equal. The dendrites of the neuron in your brain are more "sensitive" for some neuron inputs than for others. And the nucleus itself may have a higher or lower activation threshold depending on its function in the brain. So for some more sensitive neurons it takes less of a signal on the inputs to trigger the output signal being sent out the axon.

So you can imagine how neuroscientists might measure the sensitivity of individual dendrites and neurons with experiments on real neurons. And this

sensitivity can be given a numerical value. Rosenblatt's perceptron abstracts this biological neuron to create an artificial neuron with a *weight* associated with each input (dendrite). For artificial neurons, such as Rosenblatt's perceptron, we represent the sensitivity of individual dendrites as a numerical *weight* or *gain* for that particular path. A biological cell *weights* incoming signals when deciding when to fire. A higher weight represents a higher sensitivity to small changes in the input.

A biological neuron will dynamically change those weights in the decision making process over the course of its life. You are going to mimic that biological learning process using the machine learning process called *back propagation*.

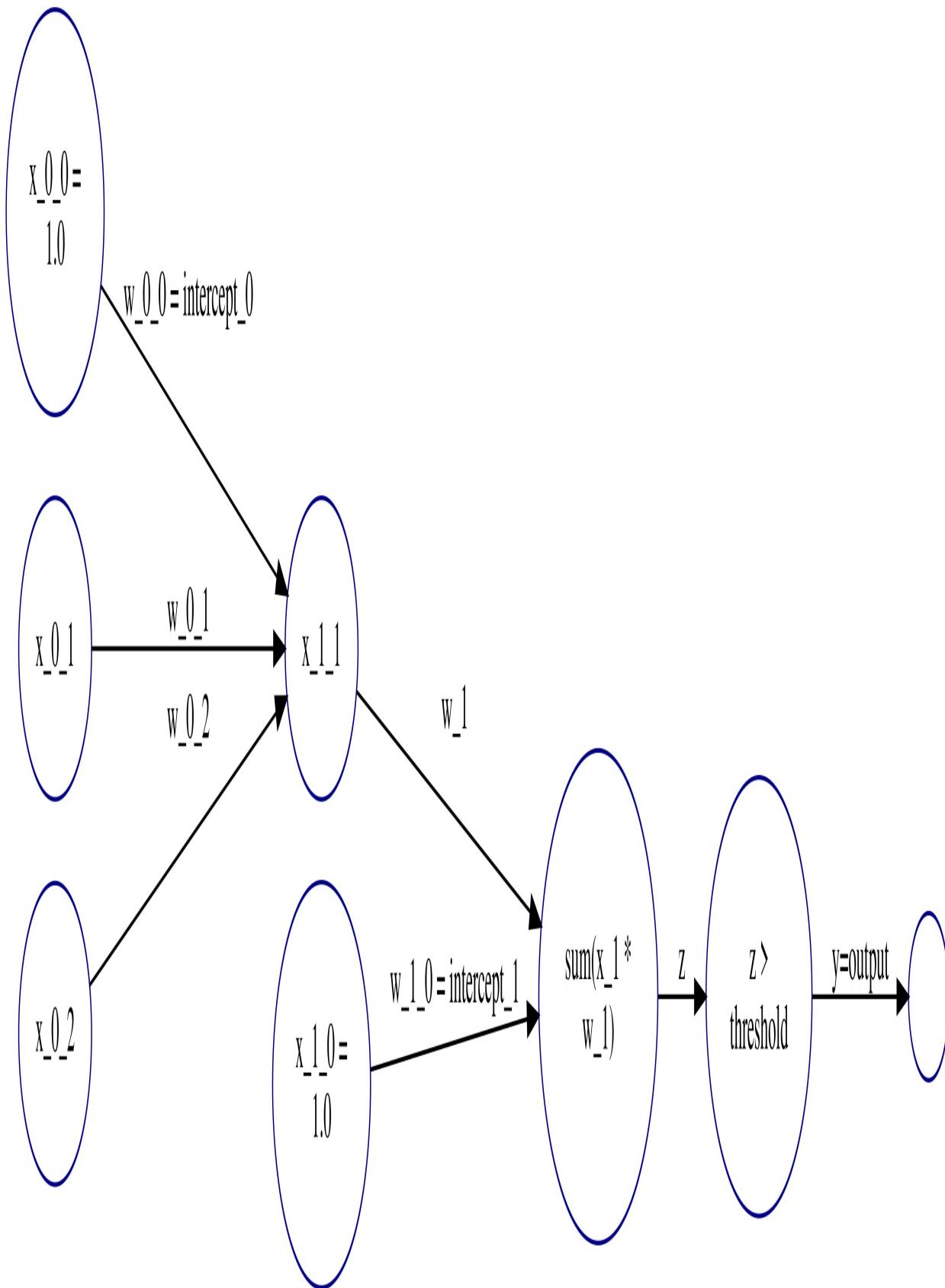
Figure 5.2. Basic perceptron



AI researchers hoped to replace the rigid math of logistic regressions and linear regressions and polynomial feature extraction with the more fuzzy and generalized logic of neural networks—tiny brains. Rosenblatt's artificial neurons even worked for trigonometric functions and other highly nonlinear functions. Each neuron solved one part of the problem and could be combined with other neurons to learn more and more complex functions. (Though not all of them - even simple functions, like an XOR gate can't be solved with a single layer perceptron). He called this collection of artificial neurons a perceptron.

Rosenblatt didn't realize it at the time, but his artificial neurons could be layered up just as biological neurons connect to each other in clusters. In modern *deep learning* we connect the predictions coming out of one group of neurons to another collection of neurons to refine the predictions. This allows us to create layered networks that can model *any* function. They can now solve any machine learning problem ... if you have enough time and data.

Figure 5.3. Neural network layers



5.1.4 Perceptron

One of the most complex things neurons do is process language. Think about how a perceptron might be used to process natural language text. Does the math shown in figure 5.2 remind you of any of the machine learning models you've used before? What machine learning models do you know of that multiply the input features by a vector of weights or coefficients? Well that would be a linear regression. But what if you used a sigmoid activation function or logistic function on the output of a linear regression? It's starting to look a lot like a *logistic regression* to me.

The sigmoid *activation function* used in a perceptron is actually the same as the logistic function used within logistic regression. Sigmoid just means s-shaped. And the logistic function has exactly the shape we want for creating a soft threshold or logical binary output. So really what your neuron is doing here is equivalent to a logistic regression on the inputs.

This is the formula for a logistic function implemented in python.

```
>>> def logistic(x, w=1., phase=0, gain=1):
...     return gain / (1. + np.exp(-w * (x - phase)))
```

And here is what a logistic function looks like, and how the coefficient (weight) and phase (intercept) affect its shape.

```
>>> import pandas as pd
>>> import numpy as np
>>> import seaborn as sns
>>> sns.set_style()

>>> xy = pd.DataFrame(np.arange(-50, 50) / 10., columns=['x'])
>>> for w, phase in zip([1, 3, 1, 1, .5], [0, 0, 2, -1, 0]):
...     kwargs = dict(w=w, phase=phase)
...     xy[f'{kwargs}'] = logistic(xy['x'], **kwargs)
>>> xy.plot(grid="on", ylabel="y")
```

What were your inputs when you did a logistic regression on natural language sentences in earlier chapters? You first processed the text with a keyword detector, CountVectorizer, or TfidfVectorizer. These models use

a tokenizer, like the ones you learned about in chapter 2 to split the text into individual words, and then count them up. So for NLP it's common to use the BOW counts or the TF-IDF vector as the input to an NLP model, and that's true for neural networks as well.

Each of Rosenblatt's input weights (biological dendrites) had an adjustable value for the weight or sensitivity of that signal. Rosenblatt implemented this weight with a potentiometer, like the volume knob on an old fashioned stereo receiver. This allowed researchers to manually adjust the sensitivity of their neuron to each of its inputs individually. A perceptron can be made more or less sensitive to the counts of each word in the BOW or TF-IDF vector by adjusting this sensitivity knob.

Once the signal for a particular word was increased or decreased according to the sensitivity or weight it passed into the main body of the biological neuron cell. It's here in the body of the perceptron, and also in a real biological neuron, where the input signals are added together. Then that signal is passed through a soft thresholding function like a sigmoid before sending the signal out the axon. A biological neuron will only *fire* if the signal is above some threshold. The sigmoid function in a perceptron just makes it easy to implement that threshold at 50% of the min-max range. If a neuron doesn't fire for a given combination of words or input signals, that means it was a negative classification match.

5.1.5 A Python perceptron

So a machine can simulate a really simple neuron by multiplying numerical features by "weights" and combining them together to create a prediction or make a decision. These numerical features represent your object as a numerical vector that the machine can "understand". For the home price prediction problem of Zillow's zestimate, how do you think they might build an NLP-only model to predict home price? But how do you represent the natural language description of a house as a vector of numbers so that you can predict its price? You could take a verbal description of the house and use the counts of each word as a feature, just as you did in chapter 2 and 3. Or you could use a transformation like PCA to compress these thousands of dimensions into topic vectors, as you did with PCA in chapter 4.

But these approaches are just a guess at which features are important, based on the variability or variance of each feature. Perhaps the key words in the description are the numerical values for the square footage and number of bedrooms in the home. Your word vectors and topic vectors would miss these numerical values entirely.

In "normal" machine learning problems, like predicting home prices, you might have structured numerical data. You will usually have a table with all the important features listed, such as square footage, last sold price, number of bedrooms, and even latitude and longitude or zip code. For natural language problems, however, we want your model to be able to work with unstructured data, text. Your model has to figure out exactly which words and in what combination or sequence are predictive of your target variable. Your model must read the home description, and, like a human brain, make a guess at the home price. And a neural network is the closest thing you have to a machine that can mimic some of your human intuition.

The beauty of deep learning is that you can use as your input every possible feature you can dream up. This means you can input the entire text description and have your transformer produce a high dimensional TF-IDF vector and a neural network can handle it just fine. You can even go higher dimensional than that. You can pass it the raw, unfiltered text as 1-hot encoded sequences of words. Do you remember the piano roll we talked about in chapter 2? Neural networks are made for these kinds of raw representations of natural language data.

Shallow learning

For your first deep learning NLP problem, you will keep it shallow. To understand the magic of deep learning it helps to see how a single neuron works. A single neuron will find a *weight* for each feature you input into the model. You can think of these weights as a percentage of the signal that is let into the neuron. If you're familiar with linear regression, then you probably recognize these diagrams and can see that the weights are just the slopes of a linear regression. And if you throw in a logistic function, these weights are the coefficients that a logistic regression learns as you give it examples from your dataset. To put it in different words, the weights for the inputs to a

single neuron are mathematically equivalent to the slopes in a multivariate linear regression or logistic regression.



Tip

Just as with the SciKit-Learn machine learning models, the individual features are denoted as x_i or in Python as $x[i]$. The i is an indexing integer denoting the position within the input vector. And the collection of all features for a given example are within the vector \mathbf{x} .

$$\mathbf{x} = x_1, x_2, \dots, x_i, \dots, x_n$$

And similarly, you'll see the associate weights for each feature as w_i , where i corresponds to the integer in x . And the weights are generally represented as a vector \mathbf{W}

$$\mathbf{w} = w_1, w_2, \dots, w_i, \dots, w_n$$

With the features in hand, you just multiply each feature (x_i) by the corresponding weight (w_i) and then sum up.

$$y = (x_1 * w_1) + (x_2 * w_2) + \dots + (x_i * w_i)$$

Here's a fun, simple example to make sure you understand this math. Imagine an input BOW vector for a phrase like "green egg egg ham ham ham spam spam spam spam":

```
>>> from collections import Counter  
  
>>> np.random.seed(451)  
>>> tokens = "green egg egg ham ham ham spam spam spam spam".split()  
>>> bow = Counter(tokens)  
>>> x = pd.Series(bow)  
>>> x  
green      1  
egg        2  
ham        3  
spam       4  
  
>>> x1, x2, x3, x4 = x
```

```

>>> x1, x2, x3, x4
(1, 2, 3, 4)

>>> w0 = np.round(.1 * np.random.randn(), 2)
>>> w0
0.07
>>> w1, w2, w3, w4 = (.1 * np.random.randn(len(x))).round(2)
>>> w1, w2, w3, w4
(0.12, -0.16, 0.03, -0.18)

>>> x = np.array([1, x1, x2, x3, x4])      #1
>>> w = np.array([w0, w1, w2, w3, w4])      #2
>>> y = np.sum(w * x)      #3
>>> y
-0.76

```

So this 4-input, 1-output, single-neuron network outputs a value of -0.76 for these random weights in a neuron that hasn't yet been trained.

There's one more piece you're missing here. You need to run a nonlinear function on the output (y) to change the shape of the output so it's not just a linear regression. Often a thresholding or clipping function is used to decide whether the neuron should fire or not. For a thresholding function, if the weighted sum is above a certain threshold, the perceptron outputs 1. Otherwise it outputs 0. You can represent this threshold with a simple *step function* (labeled "Activation Function" in figure 5.2).

Here's the code to apply a step function or thresholding function to the output of your neuron:

```

>>> threshold = 0.0
>>> y = int(y > threshold)

```

And if you want your model to output a continuous probability or likelihood rather than a binary 0 or 1, you probably want to use the logistic activation function that we introduced earlier in this chapter.[\[178\]](#)

```
>>> y = logistic(x)
```

A neural network works like any other machine learning model—you present it with numerical examples of inputs (feature vectors) and outputs (predictions) for your model. And like a conventional logistic regression, the

neural network will use trial and error to find the weights on your inputs that create the best predictions. Your *loss function* will measure how much error your model has.

Make sure this Python implementation of the math in a neuron makes sense to you. Keep in mind, the code we've written is only for the *feed forward* path of a neuron. The math is very similar to what you would see in the `LogisticRegression.predict()` function in SciKit-Learn for a 4-input, 1-output logistic regression.[\[179\]](#)



Note

A *loss function* is a function that outputs a score to measure how bad your model is, the total error of its predictions. An *objective function* is just measures how good your model is based on how small the error is. A *loss function* is like the percentage of questions a student got wrong on a test. An *objective function* is like the grade or percent score on that test. You can use either one to help you learn the right answers and get better and better on your tests.

Why the extra weight?

Did you notice that you have one additional weight, w_0 ? There is no input labeled x_0 . So why is there a w_0 ? Can you guess why we always give our neural neurons an input signal with a constant value of "1.0" for x_0 ? Think back to the linear and logistic regression models you have built in the past. Do you remember the extra coefficient in the single-variable linear regression formula?

$$y = m * x + b$$

The y variable is for the output or predictions from the model. The x variable if for the single independent feature variable in this model. And you probably remember that m represents the slope. But do you remember what b is for?

$$y = \text{slope} * x + \text{intercept}$$

Now can you guess what the extra weight w_0 is for, and why we always make sure it isn't affected by the input (multiply it by an input of 1.0)?

$$w_0 * 1.0 + w_1 * x_1 + \dots + (x_n * w_n)$$

It's the *intercept* from your linear regression, just "rebranded" as the *bias* weight (w_0) for this layer of a neural network.

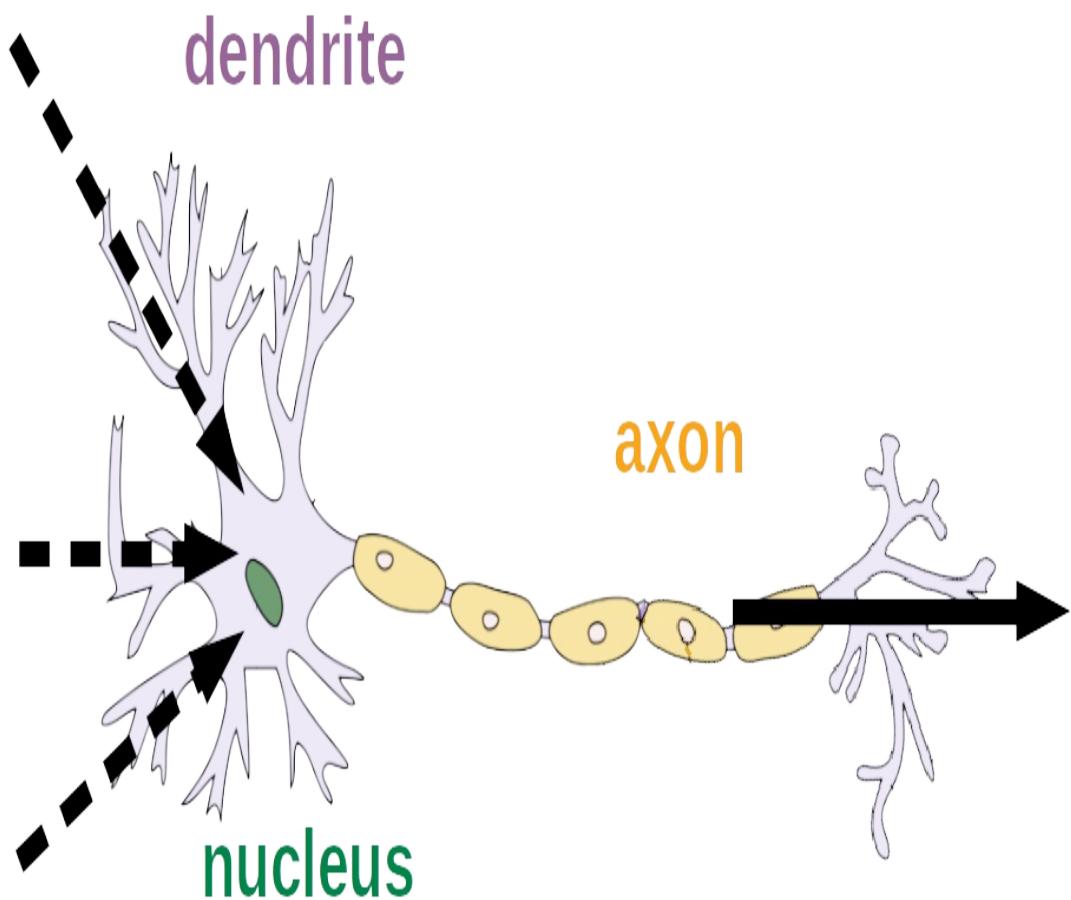
Figure 5.2 and this example reference *bias*. What is this? The bias is an "always on" input to the neuron. The neuron has a weight dedicated to it just as with every other element of the input, and that weight is trained along with the others in the exact same way. This is represented in two ways in 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.

The reason for having the bias weight at all is that you need the neuron to be resilient to inputs of all zeros. It may be the case that the network needs to learn to output 0 in the face of inputs of 0, but it may not. Without the bias term, the neuron would output $0 * \text{weight} = 0$ for any weights you started with or tried to learn. With the bias term, you wouldn't 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.

Figure 5.3 is a rather neat visualization of the analogy between some of the signals within a biological neuron in your brain and the signals of an artificial neuron used for deep learning. If you want to get deep, think about how you are using a biological neuron to read this book about natural language processing to learn about deep learning.

Figure 5.4. A perceptron and a biological neuron

Biological Neuron



The Python for the simplest possible single neuron looks like this:

```
>>> def neuron(x, w):
...     z = sum(wi * xi for xi, wi in zip(x, w)) #1
...     return z > 0      #2
```

Perhaps you are more comfortable with numpy and *vectorized* mathematical operations like you learned about in linear algebra class.

```
>>> def neuron(x, w):
...     z = np.array(wi).dot(w)
...     return z > 0
```



Note

Any Python conditional expression will evaluate to a `True` or `False` boolean value. If you use that `bool` type in a mathematical operation such as addition or multiplication, Python will *coerce* a `True` value into a numerical `int` or `float` value of `1` or `1.0`. A `False` value is coerced into a `1` or `0` when you multiply boolean by, or add it to another number.

The `w` variable contains the vector of weight parameters for the model. These are the values that will be learned as the neuron's outputs are compared the desired outputs during training. The `x` variable contains the vector of signal values coming into the neuron. This is the feature vector, such as a TF-IDF vector for a natural language model. For a biological neuron the inputs are the rate of electrical pulses rippling through the dendrites. The input to one neuron is often the output from another neuron.



Tip

The sum of the pairwise multiplications of the inputs (`x`) and the weights (`w`) is exactly the same as the dot product of the two vectors `x` and `y`. If you use numpy, a neuron can be implemented with a single brief Python expression: `w.dot(x) > 0`. This is why *linear algebra* is so useful for neural networks. Neural networks are mostly just dot products of parameters by inputs. And GPUs are computer processing chips designed to do all the multiplications and additions of these dot products in parallel, one operation on each GPU core. So a 1-core GPU can often perform a dot product 250 times faster than

a 4-core CPU.

If you are familiar with the natural language of mathematics, you might prefer the summation notation:

Equation 5.1 Threshold activation function

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

Your perceptron hasn't *learned* anything just yet. But you have achieved something quite important. You've passed data into a model and received an output. That output is likely wrong, given you said nothing about where the weight values come from. But this is where things will get interesting.



Tip

The base unit of any neural network is the neuron. And the basic perceptron is a special case of the more generalized neuron. We refer to the perceptron as a neuron for now, and come back to the terminology when it no longer applies.

[168] See analysis by Dario Amodei and Danny Hernandez here
(<https://openai.com/blog/ai-and-compute/>)

[169] See the lemmatizing FAQ chatbot example in chapter 3 failed on the question about "overfitting."

[170] Wikipedia article about Julie Beth Lovins:
https://en.wikipedia.org/wiki/Julie_Beth_Lovins

[171] <https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html>

[172] <https://proai.org/middle-button-subreddit>

[173] Robin Jia, *Building Robust NLP Systems*
(https://robinjia.GitHub.io/assets/pdf/robinjia_thesis.pdf)

[174] *Not Even Wrong: The Failure of String Theory and the Search for Unity in Physical Law* by Peter Woit

[175] Lex Fridman interview with Peter Woit (<https://lexfridman.com/peter-woit/>)

[176] Rosenblatt, Frank (1957), The perceptron—a perceiving and recognizing automaton. Report 85-460-1, Cornell Aeronautical Laboratory.

[177] https://en.wikipedia.org/wiki/Universal_approximation_theorem

[178] The logistic activation function can be used to turn a linear regression into a logistic regression: (https://scikit-learn.org/stable/auto_examples/linear_model/plot_logistic.html)

[179] https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression

5.2 Example logistic neuron

It turns out you’re already familiar with a very common kind of perceptron or neuron. When you use the logistic function for the *activation function* on a neuron, you’ve essentially created a logistic regression model. A single neuron with the logistic function for its activation function is mathematically equivalent to the LogisticRegression model in SciKit-Learn. The only difference is how they’re trained. So you are going to first train a logistic regression model and compare it to a single-neuron neural network trained on the same data.

5.2.1 The logistics of clickbait

Software (and humans) often need to make decisions based on logical

criteria. For example, many times a day you probably have to decide whether to click on a particular link or title. Sometimes those links lead you to a fake news article. So your brain learns some logical rules that it follows before clicking on a particular link.

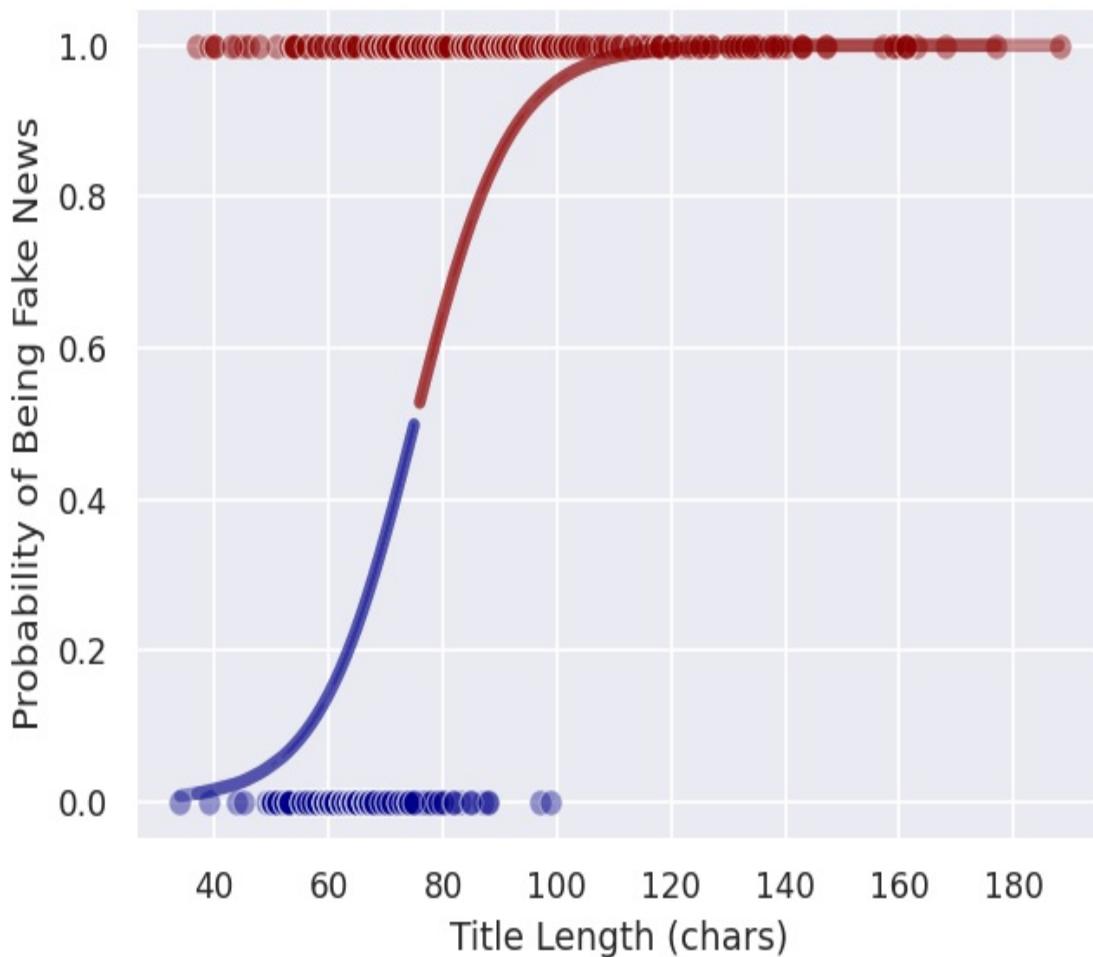
- Is it a topic you're interested in?
- Does the link look promotional or spammy?
- Is it from a reputable source that you like?
- Does it look true or factual?

Each one of these decisions could be modeled in an artificial neuron within a machine. And you could use that model to create a logic gate in a circuit board or a conditional expression (`if` statement) in software. If you did this with artificial neurons, the smallest artificial "brain" you could build to handle these 4 decisions would use 4 logistic regression gates.

To mimic your brain's *clickbait* filter you might decide to train a logistic regression model on the length of the headline. Perhaps you have a hunch that longer headlines are more likely to be sensational and exaggerated. Here's a scatter plot of fake and authentic news headlines and their headline length in characters.

The neuron input weight is equivalent to the maximum slope in the middle of the logistic regression plot in figure 5.3 for a fake news classifier with the single feature, title length.

Figure 5.5. Logistic regression - fakeness vs title length



5.2.2 Sex education

How's that for clickbait? Because the fake news (clickbait) dataset has been fully exploited on Kaggle, you're going to switch to a more fun and useful dataset. You're going to predict the sex of a name with perceptrons (artificial neurons).

The problem you're going to solve with this simple architecture is an everyday NLU problem that your brain's millions of neurons try to solve every day. Your brain is strongly incentivized to identify the birth sex of the people you interact with on social media. (If you're interested in why this is, Richard McElreath and Robert Boyd have a fascinating book on the subject. [\[180\]](#)) A single artificial neuron can solve this challenge with about 80%

accuracy using only the characters in the first name of a person. You're going to use a sample of names from a database of 317 million birth certificates across US states and territories over more than 100 years.

Biologically, identifying someone's sex is useful to your genes because they only survive if you reproduce them by finding a sexual partner to blend your genes with. Social interaction with other humans is critical to your genes' existence and survival. And your genes are the blueprint for your brain. So your brain is likely to contain at least a few neurons dedicated to this critical task. And you're going to find out how many artificial neurons it takes to predict the sex associated with a baby's given name (first name).



Sex

The word *sex* here refers to the label a doctor assigns to a baby at birth. In the US, the name, sex and date of birth are recorded on a birth certificate according to the laws that state. And the sex category is subject to interpretation and judgement by the person that fills out and signs the birth certificate. In datasets derived from US birth certificates, "sex at birth" is usually equivalent to one's *genetic sex*, but that is not always the case. It is possible to create a relatively well-defined "genetic sex" category based on the presence of XX chromosomes (female) or XY chromosomes (male). But biology and life has a way of blurring the boundaries of even this seemingly precise definition of "genetic sex".

Male and female are not the last word in *birth sex* classification. The CDC (Center for Disease Control) in recommends that USCDI (US Core Data Interoperability) standards include several nonbinary sex categories for clinical or medical use.^[181] In addition to 'female' and 'male', the categories 'unknown', and 'something not listed (specify)' are recommended by most western medical systems.

You want to make sure that your test set names don't appear anywhere in your training set. You also want to make sure that your test set only has one "right" label for each name. But this isn't what you think. There is not one correct binary sex label for any particular name. There is indeed a correct probability score (continuous value) of maleness or femaleness of a name

based on the ratio of the counts of names with a particular sex designation on their birth certificates. But that "correct" score will change as you add new examples to your dataset. Natural language processing is messy and fluid because the natural world and the language that describes it is dynamic and impossible to "pin on the wall."^[182]

This will enable the possibility that your model could *theoretically* achieve 100% accuracy. Obviously this isn't really possible for a problem like this where even humans can't achieve 100% accuracy. But your accuracy on the test set will tell you how close you are to this ideal, but only if you delete the duplicate names from your test set.

5.2.3 Pronouns and gender vs sex

Some states in the US allow one to indicate their child's *gender* on a birth certificate. Gender is often what people use to decide what pronouns they prefer. And there are various ways that people think about their gender. There's the apparent gender that they present to the world and there's the gender identity that they assign to themselves at various stages of their lives. Identifying either of these genders is a sensitive subject because it is fraught with legal and social ramifications. In many repressive cultures it can even be a matter of life and death. And gender is a very difficult thing to predict for a machine learning algorithm. For this chapter we utilized a simplified binary sex dataset to prepare the scaffolding you need to build your natural language processing skills from the ground up.

And there are practical uses for sex-estimation model even for machines that don't need it to spread their genes. A sex estimation model can be used to solve an important and difficult challenge in NLP called *coreference resolution*.^[183] Coreference resolution is when an NLP algorithm identifies the object or words associated with pronouns in natural language text. For example consider the pronouns in these sentences: "Maria was born in Ukraine. Her father was a physicist. 15 years later she left there for Israel." You may not realize it, but you resolved three coreferences in the blink of an eye. Your brain did the statistics on the likelihood that "Maria" was a "she/her" and that "Ukraine" is a "there".

Coreference resolution isn't always that easy, for machines or for humans. It is more difficult to do in languages where pronouns do not have gender. It can be even more difficult in languages with pronouns that do not discriminate between people and inanimate objects. Even languages with genderless objects like English sometimes arbitrarily assign gender to important things, such as sailing ships. Ships are referred to with feminine pronouns such as "she" and "her." And they are often given feminine names.

So knowing the sex associated with the names of people (and ships) in your text can be helpful in improving your NLU pipeline. This can be helpful even when that sex identification is a poor indicator of the presented gender of a person mentioned in the text. The author of text will often expect you to make assumptions about sex and gender based on names. In gender-bending SciFi novels, visionary authors like Gibson use this to keep you on your toes and expand your mind. [\[184\]](#)



Important

Make sure your NLP pipelines and chatbots are kind, inclusive and accessible for all human beings. In order to ensure your algorithms are unbiased you can *normalize* for any sex and gender information in the text data you process. In the next chapter you will see all the surprising ways in which sex and gender can affect the decisions your algorithms make. And you will see how gender affects the decisions of businesses or employers you deal with every day.

5.2.4 Sex logistics

First, import Pandas and set the `max_rows` to display only a few rows of your `DataFrames`.

```
>>> import pandas as pd  
>>> import numpy as np  
>>> pd.options.display.max_rows = 7
```

Now download the raw data from the `nlpia2` repository and sample only 10,000 rows, to keep things fast on any computer.

```

>>> np.random.seed(451)
>>> df = pd.read_csv(      #1
...     'https://proai.org/baby-names-us.csv.gz')
>>> df.to_csv(      #2
...     'baby-names-us.csv.gz', compression='gzip')
>>> df = df.sample(10_000)      #3
>>> df

```

The data spans more than 100 years of US birth certificates, but only includes the baby's first name:

```

[cols=",,,,,,",options="header",]
|===
|   |region |sex  |year |name |count |freq
| 6139665 |WV  |F    |1987 |Brittani |10  |0.000003
| 2565339 |MD  |F    |1954 |Ida   |18  |0.000005
| 22297  |AK  |M    |1988 |Maxwell |5   |0.000001
| ...   |...  |...  |...  |...   |...  |...
| 4475894 |OK  |F    |1950 |Leah  |9   |0.000003
| 5744351 |VA  |F    |2007 |Carley |11  |0.000003
| 5583882 |TX  |M    |2019 |Kartier |10  |0.000003
|===
10000 rows × 6 columns

```

You can ignore the region and birth year information for now. You only need the natural language name to predict sex with reasonable accuracy. If you're curious about names, you can explore these variables as features or targets. Your target variable will be sex ('M' or 'F'). There are no other sex categories provided in this dataset besides male and female.

You might enjoy exploring the dataset to discover how often your intuition about the names parents choose for their babies. Machine learning and NLP are a great way to dispell stereotypes and misconceptions.

```

>>> df.groupby(['name', 'sex'])['count'].sum()[('Timothy', )]
sex
F      5
M    3538

```

That's what makes NLP and DataScience so much fun. It gives us a broader view of the world that breaks us out of the limited perspective of our biological brains. I've never met a woman named "Timothy" but at least .1%

of babies named Timothy in the US have female on their birth certificate.

To speed the model training, you can aggregate (combine) your data across regions and years if those are not aspects of names that you'd like your model to predict. You can accomplish this with a Pandas DataFrame's `.groupby()` method.

```
>>> df = df.set_index(['name', 'sex'])
>>> groups = df.groupby(['name', 'sex'])
>>> counts = groups['count'].sum()
>>> counts
name    sex
Aaden    M      51
Aahana   F      26
Aahil    M       5
...
Zvi      M       5
Zya      F       8
Zylah   F       5
```

Because we've aggregated the numerical data for the column "count", the `counts` object is now a Pandas Series object rather than a DataFrame. It looks a little funny because we created a multilevel index on both name and sex. Can you guess why?

Now the dataset looks like an efficient set of examples for training a logistic regression. In fact, if we only wanted to predict the likely sex for the names in this database, we could just use the max count (the most common usage) for each name.

But this is a book about NLP and NLU (Natural Language Understanding). You'd like your models to *understand* the text of the name in some way. And you'd like it to work on odd names that are not even in this database, names such as "Carlana", a portmanteau of "Carl" and "Ana", her grandparents, or one-of-a-kind names such as "Cason." Examples that are not part of your training set or test set are called "out of distribution." In the real world your model will almost always encounter words and phrases never seen before. It's called "generalization" when a model can extrapolate to these out of distribution examples.

But how can you tokenize a single word like a name so that your model can generalize to completely new made-up names that it's never seen before? You can use the character n-grams within each word (or name) as your tokens. You can set up a `TfidfVectorizer` to count characters and character n-grams rather than words. You can experiment with a wider or narrower `ngram_range` but 3-grams are a good bet for most TF-IDF-based information retrieval and NLU algorithms. For example the state-of-the-art database PostgreSQL defaults to character 3-grams for its full-text search indexes. In later chapters you'll even use word piece and sentence piece tokenization which can optimally select a variety of character sequences to use as your tokens.

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer  
>>> vectorizer = TfidfVectorizer(use_idf=False,      #1  
...     analyzer='char', ngram_range=(1, 3))  
>>> vectorizer  
TfidfVectorizer(analyzer='char', ngram_range=(1, 3), use_idf=Fals
```

But now that you've indexed our names series by name *and* sex aggregating counts across states and years, there will be fewer unique rows in your Series. You can deduplicate the names before calculating TF-IDF document frequencies and character n-gram term frequencies.

```
>>> df = pd.DataFrame([list(tup) for tup in counts.index.values],  
...                     columns=['name', 'sex'])  
>>> df['count'] = counts.values  
>>> df  
    name  sex  counts  
0     Aaden   M      51  
1     Aahana   F      26  
2     Aahil   M       5  
...     ...  ..    ...  
4235      Zvi   M       5  
4236      Zya   F       8  
4237     Zylah   F       5  
  
[4238 rows x 3 columns]
```

You've aggregated 10,000 name-sex pairs into only 4238 unique name-sex pairings. Now you are ready to split the data into training and test sets.

```
>>> df['istrain'] = np.random.rand(len(df)) < .9
```

```
>>> df
      name  sex  counts  istrain
0     Aaden    M      51   True
1    Aahana    F      26   True
2     Aahil    M       5   True
...
4235     Zvi    M       5   True
4236     Zya    F       8   True
4237   Zylah    F       5   True
```

To ensure you don't accidentally swap the sexes for any of the names, recreate the name, sex multiindex:

```
>>> df.index = pd.MultiIndex.from_tuples(
...     zip(df['name'], df['sex']), names=['name_', 'sex_'])
>>> df
      name  sex  count  istrain
name_  sex_
Aaden  M     Aaden    M      51   True
Aahana F     Aahana    F      26   True
Aahil  M     Aahil    M       5   True
...
Zvi    M     Zvi     M       5   True
Zya    F     Zya     F       8   True
Zylah  F     Zylah    F       5   True
```

As you saw earlier, this dataset contains conflicting labels for many names. In real life, many names are used for both male and female babies (or other human sex categories). Like all machine learning classification problems, the math treats it as a regression problem. The model is actually predicting a continuous value rather than a discrete binary category. Linear algebra and real life only works on real values. In machine learning all dichotomies are false.[\[185\]](#) Machines don't think of words and concepts as hard categories, so neither should you.

```
>>> df_most_common = {}      #1
>>> for name, group in df.groupby('name'):
...     row_dict = group.iloc[group['count'].argmax()].to_dict()
...     df_most_common[(name, row_dict['sex'])] = row_dict
>>> df_most_common = pd.DataFrame(df_most_common).T      #3
```

Because of the duplicates the test set flag can be created from the not of the istrain.

```

>>> df_most_common['istest'] = ~df_most_common['istrain'].astype(
>>> df_most_common
      name  sex  count  istrain  istest
Aaden    M   Aaden    M      51    True  False
Aahana   F   Aahana   F      26    True  False
Aahil    M   Aahil    M       5    True  False
...
Zvi      M     Zvi    M       5    True  False
Zya      F     Zya    F       8    True  False
Zylah   F    Zylah   F       5    True  False
[4025 rows x 5 columns]

```

Now you can transfer the `istest` and `istrain` flags over to the original dataframe, being careful to fill NaNs with False for both the training set and the test set.

```

>>> df['istest'] = df_most_common['istest']
>>> df['istest'] = df['istest'].fillna(False)
>>> df['istrain'] = ~df['istest']
>>> istrain = df['istrain']
>>> df['istrain'].sum() / len(df)
0.9091... #1
>>> df['istest'].sum() / len(df)
0.0908... #2
>>> (df['istrain'].sum() + df['istest'].sum()) / len(df)
1.0

```

Now you can use the training set to fit `TfidfVectorizer` without skewing the n-gram counts with the duplicate names.

```

>>> unique_names = df['name'][istrain].unique()
>>> unique_names = df['name'][istrain].unique()
>>> vectorizer.fit(unique_names)
>>> vecs = vectorizer.transform(df['name'])
>>> vecs
<4238x2855 sparse matrix of type '<class 'numpy.float64'>'>
with 59959 stored elements in Compressed Sparse Row format>

```

You need to be careful when working with sparse data structures. If you convert them normal dense arrays with `.todense()` you may crash your computer by using up all its RAM. But this sparse matrix contains only about 17 million elements so it shows work fine within most laptops. You can use `toarray()` on sparse matrices to create a DataFrame and give meaningful

labels to the rows and columns.

```
>>> vecs = pd.DataFrame(vecs.toarray())
>>> vecs.columns = vectorizer.get_feature_names_out()
>>> vecs.index = df.index
>>> vecs.iloc[:, :7]
          a      aa     aac      aad      aah     aak     aal
Aaden  0.175188  0.392152  0.0  0.537563  0.000000  0.0  0.0
Aahana 0.316862  0.354641  0.0  0.000000  0.462986  0.0  0.0
Aahil  0.162303  0.363309  0.0  0.000000  0.474303  0.0  0.0
...
Zvi      0.000000  0.000000  0.0  0.000000  0.000000  0.0  0.0
Zya      0.101476  0.000000  0.0  0.000000  0.000000  0.0  0.0
Zylah   0.078353  0.000000  0.0  0.000000  0.000000  0.0  0.0
```

Aah, notice that the column labels (character n-grams) all start with lowercase letters. It looks like the `TfidfVectorizer` folded the case (lowercased everything). It's likely that capitalization will help the model, so let's revectorize the names without lowercasing.

```
>>> vectorizer = TfidfVectorizer(analyzer='char',
...      ngram_range=(1, 3), use_idf=False, lowercase=False)
>>> vectorizer = vectorizer.fit(unique_names)
>>> vecs = vectorizer.transform(df['name'])
>>> vecs = pd.DataFrame(vecs.toarray())
>>> vecs.columns = vectorizer.get_feature_names_out()
>>> vecs.index = df.index
>>> vecs.iloc[:, :5]
          A      Aa     Aad      Aah     Aal
name_ sex_
Aaden M      0.193989  0.393903  0.505031  0.000000  0.0
Aahana F      0.183496  0.372597  0.000000  0.454943  0.0
Aahil M      0.186079  0.377841  0.000000  0.461346  0.0
...
Zvi      M      0.000000  0.000000  0.000000  0.000000  0.0
Zya      F      0.000000  0.000000  0.000000  0.000000  0.0
Zylah   F      0.000000  0.000000  0.000000  0.000000  0.0
```

That's better. These character 1, 2, and 3-grams should have enough information to help a neural network guess the sex for names in this birth certificate database.

Choosing a neural network framework

Logistic regressions are the perfect machine learning model for any high dimensional feature vector such as a TF-IDF vector. To turn a logistic regression into a neuron you just need a way to connect it to other neurons. You need a neuron that can learn to predict the outputs of other neurons. And you need to spread the learning out so one neuron doesn't try to do all the work. Each time your neural network gets an example from your dataset that shows it the right answer it will be able to calculate just how wrong it was, the loss or error. But if you have more than one neuron working together to contribute to that prediction, they'll each need to know how much to change their weights to move the output closer to the correct answer. And to know that you need to know how much each weight affects the output, the gradient (slope) of the weights relative to the error. This process of computing gradients (slopes) and telling all the neurons how much to adjust their weights up and down so that the loss will go down is called *backpropagation* or backprop.

A deep learning package like PyTorch can handle all that for you automatically. In fact it can handle any computational graph (network) you can dream up. PyTorch can handle any network of connections between mathematical operations. This flexibility is why most researchers use it rather than TensorFlow (Keras) for their breakthrough NLP algorithms. TensorFlow is designed with a particular kind of computational graph in mind, one that can be efficiently computed on specialized chips manufactured by one of the BigTech companies. Deep Learning is a powerful money-maker for Big Tech and they want to train your brain to use only their tools for building neural networks. I had no idea BigTech would assimilate Keras into the TensorFlow "Borg", otherwise I would not have recommended it in the first edition.

The decline in portability for Keras and the rapidly growing popularity of PyTorch are the main reasons we decided a second edition of this book was in order. What's so great about PyTorch?

Wikipedia has an unbiased and detailed comparison of all DeepLearning frameworks. And Pandas let's you load it directly from the web into a DataFrame:

```
>>> import pandas as pd  
>>> import re
```

```
>>> dfs = pd.read_html('https://en.wikipedia.org/wiki/'  
... + 'Comparison_of_deep-learning_software')  
>>> tabl = dfs[0]
```

Here is how you can use some basic NLP to score the top 10 deep learning frameworks from the Wikipedia article that lists each of their pros and cons. You will find this kind of code is useful whenever you want to turn semi structured natural language into data for your NLP pipelines.

```
>>> bincols = list(tabl.loc[:, 'OpenMP support'].columns)  
>>> bincols += ['Open source', 'Platform', 'Interface']  
>>> dfd = {}  
>>> for i, row in tabl.iterrows():  
...     rowd = row.fillna('No').to_dict()  
...     for c in bincols:  
...         text = str(rowd[c]).strip().lower()  
...         tokens = re.split(r'\W+', text)  
...         tokens += '\*'  
...         rowd[c] = 0  
...         for kw, score in zip(  
...             'yes via roadmap no linux android python \*'.split(),  
...             [1, .9, .2, 0, 2, 2, 2, .1]):  
...             if kw in tokens:  
...                 rowd[c] = score  
...                 break  
...     dfd[i] = rowd
```

Now that the Wikipedia table is cleaned up, you can compute some sort of "total score" for each deep learning framework.

```
>>> tabl = pd.DataFrame(dfd).T  
>>> scores = tabl[bincols].T.sum()      #1  
>>> tabl['Portability'] = scores  
>>> tabl = tabl.sort_values('Portability', ascending=False)  
>>> tabl = tabl.reset_index()  
>>> tabl[['Software', 'Portability']][:10]  
          Software Portability  
0        PyTorch      14.9  
1    Apache MXNet      14.2  
2    TensorFlow      13.2  
3  DeepLearning4j      13.1  
4        Keras       12.2  
5        Caffe       11.2  
6      PlaidML       11.2
```

| | | |
|---|---------------------|------|
| 7 | Apache SINGA | 11.2 |
| 8 | Wolfram Mathematica | 11.1 |
| 9 | Chainer | 11 |

PyTorch got nearly a perfect score because of its support for Linux, Android and all popular deep learning applications.

Another promising one you might want to check out is ONNX. It's really a meta framework and an open standard that allows you to convert back and forth between networks designed on another framework. ONNX also has some optimization and pruning capabilities that will allow your models to run inference much faster on much more limited hardware, such as portable devices.

And just for comparison, how does SciKit Learn stack up to PyTorch for building a neural network model?

Table 5.1. SciKit-Learn vs PyTorch

| SciKit-Learn | PyTorch |
|------------------------------|---|
| for Machine Learning | for Deep Learning |
| Not GPU-friendly | Made for GPUs (parallel processing) |
| <code>model.predict()</code> | <code>model.forward()</code> |
| <code>model.fit()</code> | trained with custom <code>for-loop</code> |
| simple, familiar API | flexible, powerful API |

Enough about frameworks, you are here to learn about neurons. PyTorch is just what you need. And there's a lot left to explore to get familiar with your new PyTorch toolbox.

5.2.5 A sleek sexy PyTorch neuron

Finally it's time to build a neuron using the PyTorch framework. Let's put all this into practice by predicting the sex of the names you cleaned earlier in this chapter.

You can start by using PyTorch to implement a single neuron with logistic activation function - just like the one you used to learn the toy example at the beginning of the chapter.

```
>>> import torch
>>> class LogisticRegressionNN(torch.nn.Module):
...     def __init__(self, num_features, num_outputs=1):
...         super().__init__()
...         self.linear = torch.nn.Linear(num_features, num_outpu
...
...     def forward(self, X):
...         return torch.sigmoid(self.linear(X))

>>> model = LogisticRegressionNN(num_features=vecs.shape[1], num_
>>> model
LogisticRegressionNN(
    (linear): Linear(in_features=3663, out_features=1, bias=True)
)
```

Let's see what happened here. Our model is a *class* that extends the PyTorch class used to define neural networks, `torch.nn.Module`. As every Python class, it has a *constructor* method called `init`. The constructor is where you can define all the attributes of your neural network - most importantly, the model's layers. In our case, we have an extremely simple architecture - one layer with a single neuron, which means there will be only one output. And the number inputs, or features, will be equal to the length of your TF-IDF vector, the dimensionality of your features. There were 3663 unique 1-grams, 2-grams, and 3-grams in our names dataset, so that's how many inputs you'll have for this single-neuron network.

The second crucial method you need to implement for your neural network is the `forward()` method. This method defines how the input to your model propagates through its layers - the *forward propagation*. If you are asking yourself where the backward propagation (backprop) is, you'll soon see, but it's not in the constructor. We decided to use the logistic, or sigmoid, activation function for our neuron - so our `forward()` method will use PyTorch's built-in function `sigmoid`.

Is this all you need to train our model? Not yet. There are two more crucial pieces that your neuron needs to learn. One is the loss function, or cost function that you saw earlier in this chapter. The Mean Square Error you saw is a good candidate for regression problems. In our case, we're trying to do binary classification, and there are cost functions more appropriate for this type of problems - such as Binary Cross Entropy.

Here's what Binary Cross Entropy looks like for a single classification probability p :

Equation 5.2 Binary Cross Entropy

$$\text{BCE} = -(y \log p + (1 - y) \log(1 - p))$$

The logarithmic nature of the function allows it to penalize a "confidently wrong" example, when your model predicts with high probability the sex of a particular name is male, when it is actually more commonly labeled as female. We can help it to make the penalties even more related to reality by using another piece of information available to us - the frequency of the name for a particular sex in our dataset.

```
>>> loss_func_train = torch.nn.BCELoss(  
...      weight=torch.Tensor(df[['count']][istrain].values))  
>>> loss_func_test = torch.nn.BCELoss(    #1  
...      weight=torch.Tensor(df[['count']][~istrain].values))  
>>> loss_func_train  
BCELoss()
```

The last thing we need to choose is how to adjust our weights based on the loss - the optimizer algorithm. Remember our discussion about "skiing" down the gradient of the loss function? The most common way to implement

skiiing downward called Stochastic Gradient Descent (SGD). Instead of taking all of your dataset into account, like your Pythonic perceptron did, it only calculates the gradient based on one sample at a time or perhaps a mini-batch of samples.

Your optimizer needs two parameters to know how fast or how to ski along the loss slope - *learning rate* and *momentum*. The learning rate determines how much your weights change in response to an error - think of it as your "ski velocity". Increasing it can help your model converge to the local minimum faster, but if it's too large, you may overshoot the minimum every time you get close. Any optimizer you would use in PyTorch would have a learning rate.

Momentum is an attribute of our gradient descent algorithm that allows it to "accelerate" when it's moving in the right direction and "slow down" if it's getting away from its target. How do we decide which values to give these two attributes? As with other hyperparameters you see in this book, you'll need to optimize your them to see what's the most effective one for your problem. For now, you can chose some arbitrary values for the hyperparameters `momentum` and `lr` (learning rate).

```
>>> from torch.optim import SGD
>>> hyperparams = {'momentum': 0.001, 'lr': 0.02}      #1
>>> optimizer = SGD(
...     model.parameters(), **hyperparams)      #2
>>> optimizer
SGD (
Parameter Group 0
    dampening: 0
    differentiable: False
    foreach: None
    lr: 0.02
    maximize: False
    momentum: 0.001
    nesterov: False
    weight_decay: 0
)
```

The last step before running our model training is to get the testing and training datasets into a format that PyTorch models can digest.

```
>>> X = vecs.values
>>> y = (df[['sex']] == 'F').values
>>> X_train = torch.Tensor(X[istrain])
>>> X_test = torch.Tensor(X[~istrain])
>>> y_train = torch.Tensor(y[istrain])
>>> y_test = torch.Tensor(y[~istrain])
```

Finally, you're ready for the most important part of this chapter - the sex learning! Let's look at it and understand what happens at each step.

```
>>> from tqdm import tqdm
>>> num_epochs = 200
>>> pbar_epochs = tqdm(range(num_epochs), desc='Epoch:', total=num_epochs)

>>> for epoch in pbar_epochs:
...     optimizer.zero_grad()      #1
...     outputs = model(X_train)
...     loss_train = loss_func_train(outputs, y_train)    #2
...     loss_train.backward()       #3
...     optimizer.step()          #4
Epoch:: 100%|██████████| 200/200 [00:02<00:00, 96.]
```

That was fast! It should take only a couple seconds to train this single neuron for about 200 epochs and thousands of examples for each epoch.

Looks easy, right? We made it as simple as possible so that you can see the steps clearly. But we don't even know how our model is performing! Let's add some utility functions that will help us see if our neuron improves over time. This is called instrumentation. We can of course look at the loss, but it's also good to gage how our model is doing with a more intuitive score, such as accuracy.

First, you'll need a function to convert the PyTorch tensors we get from the module back into numpy arrays:

```
>>> def make_array(x):
...     if hasattr(x, 'detach'):
...         return torch.squeeze(x).detach().numpy()
...     return x
```

Now you use this utility function to measure the accuracy of each iteration on the tensors for your outputs (predictions):

```

>>> def measure_binary_accuracy(y_pred, y):
...     y_pred = make_array(y_pred).round()
...     y = make_array(y).round()
...     num_correct = (y_pred == y).sum()
...     return num_correct / len(y)

```

Now you can rerun your training using these utility function to see the progress of the model's loss and accuracy with each epoch:

```

for epoch in range(num_epochs):
    optimizer.zero_grad()
    outputs = model(X_train)
    loss_train = loss_func_train(outputs, y_train)
    loss_train.backward()
    epoch_loss_train = loss_train.item()
    optimizer.step()
    outputs_test = model(X_test)
    loss_test = loss_func_test(outputs_test, y_test).item()
    accuracy_test = measure_binary_accuracy(outputs_test, y_test)
    if epoch % 20 == 19:
        print(f'Epoch {epoch}:\n'
              f' loss_train/test: {loss_train.item():.4f}/{loss_test:.4f}\n'
              f' accuracy_test: {accuracy_test:.4f}')

```

Epoch 19: loss_train/test: 80.1816/75.3989, accuracy_test: 0.4275
Epoch 39: loss_train/test: 75.0748/74.4430, accuracy_test: 0.5933
Epoch 59: loss_train/test: 71.0529/73.7784, accuracy_test: 0.6503
Epoch 79: loss_train/test: 67.7637/73.2873, accuracy_test: 0.6839
Epoch 99: loss_train/test: 64.9957/72.9028, accuracy_test: 0.6891
Epoch 119: loss_train/test: 62.6145/72.5862, accuracy_test: 0.699
Epoch 139: loss_train/test: 60.5302/72.3139, accuracy_test: 0.707
Epoch 159: loss_train/test: 58.6803/72.0716, accuracy_test: 0.707
Epoch 179: loss_train/test: 57.0198/71.8502, accuracy_test: 0.720
Epoch 199: loss_train/test: 55.5152/71.6437, accuracy_test: 0.728

With just a single set of weights for a single neuron, your simple model was able to achieve more than 70% accuracy on our messy, ambiguous, real-world dataset. Now that's add some more examples from the real world of Tangible AI and some of our contributors.

```

>>> X = vectorizer.transform(
...     ['John', 'Greg', 'Vishvesh',      #1
...      'Ruby', 'Carlana', 'Sarah'])    #2
>>> model(torch.Tensor(X.todense()))
tensor([[0.0196],
       [0.1808],

```

```
[0.3729],  
[0.4964],  
[0.8062],  
[0.8199]], grad_fn=<SigmoidBackward0>)
```

Earlier we chose to use the value 1 to represent "female" and 0 to represent "male." The first three example names, "John," "Greg," and "Vishvesh," are the names of men that have generously contributed to open source projects that are important to me, including the code in this book. It looks like Vish's name doesn't appear on as many US birth certificates for male babies as John or Greg's. The model is more certain of the maleness in the character n-grams for "John" than those for "Vishvesh."

The next three names, "Sarah," "Carlana," and 'Ruby', are the first names of women at the top of my mind when writing this book.^[186] ^[187] The name "Ruby" may have some maleness in its character n-grams because a similar name "Rudy" (often used for male babies) is only 1 edit away from "Ruby." Oddly the name "Carlana," which contains within it a common male name "Carl," is confidently predicted to be a female name.

^[180] McElreath, Richard, and Robert Boyd, *Mathematical Models of Social Evolution: A guide for the perplexed*, University of Chicago Press, 2008.

^[181] USCDI (US Core Data Interoperability) ISA (Interoperability Standards Advisory) article on "Sex (Assigned at Birth)"
(<https://www.healthit.gov/isa/uscdi-data/sex-assigned-birth>)

^[182] from "When I am pinned and wriggling on the wall" in "The Love Song of J. Alfred Prufrock" by T. S. Eliot
(<https://www.poetryfoundation.org/poetrymagazine/poems/44212/the-love-song-of-j-alfred-prufrock>)

^[183] Overview of Coreference Resolution at The Stanford Natural Language Processing Group: (<https://nlp.stanford.edu/projects/coref.shtml>)

^[184] The Peripheral by William Gibson on wikipedia
(https://en.wikipedia.org/wiki/The_Peripheral)

[185] False dichotomy article on wikipedia
(https://en.wikipedia.org/wiki/False_dilemma)

[186] Sarah Goode Wikipedia article
(https://en.wikipedia.org/wiki/Sarah_E._Goode)

[187] Ruby Bridges Wikipedia article
(https://en.wikipedia.org/wiki/Ruby_Bridges)

5.3 Skiing down the error surface

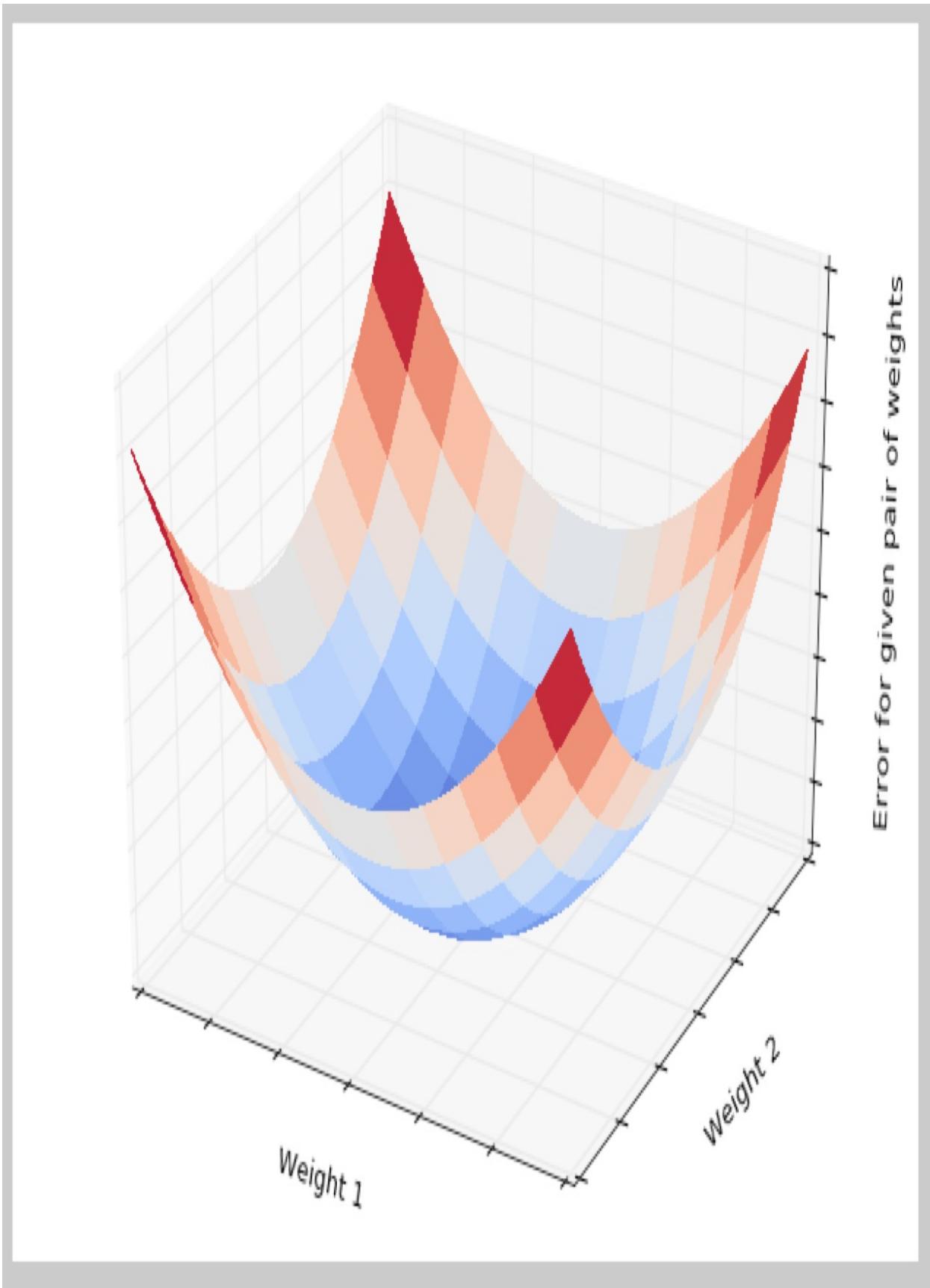
The goal of training in neural networks is to minimize a loss function by finding the best parameters (weights) for your model. Keep in mind, this is not the error for any one particular example from your data set. You want to minimize the cost for all the various errors taken together.

Creating a visualization of this side of the problem can help build a mental model of what you're doing when you adjust the weights of the network as you go.

From earlier, mean squared error is a common cost function (shown back in the "Mean squared error cost function" equation). If you imagine plotting the error as a function of the possible weights, given a specific input and a specific expected output, a point exists where that function is closest to zero; that is your *minimum*—the spot where your 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 three-dimensional bowl with two of the axes being a two-dimensional weight vector and the third being the error (see figure 5.8). That description is a vast simplification, but the concept is the same in higher dimensional spaces (for cases with more than two weights).

Figure 5.6. Convex error curve



Similarly, you can graph the error surface as a function of all possible weights across all the inputs of a training set. But you need to tweak the error function a little. You need something that represents the aggregate error across all inputs for a given set of weights. For this example, you'll use *mean squared error* as the z axis (see equation 5.5).

Here again, you'll get an error surface with a minimum that is located at the set of weights. That set of weights will represent a model that best fits the entire training set.

5.3.1 Off the chair lift, onto the slope - gradient descent and local minima

What does this visualization represent? At each epoch, the algorithm is performing *gradient descent* in trying to minimize the error. Each time you adjust the weights in a direction that will hopefully reduce your error the next time. A convex error surface will be great. Stand on the ski slope, look around, find out which way is down, and go that way!

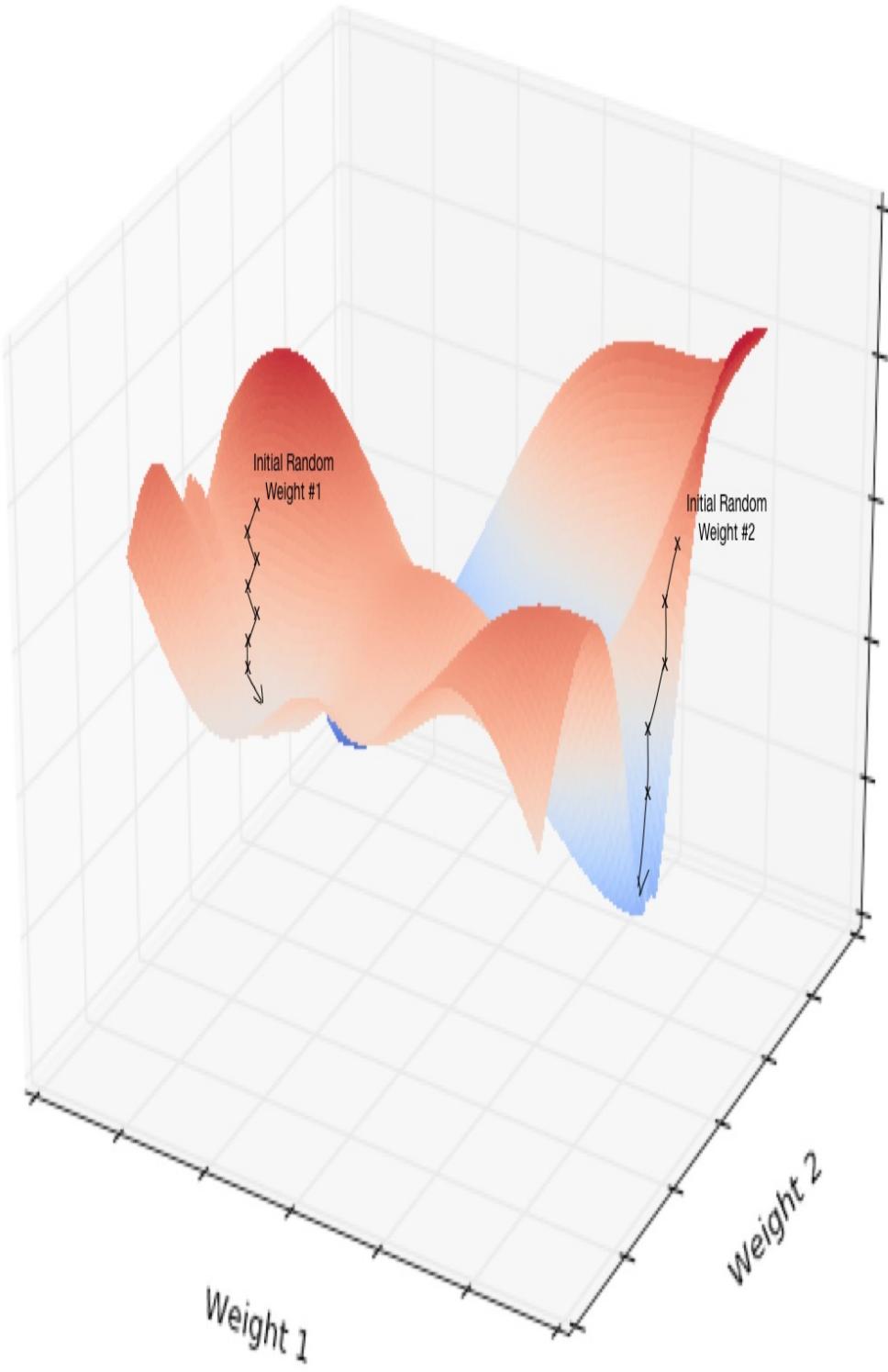
But you're not always so lucky as to have such a smooth shaped bowl; it may have some pits and divots scattered about. This situation is what is known as a *nonconvex error curve*. And, as in skiing, if these pits are big enough, they can suck you in and you might not reach the bottom of the slope.

Again the diagrams are representing weights for two-dimensional input. But the concept is the same if you have a 10-dimensional input, or 50, or 1000. In those higher dimensional spaces, visualizing it doesn't make sense anymore, so you trust the math. Once you start using neural networks, 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 zero. That will tell you 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 nonconvex error space? Aren't those divots and pits a problem? Yes, yes they are. Depending on where you randomly start your

weights, you 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* (see figure 5.9).

Figure 5.7. Nonconvex error curve



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

5.3.2 Shaking things up: stochastic gradient descent

Up until now, you have been aggregating the error for all the training examples and skiing down the steepest route as fast you can. But training on the entire training set one sample at a time is a little nearsighted. It's like choosing the downhill sections of a snow park and ignoring all the jumps. Sometimes a good ski jump can help you skip over some rough terrain.

And if you try to train on the entire dataset at once, you may run out of RAM, bogging down your training in SWAP—swapping data back and forth between RAM and your much slower persistent disk storage. And this single static error surface can have traps. Because you are starting from a random starting point (the initial model weights) you could blindly ski down hill into some local minima (divot, hole, or cave). Your may not know that better options exist for your weight values. And your error surface is static. Once you reach a local minimum in the error surface, there is no downhill slope to help your model ski out and on down the mountain.

So to shake things up you want to add some randomization to the process. You want to periodically shuffle the order of the training examples that your model is learning from. Typically you reshuffle the order of the training examples after each pass through your training dataset. Shuffling your data changes the order in which your model considers the prediction error for each sample. So it will change the path it follows in search of the global minimum (smallest model error for that dataset). This shuffling is the "stochastic" part of stochastic gradient descent.

There's still some room for improving the "gradient" estimation part of gradient descent. You can add a little humility to your optimizer so it doesn't get overconfident and blindly follow every new guess all the way to where it thinks the global minimum should be. It's pretty rare that ski slope where you are is going to point in a straight line directly to the ski lodge at the bottom of the mountain. So your model goes a short distance in the direction of the

downward slope (gradient) without going all the way. This way the gradient for each individual sample doesn't lead your model too far astray and your model doesn't get lost in the woods. You can adjust the *learning rate* hyperparameter of the SGD optimizer (stochastic gradient descent) to control how confident your model is in each individual sample gradient.

Another training approach is *batch learning*. A batch is a subset of the training data, like maybe 0.1%, 1%, 10% or 20% of your dataset. Each batch creates a new error surfaces to experiment with as you ski around searching for the unknown "global" error surface minimum. Your training data is just a sample of the examples that will occur in the real world. So your model shouldn't assume that the "global" real world error surface is shaped the same as the error surface for any portion of your training data.

And this leads to the best strategy for most NLP problems: *mini-batch learning*.^[188] Geoffrey Hinton found that a batch size of around 16 to 64 samples was optimal for most neural network training problems.^[189] This is the right size to balance the shakiness of stochastic gradient descent, with your desire to make significant progress in the correct direction towards the global minimum. And as you move toward the changing local minima on this fluctuating surface, with the right data and right hyperparameters, you can more easily bumble toward the global minimum. Mini-batch learning is a happy medium between *full batch* learning and individual example training. Mini-batch learning it gives you the benefits of both *stochastic* learning (wandering randomly) and *gradient descent* learning (speeding headlong directly down the presumed slope).

Although the details of how *backpropagation* works are fascinating^[190], they aren't trivial, and we won't explain the details here. A good mental image that can help you train your models is to imagine the error surface for your problem as the uncharted terrain of some alien planet. Your optimizer can only look at the slope of the ground at your feet. It uses that information to take a few steps downhill, before checking the slope (gradient) again. It may take a long time to explore the planet this way. But a good optimization algorithm is helping your neural network remember all the good locations on the map and use them to guess a new place on the map to explore in search of the global minimum. On Earth this lowest point on the planet's surface is the

bottom of the canyon under Denman Glacier in Antarctica—3.5 km below sea level.[\[191\]](#) A good mini-batch learning strategy will help you find the steepest way down the ski slope or glacier (not a pleasant image if you’re scared of heights) to the global minimum. Hopefully you’ll soon find yourself by the fire in the ski lodge at the bottom of the mountain or a camp fire in an ice cave below Denman Glacier.

5.3.3 PyTorch: Neural networks in Python

Artificial neural networks require thousands, millions, or even billions of neurons. And you’ll need weights (parameters) that make connections between all those neurons. Each connection or edge in the network multiplies the input by a weight to determine how much the signal is amplified or suppressed. And each node or neuron in the network then sums up all those input signals and usually computes some nonlinear function on that output. That’s a lot of multiplication and addition. And a lot of functions we’d have to write in python.

PyTorch provides a framework for building up these networks in layers. This allows you to specify what various edges and nodes in your network are supposed to do in layers rather than one-by-one on individual neurons. Early on AI researchers appreciated the dangers of proprietary AI algorithms and software. Since PyTorch has always been open source and its sponsors have given free reign to the community of contributors that maintain it. So you can use PyTorch to reproduce all the state of the art research by all the brightest minds in deep learning and AI.



Note

Keras was gradually coopted and appropriated by Big Tech to create lock-in for their products and services. So all the most aware and up-to-date researchers have migrated to PyTorch. Ports (translations) of most Keras and Tensorflow models to PyTorch are created by open source developers almost as quickly as Big Tech can churn out their proprietary models. And the community versions are often faster, more accurate, more transparent, and more efficient. This is another advantage of open source. Open source

contributors don't usually have access to Big Tech compute resources and data sets, so they are forced to make their models more efficient.

PyTorch is a powerful framework to help you create complex computational graphs that can simulate small brains. Let's learn how to use it starting from the smallest part of a brain - the lowly neuron.

See if you can add additional layers to the perceptron you created in this chapter. See if the results you get improve as you increase the network complexity. Bigger is not always better, especially for small problems.

[188] "Faster SGD training by minibatch persistency", by Fischetti et al (<https://arxiv.org/pdf/1806.07353.pdf>)

[189] Neural Networks for Machine Learning - Overview of mini-batch gradient descent by Geoffrey Hinton
(<https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf>)

[190] Wikipedia, <https://en.wikipedia.org/wiki/Backpropagation>

[191] Wikipedia list of places below sea level
(https://en.wikipedia.org/wiki/List_of_places_on_land_with_elevations_below_sea_level)

5.4 Review

1. What is the simple AI logic "problem" that Rosenblatt's artificial neurons couldn't?
2. What minor change to Rosenblatt's architecture "fixed" perceptrons and ended the first "AI Winter"?
3. What is the equivalent of a PyTorch `model.forward()` function in SciKit-Learn models?
4. What test set accuracy can you achieve with the sex-predicting `LogisticRegression` model if you aggregate names across year and region? Don't forget to stratify your test set to avoid cheating.

5.5 Summary

- Minimizing a cost function is a path toward learning.
- A backpropagation algorithm is the means by which a networks *learns*.
- The amount a weight contributes to a model's error is directly related to the amount it needs to updated.
- Neural networks are at their heart optimization engines.
- Watch out for pitfalls (local minima) during training by monitoring the gradual reduction in error.

6 Reasoning with word embeddings (word vectors)

This chapter covers

- Understanding word embeddings or word vectors
- Representing meaning with a vector
- Customizing word embeddings to create domain-specific nessvectors
- Reasoning with word embeddings
- Visualizing the meaning of words

Word embeddings are perhaps the most approachable and generally useful tools in your NLP toolbox. They can give your NLP pipeline a general understanding of words. In this chapter you will learn how to apply word embeddings to real world applications. And just as importantly you'll learn where not to use word embeddings. And hopefully these examples will help you dream up new and interesting applications in business as well as in your personal life.

You can think of word vectors as sorta like lists of attributes for Dota 2 heroes or roll playing game (RPG) characters and monsters. Now imagine that there was no text on these character sheets or profiles. You would want to keep all the numbers in a consistent order so you knew what each number meant. That's how word vectors work. The numbers aren't labeled with their meaning. They are just put in a consistent *slot* or location in the vector. That way when you add or subtract or multiply two word vectors together the attribute for "strength" in one vector lines up with the strength attribute in another vector. Likewise for "agility" and "intelligence" and alignment or philosophy attributes in D&D (Dungeons and Dragons).

Thoughtful roll playing games often encourage deeper thinking about philosophy and words with subtle combinations of character personalities such "chaotic good" or "lawful evil." I'm eternally grateful to my childhood Dungeon Master for opening my eyes to the false dichotomies suggested by

words like "good" and "evil" or "lawful" and "chaotic".^[192] The word vectors you'll learn about here have room for every possible attribute of all the words in almost any text and any language. And the word vector attributes or features are intertwined with each other in complex ways that can handle concepts like "lawful evil", "benevolent dictator" and "altruistic spite" with ease.

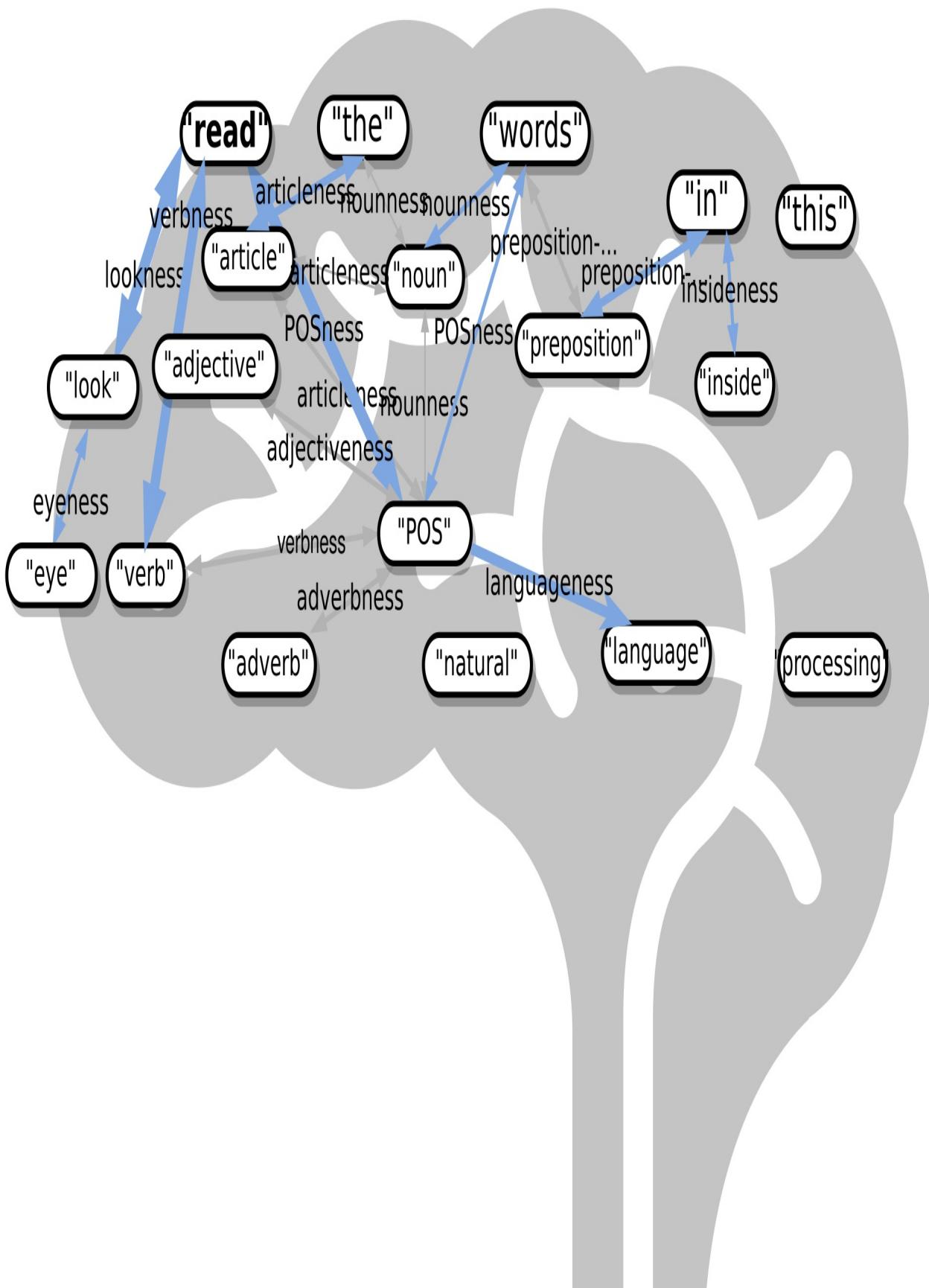
Learning word embeddings are often categorized as a *representation learning* algorithm.^[193] The goal of any word embedding is to build a compact numerical representation of a word's "character". These numerical representations enable a machine to process your words (or your Dota 2 character) are what a machine needs to process words in a meaningful way.

6.1 This is your brain on words

Word embeddings are vectors we use to represent meaning. And your brain is where meaning is stored. Your brain is "on" words—it is affected by them. Just as chemicals affect a brain, so do words. "This is your brain on drugs" was a popular slogan of the 80's anti-narcotics Television advertising campaign that featured a pair of eggs sizzling in a frying pan.^[194]

Fortunately words are much more gentle and helpful influencers than chemicals. The image of your brain on words shown in figure 6.1 looks a little different than eggs sizzling in a frying pan. The sketch gives you one way to imagine the neurons sparking and creating thoughts inside your brain as you read one of these sentences. Your brain connects the meaning of these words together by firing signals to the appropriate neighbor neurons for associated words. Word embeddings are vector representations of these connections between words. And so they are also a crude representation of the node embeddings for the network of neuron connections in your brain.^[195]

Figure 6.1. Word embeddings in your brain



You can think of a word embedding as a vector representation of the pattern of neurons firing in your brain when you think about an individual word. Whenever you think of a word, the thought creates a wave of electrical charges and chemical reactions in your brain originating at the neurons associated with that word or thought. Neurons within your brain fire in waves, like the circular ripples emanating out from a pebble dropped in a pond. But these electrical signals are selectively flowing out through some neurons and not others.

As you read the words in this sentence you are sparking flashes of activity in your neurons like those in the sketch in figure [6.1](#). In fact, researchers have found surprising similarity in the patterns of artificial neural network weights for word embeddings, and the patterns of activity within your brain as you think about words. [\[196\]](#) [\[197\]](#)

Electrons flowing out from neurons are like children running out of a school doorway when the school bell rings for recess. The word or thought is like the school bell. Of course your thoughts and the electrons in your brain are much faster than students. You don't even have to speak or hear the word to trigger its pattern in your brain. You just have to think it. And like kids running out to the playground, the electrons never flow along the same paths twice. Just as the meaning of a word evolves over time, your embedding of a word is constantly evolving. Your brain is a never ending language learner not too different from Cornell's Never Ending Language Learner system. [\[198\]](#)

Some people have gotten carried away with this idea, and they imagine that you can accomplish a form of mind control with words. When I was looking for information about NLP research on Reddit I got distracted by the r/NLP subreddit. It's not what you think. It turns out that some motivational speakers have name-squatted the word "NLP" on reddit for their 70's era "Neuro-linguistic Programming" money-making schemes. [\[199\]](#) [\[200\]](#) [\[201\]](#) Fortunately word embeddings can handle this ambiguity and misinformation just fine.

You don't even have to tell the word embedding algorithm what you want the word "NLP" to mean. It will figure out the most useful and popular meaning

of the acronym based on how it is used within the text that you use to train it. The algorithm for creating word embeddings is a self-supervised machine learning algorithm. This means you will not need a dictionary or thesaurus to feed your algorithm. You just need a lot of text. Later in this chapter you will just gather up a bunch of Wikipedia articles to use as your training set. But any text in any language will do, as long as contains a lot of words that you are interested in.

There's another "brain on words" to think about. Words not only affect the way you think but they affect how you communicate. And you are sorta like a neuron in the collective consciousness, the brain of society. That "sorta" word is an especially powerful pattern of neural connections for me, because I learned what it means from Daniel Dennet's *Intuition Pumps* book.^[202] It invokes associations with complex ideas and words such as the concept of "gradualism" used by Turing to explain how the mechanisms behind both AI and a calculator are exactly the same. Darwin used this concept of gradualism to explain how language-comprehending human brains can evolve from single cell organisms through simple mechanisms.

^[192] Thank you Marc for your chaotic good influence on me!

^[193] Representation learning methodology on Papers With Code
(<https://paperswithcode.com/area/methodology/representation-learning>)

^[194] "This is your brain on drugs"
(https://en.wikipedia.org/wiki/This_Is_Your_Brain_on_Drugs)

^[195] See "Recap: Node Embeddings" by Ted Kye for San Diego Machine Learning Book Club
(<https://github.com/SanDiegoMachineLearning/bookclub/blob/master/graph/g05-GNN1.pdf>)

^[196] "Robust Evaluation of Language-Brain Encoding Experiments"
(<https://arxiv.org/abs/1904.02547>) by Lisa Beinborn (<https://beinborn.eu/>)

^[197] footnote:["Linkng human cognitive patterns to NLP models"
(<https://soundcloud.com/nlp-highlights/130-linking-human-cognitive->

[patterns-to-nlp-models-with-lisa-beinborn](#)) interview of Lisa Beinborn (<https://beinborn.eu/>)

[198] "Never-Ending Language Learning" by T. Mitchell et al at Cornell (http://proai.org/NELL_aaai15.pdf)

[199] "Neuro-linguistic programming" explanation on Wikipedia (https://en.wikipedia.org/wiki/Neuro-linguistic_programming)"

[200] "r/NLP/ Subreddit (<https://www.reddit.com/r/NLP>)"

[201] "An authentic NLP subreddit at "r/NaturalLanguage/" (<https://www.reddit.com/r/NaturalLanguage>)"

[202] *Intuition Pumps and Other Tools for Thinking* by Daniel Dennett p.96

6.2 Applications

Well, what are these awesome word embeddings good for? Word embeddings can be used anywhere you need a machine to understand words or short N-grams. Here are some examples of N-grams where word embeddings have proven useful in the real world:

- Hashtags
- Tags and Keywords
- Named entities (People, Places, Things)
- Titles (Songs, Poems , Books, Articles)
- Job titles & business names
- Web page titles
- Web URLs and file paths
- Wikipedia article titles

Even there are many practical applications where your NLP pipeline could take advantage of the ability to understand these phrases using word embeddings:

- Semantic search for jobs, web pages, ...

- Tip-of-your-tongue word finder
- Rewording a title or sentence
- Sentiment shaping
- Answer word analogy questions
- Reasoning with words and names

And in the academic world researchers use word embeddings to solve some of the 200+ NLP problems: [\[203\]](#)

- Part-of-Speech tagging
- Named Entity Recognition (NER)
- Analogy querying
- Similarity querying
- Transliteration
- Dependency parsing

6.2.1 Search for meaning

In the old days (20 years ago) search engines tried to find all the words you typed based on their TF-IDF scores in web pages. And good search engines attempted would augment your search terms with synonyms. They would sometimes even alter your words to guess what you actually "meant" when you typed a particular combination of words. So if you searched for "sailing cat" they would change cat to catamaran to disambiguate your search for you. Behind the scenes, while ranking your results, search engines might even change a query like "positive sum game" to "nonzero sum game" to send you to the correct Wikipedia page.

Then information retrieval researchers discovered how to make latent semantic analysis more effective—word embeddings. In fact, the GloVe word embedding algorithm is just latent semantic analysis on millions of sentences extracted from web pages. [\[204\]](#) These new word embeddings (vectors) made it possible for search engines to directly match the "meaning" of your query to web pages, without having to guess your intent. The embeddings for your search terms provide a direct numerical representation of the *intent* of your search based on the average meaning of those words on the Internet.



Warning

Word embeddings do not represent *your* intended interpretation of words. They represent the average meaning of those words for everyone that composed the documents and pages that were used to train the word embedding language model. This means that word embeddings contain all the biases and stereotypes of all the people that composed the web pages used to train the model.

Search engines no longer need to do synonym substitution, stemming, lemmatization, case-folding and disambiguation based on hard-coded rules. They create word embeddings based on the text in all the pages in their search index. Unfortunately the dominant search engines decided to use this new-found power to match word embeddings with products and ads rather than real words. Word embeddings for AdWords and iAds are weighted based on how much a marketer has paid to distract you from your intended search. Basically, big tech makes it easy for corporations to bribe the search engine so that it manipulates you and trains you to become their consumption zombie.

If you use a more honest search engine such as Startpage, [\[205\]](#) DISROOT, [\[206\]](#) or Wolfram Alpha [\[207\]](#) you will find they give you what you're actually looking for. And if you have some dark web or private pages and documents you want to use as a knowledge base for your organization or personal life you can self-host a search engine with cutting edge NLP: Elastic Search, [\[208\]](#) Meilisearch, [\[209\]](#) SearX, [\[210\]](#) Apache Solr, [\[211\]](#) Apache Lucene, [\[212\]](#), Qwant, [\[213\]](#) or Sphinx. [\[214\]](#) Even PostgreSQL beats the major search engines for full-text search precision. It will surprise you how much clearer you see the world when you are using an honest-to-goodness search engine.

These semantic search engines use vector search under the hood to query a word and document embedding (vector) database.

Open source Python tools such as NBOOST or PynnDescent let you integrate word embeddings with into your favorite TF-IDF search algorithm. [\[215\]](#) Or if you want a scalable way to search your fine tuned embeddings and vectors

you can use Approximate Nearest Neighbor algorithms to index whatever vectors you like. [\[216\]](#)

That's the nice thing about word embeddings. All that vector algebra math you are used to, such as calculating distance, that will also work for word embeddings. Only now that distance represents how far apart the words are in *meaning* rather than physical distance. And these new embeddings are much more compact and dense with meaning than the thousands of dimensions you are used to with TF-IDF vectors.

You can use the meaning distance to search a database of words for all job titles that are *near* the job title you had in mind for your job search. This may reveal additional job titles you hadn't even thought of. Or your search engine could be designed to add additional words to your search query to make sure related job titles were returned. This would be like an autocomplete search box that understands what words mean - called *semantic search*.

```
>>> from nessvec.indexers import Index #1
>>> index = Index(num_vecs=100_000) #2
>>> index.get_nearest("Engineer").round(2)
Engineer      0.00
engineer     0.23
Engineers    0.27
Engineering  0.30
Architect    0.35
engineers    0.36
Technician   0.36
Programmer   0.39
Consultant   0.39
Scientist    0.39
```

You can see that finding the nearest neighbors of an word embedding is kind of like looking up a word in a Thesaurus. But this is a much fuzzier and complete thesaurus than you'll find at your local book shop or online dictionary. And you will soon see how you can customize this dictionary to work within any domain you like. For example you could train it to work with job postings only from the UK or perhaps even India or Australia, depending on your region of interest. Or you could train it to work better with tech jobs in Silicon Valley rather than finance and banking jobs in New York. You can even train it on 2-grams and 3-grams if you want it to work on

longer job titles like "Software Developer" or "NLP Engineer".

Another nice thing about word embeddings is that they are *fuzzy*. You may have noticed several nearby neighbors of "Engineer" that you'd probably not see in a thesaurus. And you can keep expanding the list as far as you like. So if you were thinking of a Software Engineer rather than an Architect you might want to scan the `get_nearest()` list for another word to do a search for, such as "Programmer":

```
>>> index.get_nearest("Programmer").round(2)
Programmer      -0.00
programmer      0.28
Developer       0.33
Programmers     0.34
Programming     0.37
Engineer        0.39
Software         0.40
Consultant      0.42
programmers     0.42
Analyst         0.42
dtype: float64
>>> index.get_nearest("Developer").round(2)
Developer      -0.00
developer      0.25
Developers     0.25
Programmer     0.33
Software        0.35
developers     0.37
Designer        0.38
Architect       0.39
Publisher       0.39
Development    0.40
```

Well that's surprising. It seems that the title "Developer" is often also associated with the word "Publisher." I would have never guessed why this would be before having worked with the Development Editors, Development Managers, and even a Tech Development Editor at Manning Publishing. Just today these "Developers" cracked the whip to get me moving on writing this Chapter.

6.2.2 Combining word embeddings

Another nice thing about word embeddings is that you can combine them any way you like to create new words! Well, of course, you can combine multiple words the old fashioned way just appending the strings together. In Python you do that with addition or the + operator:

```
>>> "Chief" + "Engineer"  
'ChiefEngineer'  
>>> "Chief" + " " + "Engineer"  
'Chief Engineer'
```

Word embedding math works even better than that. You can add the meanings of the words together to try to find a single word that captures the meaning of the two words you added together

```
>>> chief = (index.data[index.vocab["Chief"]]  
...      + index.data[index.vocab["Engineer"]])  
>>> index.get_nearest(chief)  
Engineer    0.110178  
Chief       0.128640  
Officer     0.310105  
Commander   0.315710  
engineer    0.329355  
Architect   0.350434  
Scientist   0.356390  
Assistant   0.356841  
Deputy      0.363417  
Engineers   0.363686
```

So if you want to one day become a "Chief Engineer" it looks like "Scientist", "Architect", and "Deputy" might also be job titles you'll encounter along the way.

What about that tip-of-your-tongue word finder application mentioned at the beginning of this chapter? Have you ever tried to recall a famous person's name but you only have a general impression of them, like maybe this:

She invented something to do with physics in Europe in the early 20th century.

If you enter that sentence into Google or Bing, you may not get the direct answer you are looking for, "Marie Curie." Google Search will most likely only give you links to lists of famous physicists, both men and women.

You would have to skim several pages to find the answer you are looking for. But once you found "Marie Curie," Google or Bing would keep note of that. They might get better at providing you search results the next time you look for a scientist. (At least, that is what it did for us in researching this book. We had to use private browser windows to ensure that your search results would be similar to ours.)

With word embeddings, you can search for words or names that combine the meaning of the words "woman," "Europe," "physics," "scientist," and "famous," and that would get you close to the token "Marie Curie" that you are looking for. And all you have to do to make that happen is add up the vectors for each of those words that you want to combine:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv['physics'] +  
...      wv['scientist']
```

In this chapter, we show you the exact way to do this query. You can even see how you might be able to use word embedding math to subtract out some of the gender bias within a word:

```
>>> answer_vector = wv['woman'] + wv['Europe'] + wv['physics'] +\n...      wv['scientist'] - wv['male'] - 2 * wv['man']
```

With word embeddings, you can take the "man" out of "woman"!

6.2.3 Analogy questions

What if you could rephrase your question as an analogy question? What if your "query" was something like this:

Who is to nuclear physics what Louis Pasteur is to germs?

Again, Google Search, Bing, and even Duck Duck Go are not much help with this one.^[217] But with word embeddings, the solution is as simple as subtracting "germs" from "Louis Pasteur" and then adding in some "physics":

```
>>> answer_vector = wv['Louis_Pasteur'] - wv['germs'] + wv['physi
```

And if you are interested in trickier analogies about people in unrelated

fields, such as musicians and scientists, you can do that, too.

Who is the Marie Curie of music?

OR

Marie Curie is to science as who is to music?

Can you figure out what the vector space math would be for that question?

You might have seen questions like these on the English analogy section of standardized tests such as SAT, ACT, or GRE exams. Sometimes they are written in formal mathematical notation like this:

MARIE CURIE : SCIENCE :: ? : MUSIC

Does that make it easier to guess the vector math for these words? One possibility is this:

```
>>> wv['Marie_Curie'] - wv['science'] + wv['music']
```

And you can answer questions like this for things other than people and occupations, like perhaps sports teams and cities:

The Timbers are to Portland as what is to Seattle?

In standardized test form, that is:

TIMBERS : PORTLAND :: ? : SEATTLE

But, more commonly, standardized tests use English vocabulary words and ask less fun questions, like the following:

WALK : LEGS :: ? : MOUTH

OR

ANALOGY : WORDS :: ? : NUMBERS

All those "tip of the tongue" questions are a piece of cake for word

embeddings, even though they are not multiple choice. It can be difficult to get analogy questions right, even when you have multiple choice options to choose from. NLP comes to the rescue with word embeddings.

Word embeddings can be used to answer even these vague questions and analogy problems. Word embeddings can help you remember any word or name on the tip of your tongue, as long as the vector for the answer exists in your vocabulary. (For Google's pretrained Word2Vec model, your word is almost certainly within the 100B word news feed that Google trained it on, unless your word after 2013.) And embeddings work well even for questions that you cannot even pose in the form of a search query or analogy.

You can learn about some of the math with embeddings in the "analogical reasoning" section later in this chapter.

6.2.4 Word2Vec Innovation

Words that are used near each other sort of pile up on top of each other in our minds and eventually define what those words mean within the connections of the neurons of our brains. As a toddler you hear people talking about things "soccer balls," "fire trucks," "computers," and "books," and you can gradually figure out what each of them are. The surprising thing is that your machine does not need a body or brain to understand words as well as a toddler.

A child can learn a word after pointing out objects in the real world or a picture book a few times. A child never needs to read a dictionary or thesaurus. Like a child, a machine "figures it out" without a dictionary or thesaurus or any other supervised machine learning dataset. A machine does not even need to see objects or pictures. The machine is completely self-supervised by the way you parse the text and set up the dataset. All you need is a lot of text.

In previous chapters, you could ignore the nearby context of a word. All you needed to do was count up the uses of a word within the same *document*. It turns out, if you make your documents very very short, these counts of co-occurrences becomes useful for representing the meaning of words

themselves. This was the key innovation of Tomas Mikolov and his Word2vec NLP algorithm. John Rubert Firth popularized the concept that "a word is characterized by the company it keeps."^[218] But to make word embeddings useful required Tomas Mikolov's focus on a very small "company" of words and the computational power of 21st century computers as well as massive corpora machine-readable text. You do not need a dictionary or thesaurus to train your word embeddings. You only need a large body of text.

That is what you are going to do in this chapter. You are going to teach a machine to be sponge, like a toddler. You are going to help machines figure out what words mean, without ever explicitly labeling words with their dictionary definitions. All you need is a bunch of random sentences pulled from any random book or web page. Once you tokenize and segment those sentences, which you learned how to do in previous chapters, your NLP pipeline will get smarter and smarter each time it reads a new batch of sentences.

In chapter 2 and 3 you isolated words from their neighbors and only worried about whether they were present or absent in each *document*. You ignored the effect the neighbors of a word have on its meaning and how those relationships affect the overall meaning of a statement. Our bag-of-words concept jumbled all the words from each document together into a statistical bag. In this chapter, you will create much smaller bags of words from a "neighborhood" of only a few words, typically fewer than ten tokens. You will also ensure that these neighborhoods have boundaries to prevent the meaning of words from spill over into adjacent sentences. This process will help focus your word embedding language model on the words that are most closely related to one another.

Word embeddings are able to identify synonyms, antonyms, or words that just belong to the same category, such as people, animals, places, plants, names, or concepts. We could do that before, with semantic analysis in chapter 4, but your tighter limits on a word's neighborhood will be reflected in tighter accuracy on the word embeddings. Latent semantic analysis (LSA) of words, *n*-grams, and documents did not capture all the literal meanings of a word, much less the implied or hidden meanings. Some of the connotations

of a word are fuzzier for LSA's oversized bags of words.



Word embeddings

Word embeddings (sometimes called *word vectors*) are high dimensional numerical vector representations of what a word means, including its literal and implied meaning. So word embeddings can capture the *connotation* of words. Somewhere inside an embedding there is a score for "peopleness," "animalness," "placeness," "thingness" and even "conceptness." And a word embedding combines all those scores, and all the other *ness* of words, into a dense vector (no zeros) of floating point values.

The density and high (but not too high) dimensionality of word embeddings is a source of their power as well as their limitations. This is why dense, high dimensional embeddings are most valuable when you use them in your pipeline along side sparse hyper-dimensional TFIDF vectors or discrete bag-of-words vectors.

[203] Papers With Code topic "Word Embeddings"
(<https://paperswithcode.com/task/word-embeddings>)

[204] Standford's open source GloVe algorithm in C
(<https://github.com/stanfordnlp/GloVe>) and Python (https://github.com/lapis-zero09/compare_word_embedding/blob/master/glove_train.py)

[205] Startpage proviacy-protecting web search (<https://www.startpage.com/>)

[206] DISROOT nonprofit search engine (<https://search.disroot.org>)

[207] Wolfram Alpha uses state-of-the art NLP (<https://wolframalpha.com/>)

[208] ElasticSearch backend source code
(<https://github.com/elastic/elasticsearch>) and frontend SearchKit demo
(<https://demo.searchkit.co/type/all?query=prosocial%20AI>)

[209] Meilisearch source code and self-hosting docker images
(<https://github.com/meilisearch/meilisearch>) and managed hosting

(<https://www.meilisearch.com/>)

[210] SearX git repository (<https://github.com/searx/searx>)

[211] Apache Solr home page and Java source code (<https://solr.apache.org/>)

[212] Apache Lucene home page (<https://lucene.apache.org/>)

[213] Qwant web search engine is based in Europe where regulations protect you from manipulation and deception (<https://www.qwant.com/>)

[214] Sphinx home page and C source code (<http://sphinxsearch.com/>)

[215] "How to Build a Semantic Search Engine in 3 minutes" by Cole Thienes and Jack Pertschuk (<http://mng.bz/yvjG>)

[216] PynnDescent Python package (<https://pypi.org/project/pynndescent/>)

[217] Try them all if you don't believe us.

[218] See wikipedia article (https://en.wikipedia.org/wiki/John_Rupert_Firth)

6.3 Artificial Intelligence Relies on Embeddings

Word embeddings were a big leap forward in not only natural language understanding accuracy but also a breakthrough in the hope for Artificial General Intelligence, or AGI.

Do you think you could tell the difference between intelligent and unintelligent messages from a machine? It may not be as obvious as you think. Even the "deep minds" at BigTech were fooled by the surprisingly unintelligent answers from their latest and greatest chatbots in 2023, Bing and Bard. Simpler, more authentic conversational search tools such as you.com and neeva.com and their chat interfaces outperform BigTech search on most Internet research tasks.

The philosopher Douglas Hofstader pointed out a few things to look out for

when measuring intelligence. footnote[Douglas R. Hofstadter, "Gödel, Escher, Bach: an Eternal Golden Braid (GEB), p. 26]

- flexibility
- dealing with ambiguity
- ignoring irrelevant details
- finding similarities and analogies
- generating new ideas

You'll soon see how word embeddings can enable these aspects of intelligence within your software. For example, word embeddings make it possible to respond with flexibility by giving words fuzziness and nuance that previous representations like TF-IDF vectors could not. In previous iterations of your chatbot you would have to enumerate all the possible ways to say "Hi" if you want your bot to be flexible in its response to common greetings.

But with word embeddings you can recognize the **meaning** of the word "hi", "hello", and "yo" all with a single embedding vector. And you can create embeddings for all the concepts your bot is likely to encounter by just feeding it as much text as you can find. There is no need to hand-craft your vocabularies anymore.



Caution

Like word embeddings, intelligence itself is a high dimensional concept. This makes *Artificial General Intelligence* (AGI) an elusive target. Be careful not to allow your users or your bosses to thinking that your chatbot is generally intelligent, even if it appears to achieve all of Hofstadter's "essential elements."

6.4 Word2Vec

In 2012, Tomas Mikolov, an intern at Microsoft, found a way to embed the meaning of words into vector space. Word embeddings or word vectors typically have 100 to 500 dimensions, depending on the breadth of information in the corpus used to train them. Mikolov trained a neural

network to predict word occurrences near each target word. Mikolov used a network with a single hidden layer, so almost any linear machine learning model will also work. Logistic regression, truncated SVD, linear discriminant analysis, or Naive Bayes would all work well and were used successfully by others to duplicates Mikolov's results. In 2013, once at Google, Mikolov and his teammates released the software for creating these word vectors and called it "Word2Vec."[\[219\]](#)

The Word2Vec language model learns the meaning of words merely by processing a large corpus of unlabeled text. No one has to label the words in the Word2Vec vocabulary. No one has to tell the Word2Vec algorithm that "Marie Curie" is a scientist, that the "Timbers" are a soccer team, that Seattle is a city, or that Portland is a city in both Oregon and Maine. And no one has to tell Word2Vec that soccer is a sport, or that a team is a group of people, or that cities are both "places" as well as "communities." Word2Vec can learn that and much more, all on its own! All you need is a corpus large enough to mention "Marie Curie," "Timbers," and "Portland" near other words associated with science, soccer, or cities.

This unsupervised nature of Word2Vec is what makes it so powerful. The world is full of unlabeled, uncategorized, and unstructured natural language text.

Unsupervised learning and *supervised* learning are two radically different approaches to machine learning.



Supervised learning

In supervised learning, a human or team of humans must label data with the correct value for the target variable. An example of a label is the "spam" categorical label for an SMS message in chapter 4. A more difficult label for a human might be a percentage score for the hotness connotation of the word "red" or "fire". Supervised learning is what most people think of when they think of machine learning. A supervised model can only get better if it can measure the difference between the expected output (the label) and its predictions.

In contrast, unsupervised learning enables a machine to learn directly from data, without any assistance from humans. The training data does not have to be organized, structured, or labeled by a human. So unsupervised learning algorithms like Word2Vec are perfect for natural language text.



Unsupervised learning

In unsupervised learning, you train the model to perform a task, but without any labels, only the raw data. Clustering algorithms such as k-means or DBSCAN are examples of unsupervised learning. Dimension reduction algorithms like principal component analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) are also unsupervised machine learning techniques. In unsupervised learning, the model finds patterns in the relationships between the data points themselves. An unsupervised model can get smarter (more accurate) just by throwing more data at it.

Instead of trying to train a neural network to learn the target word meanings directly (on the basis of labels for that meaning) you teach the network to predict words near the target word in your sentences. So in this sense, you do have labels: the nearby words you are trying to predict. But because the labels are coming from the dataset itself and require no hand-labeling, the Word2Vec training algorithm is definitely an unsupervised learning algorithm.

Another domain where this unsupervised training technique is used is in time series modeling. Time series models are often trained to predict the next value in a sequence based on a window of previous values. Time series problems are remarkably similar to natural language problems in a lot of ways because they deal with ordered sequences of values (words or numbers).

And the prediction itself is not what makes Word2Vec work. The prediction is merely a means to an end. What you do care about is the internal representation, the vector, that Word2Vec gradually builds up to help it generate those predictions. This representation will capture much more of the meaning of the target word (its semantics) than the word-topic vectors that came out of latent semantic analysis (LSA) and latent Dirichlet allocation

(LDiA) in chapter 4.



Note

Models that learn by trying to repredict the input using a lower-dimensional internal representation are called *autoencoders*. This may seem odd to you. It is like asking the machine to echo back what you just asked them, only they cannot write the question down as you are saying it. The machine has to compress your question into shorthand. And it has to use the same shorthand algorithm (function) for all the questions you ask it. The machine learns a new shorthand (vector) representation of your statements.

If you want to learn more about unsupervised deep learning models that create compressed representations of high-dimensional objects like words, search for the term "autoencoder."^[220] They are also a common way to get started with neural nets, because they can be applied to almost any dataset.

Word2Vec will learn about things you might not think to associate with all words. Did you know that every word has some geography, sentiment (positivity), and gender associated with it? If any word in your corpus has some quality, like "placeness", "peopleness", "conceptness" or "femaleness", all the other words will also be given a score for these qualities in your word vectors. The meaning of a word "rubs off" on the neighboring words when Word2Vec learns word vectors.

All words in your corpus will be represented by numerical vectors, similar to the word-topic vectors discussed in chapter 4. Only this time the "topics" mean something more specific, more precise. In LSA, words only had to occur in the same document to have their meaning "rub off" on each other and get incorporated into their word-topic vectors. For Word2Vec word vectors, the words must occur near each other—typically fewer than five words apart and within the same sentence. And Word2Vec word vector "topic" weights can be added and subtracted to create new word vectors that mean something!

A mental model that may help you understand word vectors is to think of word vectors as a list of weights or scores. Each weight or score is associated

with a specific dimension of meaning for that word.

Listing 6.1. Compute nessvector

```
>>> from nlpia.book.examples.ch06_nessvectors import *      #1
>>> nessvector('Marie_Curie').round(2)
placeness      -0.46
peopleness     0.35      #2
animalness     0.17
conceptness    -0.32
femaleness      0.26
```

You can compute "nessvectors" for any word or n -gram in the Word2Vec vocabulary using the tools from nlpia

(https://gitlab.com/tangibleai/nlpia2/blob/master/src/nlpia/book/examples/ch06_nessvectors.py)

And this approach will work for any "ness" components that you can dream up.

Mikolov developed the Word2Vec algorithm while trying to think of ways to numerically represent words in vectors. He wasn't satisfied with the less accurate word sentiment math you did in chapter 4. He wanted to do *analogical reasoning*, like you just did in the previous section with those analogy questions. This concept may sound fancy, but really it just means that you can do math with word vectors and that the answer makes sense when you translate the vectors back into words. You can add and subtract word vectors to *reason* about the words they represent and answer questions similar to your examples above, like the following. (For those not up on sports in the US, the Portland Timbers and Seattle Sounders are Major League Soccer teams.)

```
wv['Timbers'] - wv['Portland'] + wv['Seattle'] = ?
```

Ideally you'd like this math (word vector reasoning) to give you this:

```
wv['Seattle_Sounders']
```

Similarly, your analogy question "'Marie Curie' is to 'physics' as __ is to 'classical music'?" can be thought about as a math expression like this:

```
wv['Marie_Curie'] - wv['physics'] + wv['classical_music'] = ?
```

In this chapter, we want to improve on the LSA word vector representations we introduced in chapter 4. Topic vectors constructed from entire documents using LSA are great for document classification, semantic search, and clustering. But the topic-word vectors that LSA produces aren't accurate enough to be used for semantic reasoning or classification and clustering of short phrases or compound words. You'll soon learn how to train the single-layer neural networks required to produce these more accurate, more fun, word vectors. And you'll see why they have replaced LSA word-topic vectors for many applications involving short documents or statements.

6.4.1 Analogy reasoning

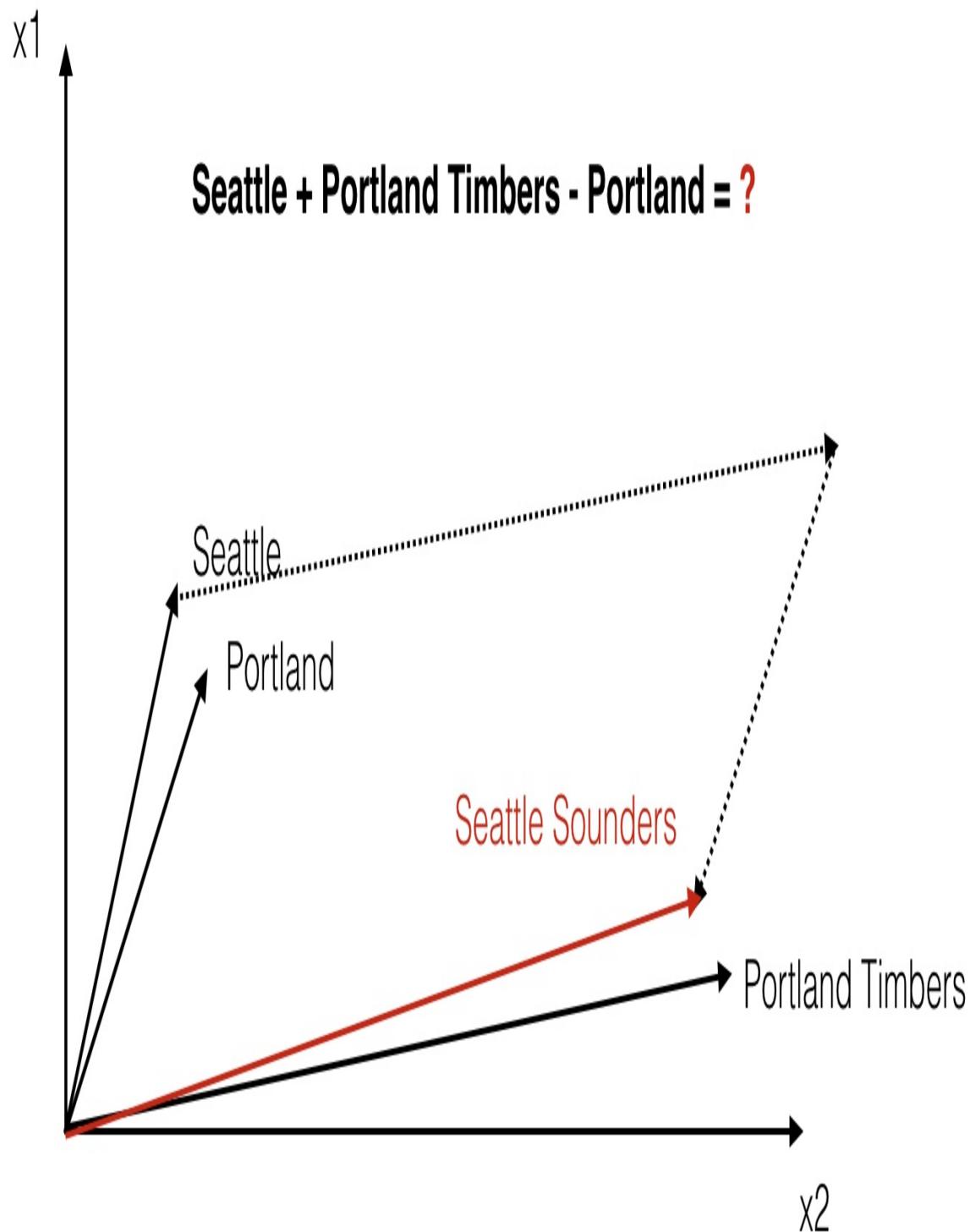
Word2Vec was first presented publicly in 2013 at the ACL conference.[\[221\]](#) The talk with the dry-sounding title "Linguistic Regularities in Continuous Space Word Representations" described a surprisingly accurate language model. Word2Vec embeddings were four times more accurate (45%) compared to equivalent LSA models (11%) at answering analogy questions like those above.[\[222\]](#) The accuracy improvement was so surprising, in fact, that Mikolov's initial paper was rejected by the International Conference on Learning Representations.[\[223\]](#) Reviewers thought that the model's performance was too good to be true. It took nearly a year for Mikolov's team to release the source code and get accepted to the Association for Computational Linguistics.

Suddenly, with word vectors, questions like

Portland Timbers + Seattle - Portland = ?

can be solved with vector algebra (see figure 6.1).

Figure 6.2. Geometry of Word2Vec math



The word2vec language model "knows" that the terms "Portland" and "Portland Timbers" are roughly the same distance apart as "Seattle" and

"Seattle Sounders". And those vector displacements between the words in each pair are in roughly the same direction. So the word2vec model can be used to answer your sports team analogy question. You can add the difference between "Portland" and "Seattle" to the vector that represents the "Portland Timbers". That should get you close to the vector for "Seattle Sounders".

Equation 6.1 Compute the answer to the soccer team question

$$\begin{bmatrix} 0.0168 \\ 0.007 \\ 0.247 \\ \dots \end{bmatrix} + \begin{bmatrix} 0.093 \\ -0.028 \\ -0.214 \\ \dots \end{bmatrix} - \begin{bmatrix} 0.104 \\ 0.0883 \\ -0.318 \\ \dots \end{bmatrix} = \begin{bmatrix} 0.006 \\ -0.109 \\ 0.352 \\ \dots \end{bmatrix}$$

After adding and subtracting word vectors, your resultant vector will almost never exactly equal one of the vectors in your word vector vocabulary. Word2Vec word vectors usually have 100s of dimensions, each with continuous real values. Nonetheless, the vector in your vocabulary that is closest to the resultant will often be the answer to your NLP question. The English word associated with that nearby vector is the natural language answer to your question about sports teams and cities.

Word2Vec allows you to transform your natural language vectors of token occurrence counts and frequencies into the vector space of much lower-dimensional Word2Vec vectors. In this lower-dimensional space, you can do your math and then convert back to a natural language space. You can imagine how useful this capability is to a chatbot, search engine, question answering system, or information extraction algorithm.



Note

The initial paper in 2013 by Mikolov and his colleagues was able to achieve

an answer accuracy of only 40%. But 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 further. This was accomplished by training it on extremely large corpora. The reference implementation was trained on the 100 billion words from the Google News Corpus. This is the pretrained model you'll see used in this book a lot.

The research team also discovered that the difference between a singular and a plural word is often roughly the same magnitude, and in the same direction:

Equation 6.2 Distance between the singular and plural versions of a word

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

But their discovery didn't stop there. They also discovered that the distance relationships go far beyond simple singular versus plural relationships. Distances apply to other semantic relationships. The Word2Vec researchers soon discovered they could answer questions that involve geography, culture, and demographics, like this:

"San Francisco is to California as what is to Colorado?"

San Francisco - California + Colorado = Denver

More reasons to use word vectors

Vector representations of words are useful not only for reasoning and analogy problems, but also for all the other things you use natural language vector space models for. From pattern matching to modeling and visualization, your NLP pipeline's accuracy and usefulness will improve if you know how to use the word vectors from this chapter.

For example, later in this chapter we show you how to visualize word vectors on 2D "semantic maps" like the one shown in figure 6.2. You can think of this like a cartoon map of a popular tourist destination or one of those

impressionistic maps you see on bus stop posters. In these cartoon maps, things that are close to each other semantically as well as geographically get squished together. For cartoon maps, the artist adjusts the scale and position of icons for various locations to match the "feel" of the place. With word vectors, the machine too can have a feel for words and places and how far apart they should be.

So your machine will be able generate impressionistic maps like the one in figure 6.3 using word vectors you are learning about in this chapter. [\[224\]](#)

Figure 6.3. Word vectors for ten US cities projected onto a 2D map



If you're familiar with these US cities, you might realize that this isn't an accurate geographic map, but it's a pretty good semantic map. I, for one, often confuse the two large Texas cities, Houston and Dallas, and they have almost identical word vectors. And the word vectors for the big California cities make a nice triangle of culture in my mind.

And word vectors are great for chatbots and search engines too. For these applications, word vectors can help overcome some of the rigidity, brittleness of pattern, or keyword matching. Say you were searching for information about a famous person from Houston, Texas, but didn't realize they'd moved to Dallas. From figure 6.2, you can see that a semantic search using word vectors could easily figure out a search involving city names such as Denver and Houston. And even though character-based patterns wouldn't understand

the difference between "tell me about a Denver omelette" and "tell me about the Denver Nuggets", a word vector pattern could. Patterns based on word vectors would likely able to differentiate between the food item (omelette) and the basketball team (Nuggets) and respond appropriately to a user asking about either.

6.4.2 Learning word embeddings

Word embeddings are vectors that represent the meaning (semantics) of words. However the meaning of words is an elusive, fuzzy thing to capture. An isolated individual word has very ambiguous meaning. Here are some of the things that can affect the meaning of a word:

- Whose thought is being communicated with the word
- Who is the word intended to be understood by
- The context (where and when) the word is being used
- The domain knowledge or background knowledge assumed
- The sense of the word intended

Your brain will likely understand a word quite differently than mine. And the meaning of a word in your brain changes over time. You learn new things about a word as you make new connections to other concepts. And as you learn new concepts and words, you learn new connections to these new words depending on the impression of the new words on your brain. Embeddings are used to represent this evolving pattern of neuron connections in your brain created by the new word. And these new vectors have 100s of dimensions.

Imagine a young girl who says "My mommy is a doctor."^[225] Imagine what that word "doctor" means to her. And then think about how her understanding of that word, her NLU processing algorithm, evolves as she grows up. Over time she will learn to differentiate between a medical doctor (M.D.) and an academic doctor of philosophy (Ph.D.). Imagine what that word means to her just a few years later when she herself begins to think about the possibility of applying to med school or a Ph.D. program. And imagine what that word means to her father or her mother, the doctor. And imagine what that word means to someone who doesn't have access to healthcare.

Creating useful numerical representations of words is tricky. The meaning you want to encode or embed in the vector depends not only on whose meaning you want to represent, but also on when and where you want your machine to process and understand that meaning. In the case of GloVe, Word2Vec and other early word embeddings the goal was to represent the "average" or most popular meaning. The researchers creating these representations were focused on analogy problems and other benchmark tests that measure human and machine understanding of words. For example we used pretrained fastText word embeddings for the code snippets earlier in this chapter.



Tip

Pretrained word vector representations are available for corpora like Wikipedia, DBpedia, Twitter, and Freebase.^[226] These pretrained models are great starting points for your word vector applications.

- Google provides a pretrained word2vec model based on English Google News articles.^[227]
- Facebook published their word models, called *fastText*, for 294 languages.^[228]

Fortunately, once you've decided your "audience" or "users" for the word embeddings, you only need to gather up example usages of those words. Word2Vec, GloVe, and fastText are all unsupervised learning algorithms. All you need is some raw text from the *domain* that you and your users are interested in. If you are mainly interested in medical doctors you can train your embeddings on a collection of texts from medical journals. Or if you want the most general understanding of words represented in your vectors, ML engineers often use Wikipedia and online news articles to capture the meaning of words. After all, Wikipedia represents our collective understanding of everything in the world.

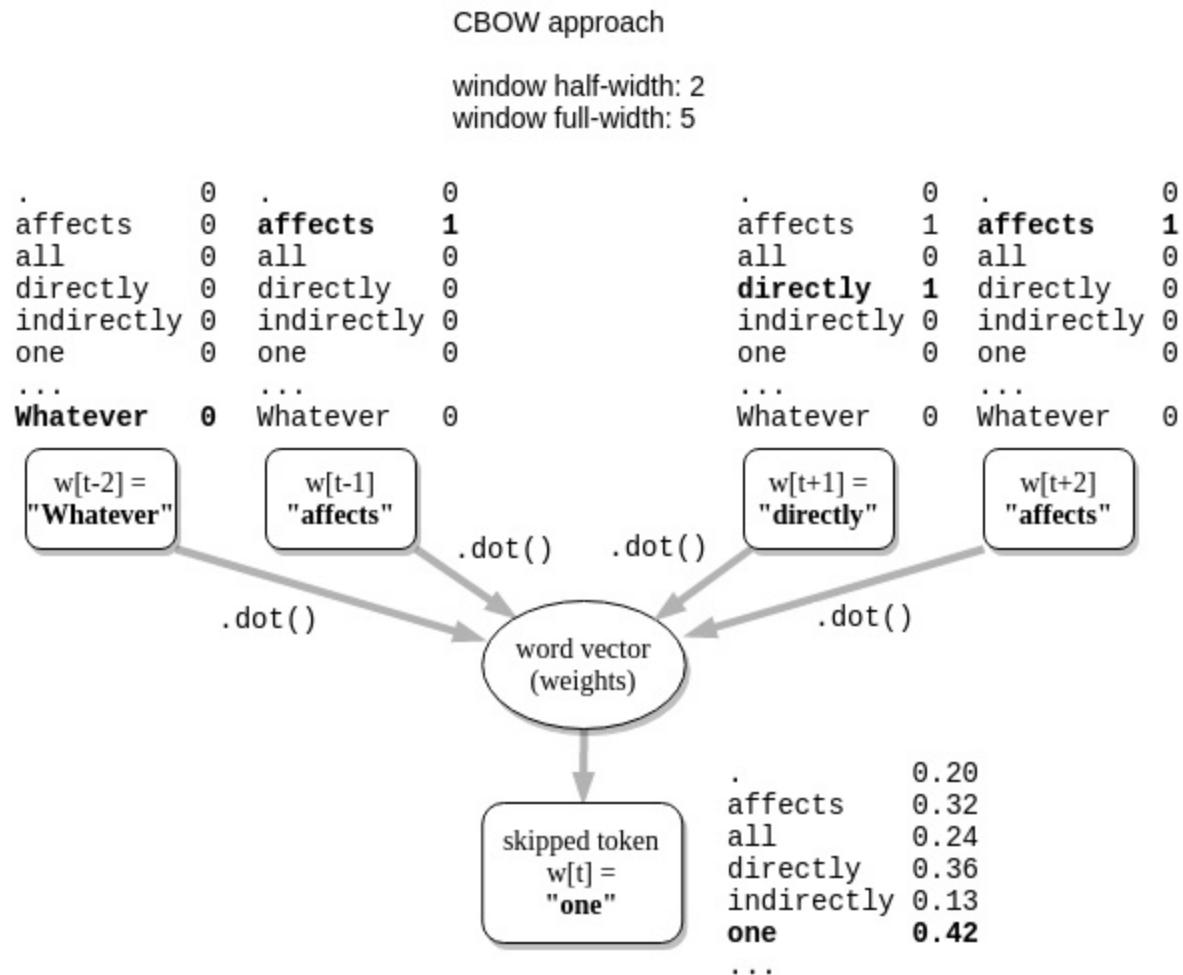
Now that you have your corpus how exactly do you create a training set for your word embedding language model? In the early days there were two main approaches:

1. *Continuous bag-of-words* (CBOW)
2. *Continuous skip-gram*

The *continuous bag-of-words* (CBOW) approach predicts the target word (the output or "target" word) from the nearby context words (input words). The only difference with the bag-of-words (BOW) vectors you learned about in chapter 3 is that a CBOWs are created for a continuously sliding window of words within each document. So you will have almost as many CBOW vectors as you have words in the sequence of words from all of your documents. Whereas for the BOW vectors you only had one vector for each document. This gives your for word embedding training set a lot more information to work with so it will produce more accurate embedding vectors. With the CBOW approach you create a huge number of tiny synthetic documents from every possible phrase you can extract from your original documents.

Figure 6.4. CBOW neural network architecture

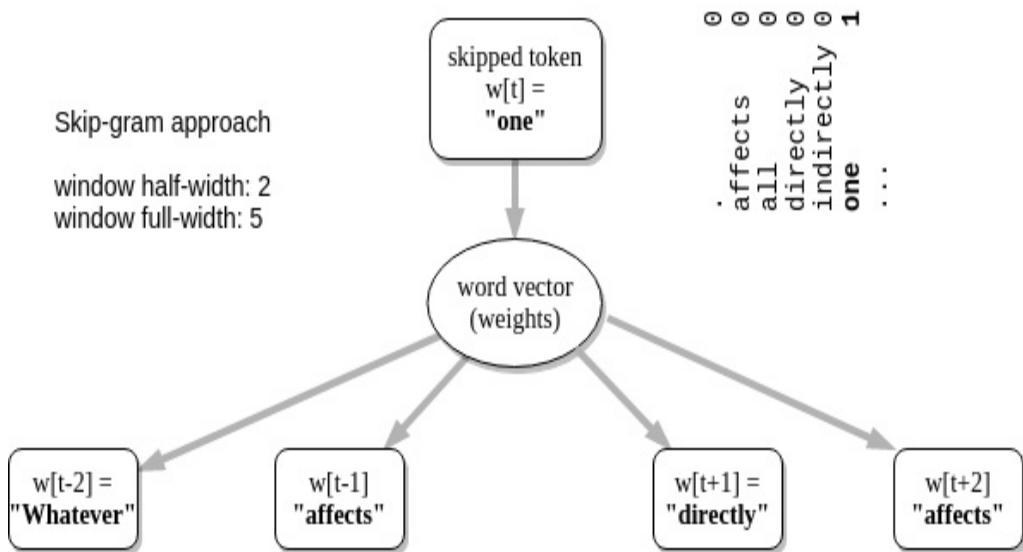
Whatever affects one directly affects all indirectly.



For the *skip-gram* approach you also create this huge number of synthetic documents. You just reverse the prediction target so that you're using the CBOW targets to predict the CBOW features. predicts the context words ("target" words) from a word of interest (the input word). Though these may seem like your pairs of words are reversed, you will see soon that the results are almost mathematically equivalent.

Figure 6.5. Skip-gram neural network architecture

Whatever affects **one** directly affects all indirectly.



You can see how the two neural approaches produce the same number of training examples and create the same number of training examples for both the skip-gram and CBOW approach.

Skip-gram approach

In the skip-gram training approach, you predict a word in the neighborhood of the context word. Imagine your corpus contains this wise rejection of individualism by Bayard Rustin and Larry Dane Brimner. [\[229\]](#)

We are all one. And if we don't know it, we will find out the hard way.

-- Bayard Rustin *We Are One: The Story of Bayard Rustin*, 2007, p.46 by Larry Dane Brimner



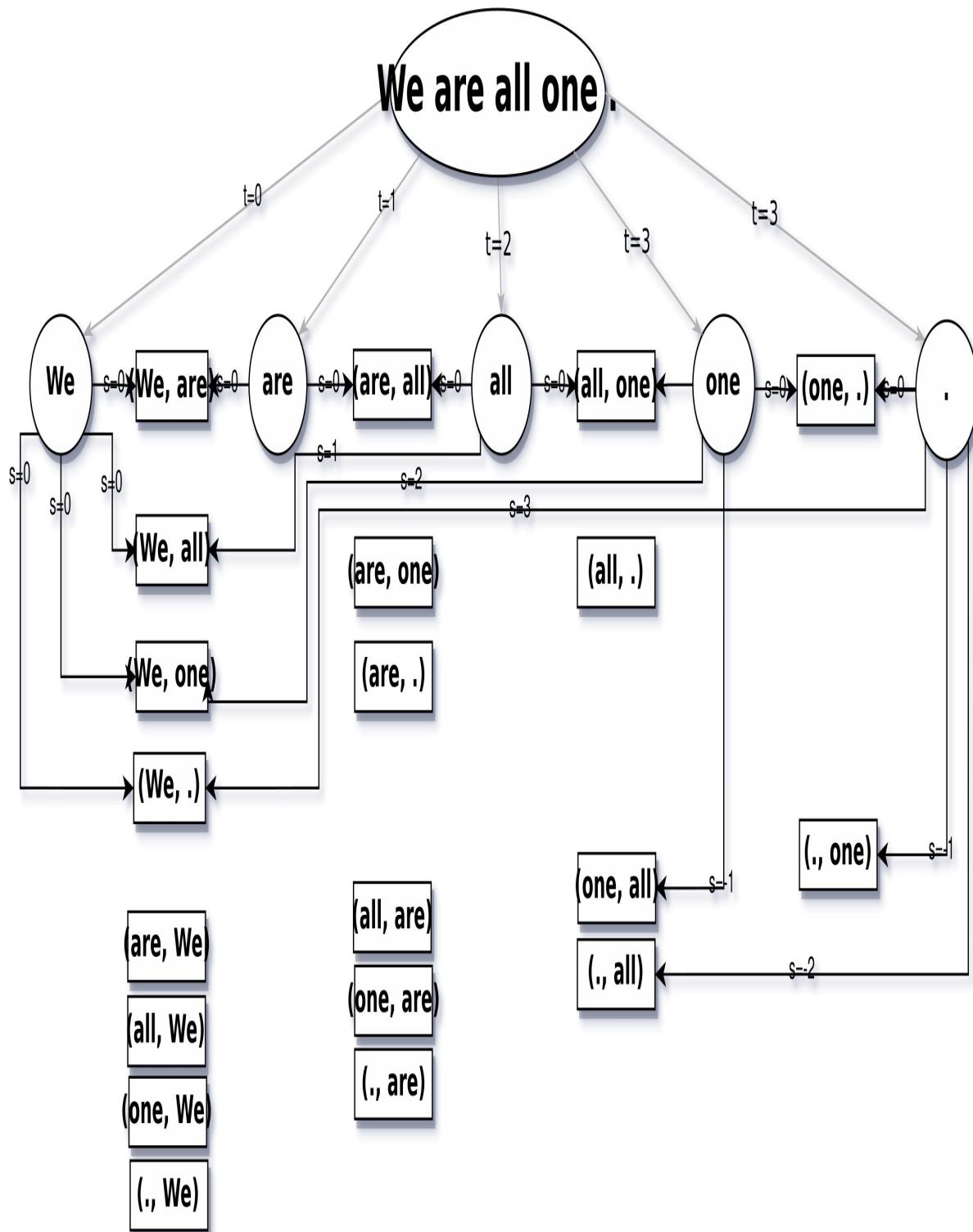
Definition

A *skip-gram* is a 2-gram or pair of grams where each gram is within the neighborhood of each other. As usual the grams can be whatever chunks of text your tokenizer is designed to predict - usually words.

For the continuous skip-gram training approach, skip-grams are word pairs that skip over zero to four words to create the skip-gram pair. When training word embeddings using the Word2Vec skip-gram method, the first word in a skip-gram is called the "context" word. The context word is the input to the Word2Vec neural network. The second word in the skip-gram pair is often called the "target" word. The target words is the word that the language model and embedding vector is being trained to predict - the output.

Figure 6.6. Training input and output example for the skip-gram approach

We are all one. And if we don't know it, we will find out the ha...



Here's what the neural network architecture looks like for the skip-gram approach to creating word embeddings.

6.4.3 Contextualized embeddings

There are two kinds of word embeddings you may encounter in the real world:

1. static
2. contextualized

Static word embeddings can be used on individual words or N-Grams in isolation. And once the training is completed, the vectors remain fixed. These are the kinds of word embeddings you'll use for analogy and other word vector reasoning problems you want to solve. You'll train a language model to create static word embeddings here. The context of a word will only be used to train the model. Once your word embeddings are trained, you will not use the context of a word's usage to adjust your word embeddings at all as you are *using* your trained word embeddings. This means that the different senses or meanings of a word are all smushed together into a single static vector.

In contrast, contextualized word embeddings can be updated or refined based on the embeddings and words that come before or after. And the order a word appears relative to other words matters for contextualized word embeddings. This means that for NLU of the bigram "not happy" it would have an embedding much closer to the embedding of "unhappy" for contextualized word embeddings than for static word embeddings.

Though Word2vec was the first word embedding algorithm, the Papers with Code website lists GloVe and fastText among the top 3 most popular approaches for researchers. Here are the most popular software packages for training static word embeddings: [\[230\]](#)

1. Word2Vec: "Efficient Estimation of Word Representations in Vector Space" - a fully-connected (dense) neural network with a single hidden layer footnote: ["Efficient Estimation of Word Representations in Vector

- Space", 2013, Mikolov et al (<https://arxiv.org/pdf/1301.3781v3.pdf>)]
- 2. "GloVe: Global Vectors for Word Representation" - PCA on the skip-gram window (2014)
 - 3. fastText: "Enriching Word Vectors with Subword Information" (2016)

Technically the GloVe package does not require the explicit construction of skip-grams. Skipping the skip-gram step can save a lot of time and even improve accuracy and numerical stability. In addition, GloVe requires only a minor adjustment to the latent semantic analysis (LSA) algorithm of chapter 3. So it has a strong theoretical basis. Most researchers now prefer GloVe for training English word embeddings.

The fastText software is a character-based algorithm that is designed to handle parts of words, also called "subwords." The fastText tokenizer will create vectors for two halves of a longer word if the longer word used much less often than the subwords that make it up. For example fastText might create vectors for "super" and "woman" if your corpus only mentions "Superwoman" once or twice but uses "super" and "woman" thousands of times. And if your fastText language model encounters the word "Superwoman" in the real world after training is over, it sums the vectors for "Super" and "woman" together to create a vector for the word "Superwoman". This reduces the number of words that fastText will have to assign the generic Out of Vocabulary (OOV) vector to. In the "mind" of your NLU pipeline, the OOV word vector looks like "Unknown Word". It has the same effect as if you heard a foreign word in a completely unfamiliar language. The fastText approach is more statistically justified, ensuring that you'll get better results more often.

The original Word2Vec skip-gram training approach is shown here because this will make it easier for you to understand encoder-decoder neural network architectures later. We've included the fastText logic for creating new vectors for OOV words in the nessvec package as well as in examples here. We've also added an enhancement to the fastText pipeline to handle misspellings and typos using Peter Norvig's famously elegant spelling corrector algorithm.^[231] This will give you the best of both worlds, an understandable training algorithm and a robust inference or prediction model when you need to use your trained vectors in the real world.

What about contextualized word embeddings? ELMo is a algorithm and open source software package for creating contextualized word embeddings. ELMo requires a bidirectional recurrent neural network which you'll learn about in later chapters.[\[232\]](#) Nonetheless, pretrained ELMo word embeddings are available from Allen AI.[\[233\]](#)

And the creators of SpaCy have come up with an efficient contextualized word embedding algorithm that is as easy to use as the SpaCy package. They called their new package Sense2Vec.[\[234\]](#)

What is softmax?

The *softmax function* is often used as the activation function in the output layer of neural networks when the network's goal is to learn classification problems. The softmax will squash the output results between 0 and 1, and the sum of all output notes will always add up to 1. That way, the results of an output layer with a softmax function can be considered as probabilities.

For each of the K output nodes, the softmax output value of the can be calculated using the normalized exponential function:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

If your output vector of a three-neuron output layer looks like this:

Equation 6.3 Example 3D vector

$$v = \begin{bmatrix} 0.5 \\ 0.9 \\ 0.2 \end{bmatrix}$$

The "squashed" vector after the softmax activation would look like this:

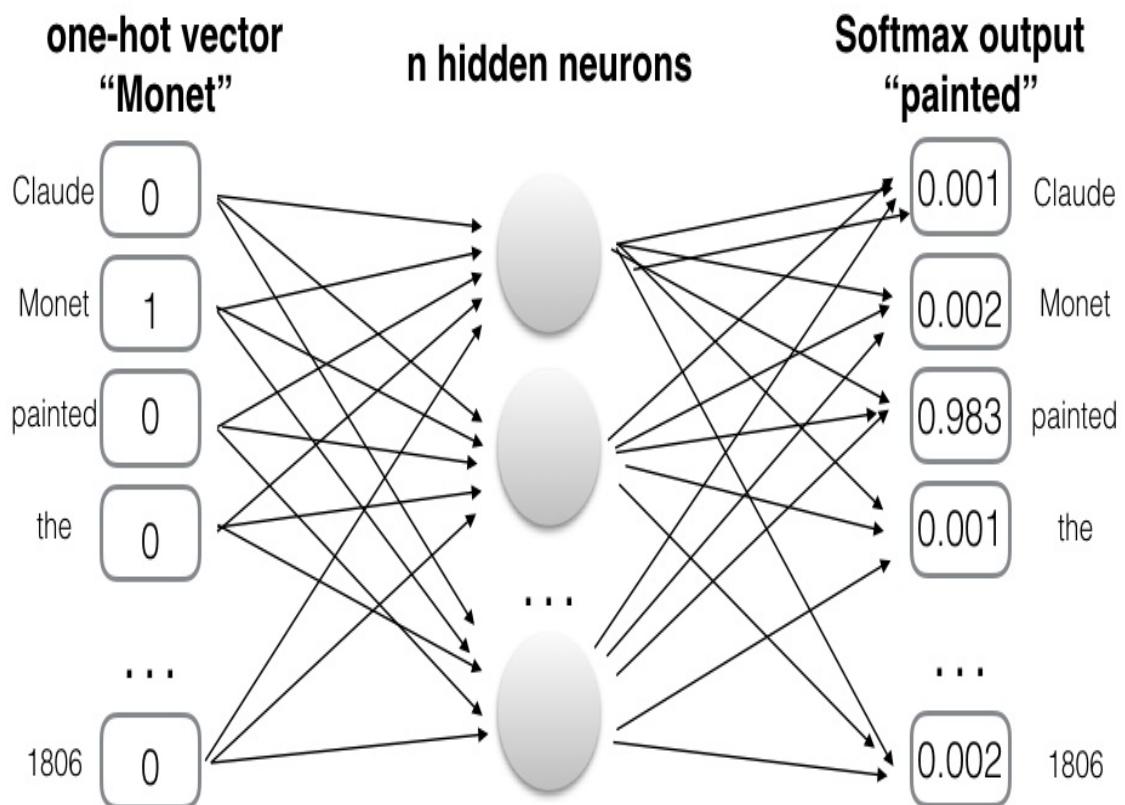
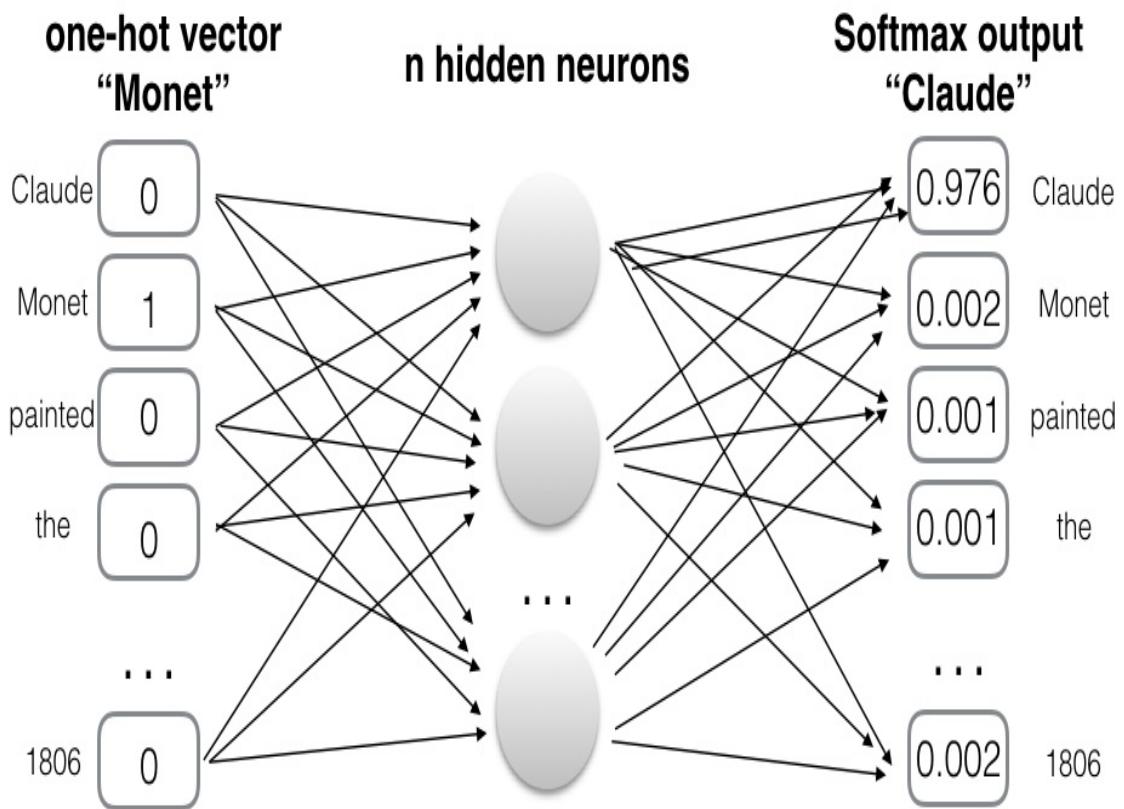
Equation 6.4 Example 3D vector after softmax

$$\sigma(v) = \begin{bmatrix} 0.309 \\ 0.461 \\ 0.229 \end{bmatrix}$$

Notice that the sum of these values (rounded to 3 significant digits) is approximately 1.0, like a probability distribution.

Figure 6.4 shows 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 6.7. Network example for the skip-gram training





Note

When you look at the structure of the neural network for word embedding, you'll notice that the implementation looks similar to what you discovered in chapter 5.

6.4.4 Learning meaning without a dictionary

For this Word2Vec training example you won't need to use a dictionary, such as wiktionary.org to explicitly define the meaning of words. Instead you can just have Word2Vec read text that contains meaningful sentences. You'll use the WikiText2 corpus that comes with PyTorch in the `torchtext` package.

```
>>> import torchtext  
  
>>> dsets = torchtext.datasets.WikiText2()  
>>> num_texts = 10000  
  
>>> filepath = DATA_DIR / f'WikiText2-{num_texts}.txt'  
>>> with open(filepath, 'wt') as fout:  
...     fout.writelines(list(dsets[0])[:num_texts])
```

To make it even less mysterious you can look at the text file you just created with about 10,000 paragraphs of from the `WikiText2` dataset:

```
>>> !tail -n 3 ~/nessvec-data/WikiText2-10000.txt  
  
When Marge leaves Dr. Zweig 's office , she says ,  
" Whenever the wind whistles through the leaves ,  
I 'll think , Lowenstein , Lowenstein ... " .  
This is a reference to The Prince of Tides ; the <unk> is Dr. Low  
= = Reception = =
```

The 99,998th paragraph just happens to contain the abbreviation "Dr.". In this case the abbreviation is for the word "doctor." You can use this to practice your "Mommy is a doctor" intuition pump. So you'll soon find out whether Word2Vec can learn what a doctor really is. Or maybe it will get confused by

street addresses that use "Dr." to mean "drive".

Conveniently, the WikiText2 dataset has already tokenized the text into words for you. Words are delimited with a single space (" ") character. So your pipeline doesn't have to decide whether "Dr." is the end of a sentence or not. If the text was not tokenized, your NLP pipeline would need to remove periods from tokens at the ends of all sentences. Even the heading delimiter text "==" has been split into two separate tokens "=" and "==". And paragraphs are delimited by a newline ("\n") character. And many "pararaphs" will be created for Wikipedia headings such as "==" Reception == as well as retaining all empty lines between paragraphs.

You can utilize a sentence boundary detector or sentence segmenter such as SpaCy to split paragraphs into sentences. This would prevent your training pairs of words from spilling over from one sentence to the other. Honoring sentence boundaries with your Word2Vec can improve the accuracy of your word embeddings. But we'll leave that to you to decide if you need the extra boost in accuracy.

One critical piece of infrastructure that your pipeline here can handle is the memory management for large corpora. If you were training your word embeddings on millions of paragraphs you will need to use a dataset object that manages the text on disk, only loading into RAM or the GPU what is needed. The Hugging Face Hub datasets package can handle this for you:

```
>>> import datasets
>>> dset = datasets.load_dataset('text', data_files=str(filepath))
>>> dset
DatasetDict({
    train: Dataset({
        features: ['text'],
        num_rows: 10000
    })
})
```

But you still need to tell Word2Vec what a word is. This is the only "supervising" of Word2Vec dataset that you need to worry about. And you can use the simplest possible tokenizer from chapter 2 to achieve good results. For this space-delimited tokenized text, you can just use the str.split() method. And you can use case folding with str.lower() to cut

your vocabulary size in half. Surprisingly, this is enough for Word2Vec to learn the meaning and connotation of words sufficiently well for the magic of analogy problems like you might see on an SAT test and even reason about the real world objects and people.

```
def tokenize_row(row):
    row['all_tokens'] = row['text'].lower().split()
    return row
```

Now you can use your tokenizer on the torchtext dataset that contains this iterable sequence of rows of data, each with a "text" key for the WikiText2 data.

```
>>> dset = dset.map(tokenize_row)
>>> dset

DatasetDict({
    train: Dataset({
        features: ['text', 'tokens'],
        num_rows: 10000
    })
})
```

You'll need to compute the vocabulary for your dataset to handle the one-hot encoding and decoding for your neural network.

```
>>> vocab = list(set(
...     [tok for row in dset['train']['tokens'] for tok in row]))
>>> vocab[:4]
['cast', 'kaifeng', 'recovered', 'doctorate']

>>> id2tok = dict(enumerate(vocab))
>>> list(id2tok.items())[:4]
[(0, 'cast'), (1, 'kaifeng'), (2, 'recovered'), (3, 'doctorate')]

>>> tok2id = {tok: i for (i, tok) in id2tok.items()}
>>> list(tok2id.items())[:4]
[('cast', 0), ('kaifeng', 1), ('recovered', 2), ('doctorate', 3)]
```

The one remaining feature engineering step is to create the skip-gram pairs using by windowizing the token sequences and then pairing up the skip-grams within those windows.

```

WINDOW_WIDTH = 10

>>> def windowizer(row, wsize=WINDOW_WIDTH):
    """ Compute sentence (str) to sliding-window of skip-gram pairs """
    doc = row['tokens']
    out = []
    for i, wd in enumerate(doc):
        target = tok2id[wd]
        window = [
            i + j for j in range(-wsize, wsize + 1, 1)
            if (i + j >= 0) & (i + j < len(doc)) & (j != 0)
        ]
        out += [(target, tok2id[doc[w]]) for w in window]
    row['moving_window'] = out
    return row

```

Once you apply the windowizer to your dataset it will have a 'window' key where the windows of tokens will be stored.

```

>>> dset = dset.map(windowizer)
>>> dset
DatasetDict({
    train: Dataset({
        features: ['text', 'tokens', 'window'],
        num_rows: 10000
    })
})

```

Here's your skip_gram generator function:

```

>>> def skip_grams(tokens, window_width=WINDOW_WIDTH):
...     pairs = []
...     for i, wd in enumerate(tokens):
...         target = tok2id[wd]
...         window = [
...             i + j for j in
...             range(-window_width, window_width + 1, 1)
...             if (i + j >= 0)
...             & (i + j < len(tokens))
...             & (j != 0)
...         ]
...
...         pairs.extend([(target, tok2id[tokens[w]]) for w in window])
# huggingface datasets are dictionaries for every text element
...     return pairs

```

Your neural network only needs the pairs of skip-grams from the windowed data:

```
>>> from torch.utils.data import Dataset

>>> class Word2VecDataset(Dataset):
...     def __init__(self, dataset, vocab_size, wsize=WINDOW_WIDTH
...                 self.dataset = dataset
...                 self.vocab_size = vocab_size
...                 self.data = [i for s in dataset['moving_window'] for i
...
...     def __len__(self):
...         return len(self.data)
...
...     def __getitem__(self, idx):
...         return self.data[idx]
```

And your DataLoader will take care of memory management for you. This will ensure your pipeline is reusable for virtually any size corpus, even all of Wikipedia.

```
from torch.utils.data import DataLoader

dataloader = {}
for k in dset.keys():
    dataloader = {
        k: DataLoader(
            Word2VecDataset(
                dset[k],
                vocab_size=len(vocab)),
            batch_size=BATCH_SIZE,
            shuffle=True,
            num_workers=CPU_CORES - 1)
    }
```

You need a one-hot encoder to turn your word pairs into one-hot vector pairs:

```
def one_hot_encode(input_id, size):
    vec = torch.zeros(size).float()
    vec[input_id] = 1.0
    return vec
```

To dispell some of the magic of the examples you saw earlier, you'll train the network from scratch, just as you did in chapter 5. You can see that a

Word2Vec neural network is almost identical to your single-layer neural network from the previous chapter.

```
from torch import nn
EMBED_DIM = 100      #1

class Word2Vec(nn.Module):
    def __init__(self, vocab_size=len(vocab), embedding_size=EMBE
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embedding_size)
        self.expand = nn.Linear(embedding_size, vocab_size, bias=

    def forward(self, input):
        hidden = self.embed(input)          #3
        logits = self.expand(hidden)       #4
        return logits
```

Once you instantiate your Word2Vec model you are ready to create 100-D embeddings for the more than 20 thousand words in your vocabulary:

```
>>> model = Word2Vec()
>>> model

Word2Vec(
    (embed): Embedding(20641, 100)
    (expand): Linear(in_features=100, out_features=20641, bias=False)
)
```

If you have a GPU you can send your model to the GPU to speed up the training:

```
>>> import torch
>>> if torch.cuda.is_available():
...     device = torch.device('cuda')
>>> else:
...     device = torch.device('cpu')
>>> device

device(type='cpu')
```

Don't worry if you do not have a GPU. On most modern CPUs this Word2Vec model will train in less than 15 minutes.

```
>>> model.to(device)
```

```

Word2Vec(
    (embed): Embedding(20641, 100)
    (expand): Linear(in_features=100, out_features=20641, bias=False)
)

```

Now is the fun part! You get to watch as Word2Vec quickly learns the meaning of "Dr." and thousands of other tokens, just by reading a lot of text. You can go get a tea or some chocolate or just have a 10 minute meditation to contemplate the meaning of life while your laptop contemplates the meaning of words. First, let's define some training parameters

```

>>> from tqdm import tqdm # noqa
>>> EPOCHS = 10
>>> LEARNING_RATE = 5e-4
EPOCHS = 10
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=LEARNING_RATE)

running_loss = []
pbar = tqdm(range(EPOCHS * len(dataloader['train'])))
for epoch in range(EPOCHS):
    epoch_loss = 0
    for sample_num, (center, context) in enumerate(dataloader['train']):
        if sample_num % len(dataloader['train']) == 2:
            print(center, context)
            # center: tensor([ 229,      0,  2379,     ... ,   402,   553,
            # context: tensor([ 112,  1734,   802,     ... ,    28,   852
        center, context = center.to(device), context.to(device)
        optimizer.zero_grad()
        logits = model(input=context)
        loss = loss_fn(logits, center)
        if not sample_num % 10000:
            # print(center, context)
            pbar.set_description(f'loss[{sample_num}] = {loss.item():.4f}')
        epoch_loss += loss.item()
        loss.backward()
        optimizer.step()
        pbar.update(1)
    epoch_loss /= len(dataloader['train'])
    running_loss.append(epoch_loss)

save_model(model, loss)

```

6.4.5 Computational tricks of Word2Vec

After the initial publication, the performance of word2vec models have been improved through various computational tricks. In this section, we highlight the three key improvements that help word embeddings achieve greater accuracy with less computational resources or training data:

1. Add frequent bigrams to the vocabulary
2. Undersampling (subsampling) frequent tokens
3. Undersampling of negative examples

Frequent bigrams

Some words often occur in combination with other words creating a compound word. For example "Aaron" is often followed by "Swartz" and "AI" is often followed by "Ethics". Since the word "Swartz" would follow the word "Aaron" with an above average probability you probably want to create a single word vector for "Aaron Swartz" as a single compound proper noun. In order to improve the accuracy of the Word2Vec embedding for their applications involving proper nouns and compound words, Mikolov's team included some bigrams and trigrams in their Word2Vec vocabulary. The team [\[235\]](#) used co-occurrence frequency to identify bigrams and trigrams that should be considered single terms using the following scoring function:

Equation 6.5 Bigram scoring function

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

When words occur often enough next to each other, they will be included in the Word2Vec vocabulary as a pair term. You'll notice that the vocabulary of many word embeddings models such as Word2vec contains terms like "New_York" or "San_Francisco". That way, these terms will be represented as a single vector instead of two separate ones, such as for "San" and "Francisco".

Another effect of the word pairs is that the word combination often represents

a different meaning than the sum of the vectors for the individual words. For example, the MLS soccer team "Portland Timbers" has a different meaning than the individual words "Portland" or "Timbers". But by adding oft-occurring bigrams like team names to the word2vec model, they can easily be included in the one-hot vector for model training.

Undersampling frequent tokens

Another accuracy improvement to the original algorithm was to undersample (subsample) frequent words. This is also referred to as "undersampling the majority classes" in order to balance the class weights. Common words such as the articles "the" and "a" often don't contain a lot of information and meaning relevant to most NLP problems so they are referred to as stop words. Mikolov and others often chose to *subsample* these words.

Subsampling just means you randomly ignore them during your sampling of the corpus of continuous skip-grams or CBOWs. Many blogger will take this to the extreme and completely remove them from the corpus during prepossessing. Though subsampling or filtering stopwords may help your word embedding algorithm train faster, it can sometimes be counterproductive. And with modern computers and applications, a 1% improvement in training time is not likely to be worth the loss in precision of your word vectors. FastText It might be helpful in a small corprusAnd the co-occurrence of stop words with other "words" in the corpus might create less meaningful connections between words muddying the Word2Vec representation with this false semantic similarity training.



Important

All words carry meaning, including stop words. So stop words should not be completely ignored or skipped while training your word vectors or composing your vocabulary. In addition, because word vectors are often used in generative models (like the model Cole used to compose sentences in this book), stop words and other common words must be included in your vocabulary and are allowed to affect the word vectors of their neighboring words.

To reduce the emphasis on frequent words like stop words, words are sampled during training in inverse proportion to their frequency. The effect of this is similar to the IDF affect on TF-IDF vectors. Frequent words are given less influence over the vector than the rarer words. Tomas Mikolov used the following equation to determine the probability of sampling a given word. This probability determines whether or not a particular word is included in a particular skip-gram during training:

Equation 6.6 Subsampling probability in Mikolov's Word2Vec paper

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

The word2vec C++ implementation uses a slightly different sampling probability than the one mentioned in the paper, but it has the same effect:

Equation 6.7 Subsampling probability in Mikolov's word2vec code

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

In the preceding equations, $f(w_i)$ represents the frequency of a word across the corpus, and t represents a frequency threshold above which you want to apply the subsampling probability. The threshold depends on your corpus size, average document length, and the variety of words used in those documents. Values between 10^{-5} and 10^{-6} are often found in the literature.

If a word shows up 10 times across your entire corpus, and your corpus has a vocabulary of one million distinct words, and you set the subsampling threshold to 10^{-6} , 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.

Mikolov showed that subsampling improves the accuracy of the word vectors for tasks such as answering analogy questions.

Negative sampling

One last trick the Mikolov came up with was the idea of negative sampling. If a single training example with a pair of words is presented to the network it will cause all weights for the network to be updated. This changes the values of all the vectors for all the words in your vocabulary. But if your vocabulary contains thousands or millions of words, updating all the weights for the large one-hot vector is inefficient. To speed up the training of word vector models, Mikolov used negative sampling.

Instead of updating all word weights that weren't included in the word window, Mikolov suggested sampling just a few negative samples (in the output vector) to update their weights. Instead of updating all weights, you pick n negative example word pairs (words that don't match your target output for that example) and update the weights that contributed to their specific output. That way, the computation can be reduced dramatically and the performance of the trained network doesn't decrease significantly.



Note

If you train your word model with a small corpus, you might want to use a negative sampling rate of 5 to 20 samples. For larger corpora and vocabularies, you can reduce the negative sample rate to as low as two to five samples, according to Mikolov and his team.

6.4.6 Using the `gensim.word2vec` module

If the previous section sounded too complicated, don't worry. Various companies provide their pretrained word vector models, and popular NLP libraries for different programming languages allow you to use the pretrained models efficiently. In the following section, we look at how you can take

advantage of the magic of word vectors. For word vectors you'll use the popular `gensim` library, which you first saw in chapter 4.

If you've already installed the `nlpia` package,^[236] you can download a pretrained `word2vec` model with the following command:

```
>>> from nlpia.data.loaders import get_data  
>>> word_vectors = get_data('word2vec')
```

If that doesn't work for you, or you like to "roll your own," you can do a Google search for `word2vec` models pretrained on Google News documents.^[237] After you find and download the model in Google's original binary format and put it in a local path, you can load it with the `gensim` package like this:

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

Working with word vectors can be memory intensive. If your available memory is limited or if you don't want to wait minutes for the word vector model to load, you can reduce the number of words loaded into memory by passing in the `limit` keyword argument. In the following example, you'll load the 200k most common words from the Google News corpus:

```
>>> from gensim.models.keyedvectors import KeyedVectors  
>>> from nlpia.loaders import get_data  
>>> word_vectors = get_data('w2v', limit=200000)      #1
```

But keep in mind that a word vector model with a limited vocabulary will lead to a lower performance of your NLP pipeline if your documents contain words that you haven't loaded word vectors for. Therefore, you probably only want to limit the size of your word vector model during the development phase. For the rest of the examples in this chapter, you should use the complete Word2Vec model if you want to get the same results we show here.

The `gensim.KeyedVectors.most_similar()` method provides an efficient way to find the nearest neighbors for any given word vector. The keyword argument `positive` takes a list of the vectors to be added together, similar to your soccer team example from the beginning of this chapter. Similarly, you

can use the negative argument for subtraction and to exclude unrelated terms. The argument `topn` determines how many related terms should be provided as a return value.

Unlike a conventional thesaurus, Word2Vec synonymy (similarity) is a continuous score, a distance. This is because Word2Vec itself is a continuous vector space model. Word2Vec's high dimensionality and continuous values for each dimension enable it to capture the full range of meaning for any given word. That's why analogies and even zeugmas, odd juxtapositions of multiple meanings within the same word, are no problem. Handling analogies and zeugmas is a really big deal. Understanding analogies and zeugmas takes human-level understanding of the world, including common sense knowledge and reasoning.^[238] Word embeddings are enough to give machines at least a passing understanding on the kinds of analogies you might see on an SAT quiz.

```
>>> 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=5)
[('europe', 0.7222039699554443)]
```

Word vector models also allow you to determine unrelated terms. The `gensim` library provides a method called `doesnt_match`:

```
>>> word_vectors.doesnt_match("potatoes milk cake computer".split())
['computer']
```

To determine the most unrelated term of the list, the method returns the term with the highest distance to all other list terms.

If you want to perform calculations (such as 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 square bracket syntax (`[]`) or the `get()` method on a `KeyedVector` instance. You can treat the loaded model object as a dictionary where your word of interest is the dictionary key. Each float in the returned array represents one of the vector dimensions. In the case of Google's word model, your numpy arrays will have a shape of `1x300`.

```
>>> word_vectors['phone']
array([-0.01446533, -0.12792969, -0.11572266, -0.22167969, -0.073
       -0.05981445, -0.10009766, -0.06884766, 0.14941406, 0.101
       -0.03076172, -0.03271484, -0.03125, -0.10791016, 0.121
       0.16015625, 0.19335938, 0.0065918, -0.15429688, 0.037
       ...]
```

If you're wondering what all those numbers *mean*, you can find out. But it would take a lot of work. You would need to examine some synonyms and see which of the 300 numbers in the array they all share. Alternatively you can find the linear combination of these numbers that make up dimensions for things like "placeness" and "femaleness", like you did at the beginning of this chapter.

6.4.7 Generating your own Word vector representations

In some cases you may want to create your own domain-specific word vector models. Doing so can improve the accuracy of your model if your NLP pipeline is processing documents that use words in a way that you wouldn't find on Google News before 2006, when Mikolov trained the reference word2vec model. Keep in mind, you need a *lot* of documents to do this as well as Google and Mikolov did. But if your words are particularly rare on Google News, or your texts use them in unique ways within a restricted domain, such as medical texts or transcripts, a domain-specific word model

may improve your model accuracy. In the following section, we show you how to train your own word2vec model.

For the purpose of training a domain-specific word2vec model, you'll again turn to gensim, but before you can start training the model, you'll need to preprocess your corpus using tools you discovered in chapter 2.

Preprocessing steps

First you need to break your documents into sentences and the sentences into tokens. The gensim word2vec model expects a list of sentences, where each sentence is broken up into tokens. This prevents word vectors learning from irrelevant word occurrences in neighboring sentences. Your training input should look similar to the following structure:

```
>>> token_list
[
    ['to', 'provide', 'early', 'intervention/early', 'childhood', 'education', 'services', 'to', 'eligible', 'children', 'and', 'families'],
    ['essential', 'job', 'functions'],
    ['participate', 'as', 'a', 'transdisciplinary', 'team', 'member', 'complete', 'educational', 'assessments', 'for']
    ...
]
```

To segment sentences and then convert sentences into tokens, you can apply the various strategies you learned in chapter 2. Let's add another one:

Detector Morse is a sentence segmenter that improves upon the accuracy segmenter available in NLTK and gensim for some applications.[\[239\]](#) It 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, Detector Morse is likely to give you the highest accuracy currently possible. You can also retrain Detector Morse on your own dataset if you have a large set of sentences from your domain. Once you've converted your documents into lists of token lists (one for each sentence), you're ready for your word2vec training.

Train your domain-specific word2vec model

Get started by loading the *word2vec* module:

```
>>> from gensim.models.word2vec import Word2Vec
```

The training requires a few setup details.

Listing 6.2. Parameters to control word2vec model training

```
>>> num_features = 300      #1
>>> min_word_count = 3       #2
>>> num_workers = 2          #3
>>> window_size = 6          #4
>>> subsampling = 1e-3        #5
```

Now you're ready to start your training.

Listing 6.3. Instantiating a word2vec model

```
>>> model = Word2Vec(
...     token_list,
...     workers=num_workers,
...     size=num_features,
...     min_count=min_word_count,
...     window>window_size,
...     sample=subsampling)
```

Depending on your corpus size and your CPU performance, the training will take a significant amount of time. For smaller corpora, the training can be completed in minutes. But for a comprehensive word model, the corpus will contain millions of sentences. You need to have several examples of all the different ways all the different words in your corpus are used. If you start processing larger corpora, such as the Wikipedia corpus, expect a much longer training time and a much larger memory consumption.

In addition, Word2Vec models can consume quite a bit of memory. But remember that only the weight matrix for the hidden layer is of interest. Once you've trained your word model, you can reduce the memory footprint by about half if you freeze your model and discard the unnecessary information. The following command will discard the unneeded output weights of your neural network:

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

The `init_sims` method will freeze the model, storing the weights of the hidden layer and discarding the output weights that predict word co-occurrences. The output weights aren't part of the vector used for most Word2Vec applications. But the model cannot be trained further once the weights of the output layer have been discarded.

You can save the trained model with the following command and preserve it for later use:

```
>>> model_name = "my_domain_specific_word2vec_model"  
>>> model.save(model_name)
```

If you want to test your newly trained model, you can use it with the same method you learned in the previous section.

Listing 6.4. Loading a saved word2vec model

```
>>> from gensim.models.word2vec import Word2Vec  
>>> model_name = "my_domain_specific_word2vec_model"  
>>> model = Word2Vec.load(model_name)  
>>> model.most_similar('radiology')
```

6.4.8 Word2Vec vs GloVe (Global Vectors)

Word2Vec was a breakthrough, but it relies on a neural network model that must be trained using backpropagation. Since Mikolov first popularized word embeddings, researchers have come up with increasingly more accurate and efficient ways to embed the meaning of words in a vector space.

1. Word2vec
2. GloVe
3. fastText
4. Bloom Embeddings [\[240\]](#)

Backpropagation is usually less efficient than direct optimization of a cost function using gradient descent. Stanford NLP researchers [\[241\]](#) led by Jeffrey Pennington set about to understand the reason why Word2Vec worked so

well and to find the cost function that was being optimized. They started by counting the word co-occurrences and recording them in a square matrix. They found they could compute the singular value decomposition (SVD)^[242] of this co-occurrence matrix, splitting it into the same two weight matrices that Word2Vec produces.^[243] The key was to normalize the co-occurrence matrix the same way. But in some cases the Word2Vec model failed to converge to the same global optimum that the Stanford researchers were able to achieve with their SVD approach. It's this direct optimization of the global vectors of word co-occurrences (co-occurrences across the entire corpus) that gives GloVe its name.

GloVe can produce matrices equivalent to the input weight matrix and output weight matrix of Word2Vec, producing a language model with the same accuracy as Word2Vec but in much less time. GloVe speeds the process by using the text data more efficiently. GloVe can be trained on smaller corpora and still converge.^[244] And SVD algorithms have been refined for decades, so GloVe has a head start on debugging and algorithm optimization. Word2Vec relies on backpropagation to update the weights that form the word embeddings. Neural network backpropagation is less efficient than more mature optimization algorithms such as those used within SVD for GloVe

Even though Word2Vec first popularized the concept of semantic reasoning with word vectors, your workhorse should probably be GloVe to train new word vector models. With GloVe you'll be more likely to find the global optimum for those vector representations, giving you more accurate results.

Advantages of GloVe:

- Faster training
- Better RAM/CPU efficiency (can handle larger documents)
- More efficient use of data (helps with smaller corpora)
- More accurate for the same amount of training

6.4.9 fastText

Researchers from Facebook took the concept of Word2Vec one step further

[245] by adding a new twist to the model training. The new algorithm, which they named fastText, predicts the surrounding *n-character* grams rather than just the surrounding words, like Word2Vec does. For example, the word "whisper" would generate the following 2- and 3-character grams:

```
['wh', 'whi', 'hi', 'his', 'is', 'isp', 'sp', 'spe', 'pe', 'per',
```

fastText is then training a vector representation for every *n*-character gram, which includes words, misspelled words, partial words, and even single characters. The advantage of this approach is that it handles rare words much better than the original Word2Vec approach.

As part of the fastText release, Facebook published pretrained fastText models for 294 languages. On the Github page of Facebook research, [246] you can find models ranging from *Abkhazian* to *Zulu*. The model collection even includes rare languages such as *Saterland Frisian*, which is only spoken by a handful of Germans. The pretrained fastText models provided by Facebook have only been trained on the available Wikipedia corpora. Therefore the vocabulary and accuracy of the models will vary across languages.

Power up your NLP with pretrained model

Supercharge your NLP pipeline by taking advantage of the open source pretrained embeddings from the most powerful corporations on the planet. Pretrained fastText vectors are available in almost every language conceivable. An If you want to see the all the options available for your word embeddings check out the fastText model repository (<https://github.com/facebookresearch/fastText/blob/main/docs/pretrained-vectors.md>). And for multilingual power you can find combined models for many of the 157 languages supported in the Common Crawl version of fastText embeddings (<https://fasttext.cc/docs/en/crawl-vectors.html>). If you want you can download all the different versions of the embeddings for your language using the *bin+text* links on the fastText pages. But if you want to save some time and just download the 1 million



Warning

The *bin+text* `wiki.en.zip` file (<https://dl.fbaipublicfiles.com/fasttext/vectors-wiki/wiki.en.zip>) is 9.6 GB. The text-only `wiki.en.vec` file is 6.1 GB (<https://dl.fbaipublicfiles.com/fasttext/vectors-wiki/wiki.en.vec>). If you use the `nessvec` package rather than `gensim` it will download just the 600MB `wiki-news-300d-1M.vec.zip` file (<http://mng.bz/AI6x>). That `wiki-news-300d-1M.vec.zip` contains the 300-D vectors for the 1 million most popular words (case-insensitive) from Wikipedia and news web pages.

The `nessvec` package will create a memory-mapped `DataFrame` of all your pretrained vectors. The memory-mapped file (`.hdf5`) keeps you from running out of memory (RAM) on your computer by lazy-loading just the vectors you need, when you need them.

```
>>> from nessvec.files import load_fasttext
>>> df = load_fasttext()      #1
>>> df.head().round(2)
          0        1        2      ...     297     298     299
,    0.11   0.01   0.00      ...    0.00   0.12  -0.04
the  0.09   0.02  -0.06      ...    0.16  -0.03  -0.03
.    0.00   0.00  -0.02      ...    0.21   0.07  -0.05
and -0.03   0.01  -0.02      ...    0.10   0.09   0.01
of  -0.01  -0.03  -0.03      ...    0.12   0.01   0.02
>>> df.loc['prosocial']    #2
0        0.0004
1       -0.0328
2      -0.1185
...
297     0.1010
298    -0.1323
299     0.2874
Name: prosocial, Length: 300, dtype: float64
```



Note

To turbocharge your word embedding pipeline you can use Bloom embeddings. Bloom embeddings aren't a new algorithm for creating embeddings, but a faster more accurate indexing approach for storing and retrieving a high dimensional vector. The vectors in a Bloom embedding table each represent the meaning of two or more words combined together. The trick is to subtract out the words you don't need in order to recreate the

original embedding that you're looking for. Fortunately SpaCy has implemented all this efficiency under the hood with its v2.0 language model. This is how SpaCy can create word embeddings for millions of words while storing only 20k unique vectors.[\[247\]](#)

6.4.10 Word2Vec vs LSA

You might now be wondering how word embeddings compare to the LSA topic-word vectors of chapter 4. These are word embeddings that you created using PCA (principal component analysis) on your TF-IDF vectors. And LSA also gives you topic-document vectors which you used as embeddings of entire documents. LSA topic-document vectors are the sum of the topic-word vectors for all the words in whatever document you create the embedding for. If you wanted to get a word vector for an entire document that is analogous to topic-document vectors, you'd sum all the word vectors for your document. That's pretty close to how Doc2vec document vectors work.

If your LSA matrix of topic vectors is of size $N_{\text{words}} \times N_{\text{topics}}$, 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 to 300 real values, like Word2Vec does. And LSA topic-word vectors are just as useful as Word2Vec vectors for finding both related and unrelated terms. As you learned in the GloVe discussion, Word2Vec vectors can be created using the exact same SVD algorithm used for LSA. But Word2Vec gets more use out of the same number of words in its documents by creating a sliding window that overlaps from one document to the next. This way it can reuse the same words five times before sliding on.

What about incremental or online training? Both LSA and Word2Vec algorithms allow adding new documents to your corpus and adjusting your existing word vectors to account for the co-occurrences in the new documents. But only the existing "bins" in your lexicon can be updated. Adding completely new words would change the total size of your vocabulary and therefore your one-hot vectors would change. That requires starting the training over if you want to capture the new word in your model.

LSA trains faster than Word2Vec does. And for long documents, it does a

better job of discriminating and clustering those documents. In fact, Stanford researchers used this faster PCA-based method to train the GloVe vectors. You can compare the three most popular word embeddings using the `nessvec` package.[\[248\]](#)

The "killer app" for Word2Vec is the semantic reasoning that it made possible. LSA topic-word vectors can do that, too, but it usually isn't accurate. You'd have to break documents into sentences and then only use short phrases to train your LSA model if you want to approach the accuracy and "wow" factor of Word2Vec reasoning. With Word2Vec you can determine the answer to questions like *Harry Potter + "University" = Hogwarts*. As a great example for domain-specific word2vec models, check out the models for words from Harry Potter, the Lord of the Rings by [\[249\]](#).

Advantages of LSA:

- Faster training
- Better discrimination between longer documents

Advantages of Word2Vec and GloVe:

- More efficient use of large corpora
- More accurate reasoning with words, such as answering analogy questions

6.4.11 Visualizing word relationships

The semantic word relationships can be powerful and their visualizations can lead to interesting discoveries. In this section, we demonstrate steps to visualize the word vectors in 2D.

To get started, let's load all the word vectors from the Google Word2Vec model of the Google News corpus. As you can imagine, this corpus included a lot of mentions of "Portland" and "Oregon" and a lot of other city and state names. You'll use the `nlpia` package to keep things simple, so you can start playing with Word2Vec vectors quickly.

Listing 6.5. Load a pretrained FastText language model using nlpia

```
>>> from nessvec.indexers import Index  
>>> index = Index()      #1  
>>> vecs = index.vecs  
>>> vecs.shape  
(3000000, 300)
```



Warning

The Google News word2vec model is huge: 3 million words with 300 vector dimensions each. The complete word vector model requires 3 GB of available memory. If your available memory is limited or you quickly want to load a few most frequent terms from the word model, check out chapter 13.

This `KeyedVectors` object in `gensim` now holds a table of 3 million Word2Vec vectors. We loaded these vectors from a file created by Google to store a Word2Vec model that they trained on a large corpus based on Google News articles. There should definitely be a lot of words for states and cities in all those news articles. The following listing shows just a few of the words in the vocabulary, starting at the 1 millionth word:

Listing 6.6. Examine word2vec vocabulary frequencies

```
>>> import pandas as pd  
>>> vocab = pd.Series(wv.vocab)  
>>> vocab.iloc[1000000:100006]  
Illington_Fund          Vocab(count:447860, index:2552140)  
Illingworth             Vocab(count:2905166, index:94834)  
Illingworth_Halifax    Vocab(count:1984281, index:1015719)  
Illini                  Vocab(count:2984391, index:15609)  
IlliniBoard.com         Vocab(count:1481047, index:1518953)  
Illini_Bluffs           Vocab(count:2636947, index:363053)
```

Notice that compound words and common *n*-grams are joined together with an underscore character ("`_`"). Also notice that the "value" in the key-value mapping is a `gensim` `Vocab` object that contains not only the index location for a word, so you can retrieve the Word2Vec vector, but also the number of times it occurred in the Google News corpus.

As you've seen earlier, if you want to retrieve the 300-D vector for a particular word, you can use the square brackets on this `KeyedVectors` object

to *getitem* any word or *n*-gram:

```
>>> wv['Illini']
array([ 0.15625   ,  0.18652344,  0.33203125,  0.55859375,  0.036
       -0.09375   , -0.05029297,  0.16796875, -0.0625     ,  0.099
      -0.0291748   ,  0.39257812,  0.05395508,  0.35351562, -0.022
       ...
      ])
```

The reason we chose the 1 millionth word (in lexical alphabetic order) is because the first several thousand "words" are punctuation sequences like "#\#\#\#\#" and other symbols that occurred a lot in the Google News corpus. We just got lucky that "Illini" showed up in your list. The word "Illini" refers to a group of people, usually football players and fans, rather than a single geographic region like "Illinois"—where most fans of the "Fighting Illini" live. Let's see how close this "Illini" vector is to the vector for "Illinois":

Listing 6.7. Distance between "Illinois" and "Illini"

```
>>> import numpy as np
>>> np.linalg.norm(wv['Illinois'] - wv['Illini'])    #1
3.3653798
>>> cos_similarity = np.dot(wv['Illinois'], wv['Illini']) / (
...     np.linalg.norm(wv['Illinois']) * \
...     np.linalg.norm(wv['Illini']))    #2
>>> cos_similarity
0.5501352
>>> 1 - cos_similarity    #3
0.4498648
```

These distances mean that the words "Illini" and "Illinois" are only moderately close to one another in meaning.

Now let's retrieve all the Word2Vec vectors for US cities so you can use their distances to plot them on a 2D map of meaning. How would you find all the cities and states in that Word2Vec vocabulary in that KeyedVectors object? You could use cosine distance like you did in the previous listing to find all the vectors that are close to the words "state" or "city".

But rather than reading through all 3 million words and word vectors, lets load another dataset containing a list of cities and states (regions) from

around the world.

Listing 6.8. Some US city data

```
>>> from nlpia.data.loaders import get_data
>>> cities = get_data('cities')
>>> cities.head(1).T
geonameid           3039154
name                El Tarter
asciiname           El Tarter
alternatenames      Ehл Tarter, Эл Тартер
latitude             42.5795
longitude            1.65362
feature_class        P
feature_code          PPL
country_code          AD
cc2                  NaN
admin1_code            02
admin2_code            NaN
admin3_code            NaN
admin4_code            NaN
population            1052
elevation              NaN
dem                  1721
timezone              Europe/Andorra
modification_date     2012-11-03
```

This dataset from Geocities contains a lot of information, including latitude, longitude, and population. You could use this for some fun visualizations or comparisons between geographic distance and Word2Vec distance. But for now you're just going to try to map that Word2Vec distance on a 2D plane and see what it looks like. Let's focus on just the United States for now:

Listing 6.9. Some US state data

```
>>> us = cities[(cities.country_code == 'US') & \
...      (cities.admin1_code.notnull())].copy()
>>> states = pd.read_csv(\n...     'http://www.fonz.net/blog/wp-content/uploads/2008/04/stat'
>>> states = dict(zip(states.Abbreviation, states.State))
>>> us['city'] = us.name.copy()
>>> us['st'] = us.admin1_code.copy()
>>> us['state'] = us.st.map(states)
>>> us[us.columns[-3:]].head()
```

| | city | st | state |
|-----------|----------------|----|---------|
| geonameid | | | |
| 4046255 | Bay Minette | AL | Alabama |
| 4046274 | Edna | TX | Texas |
| 4046319 | Bayou La Batre | AL | Alabama |
| 4046332 | Henderson | TX | Texas |
| 4046430 | Natalia | TX | Texas |

Now you have a full state name for each city in addition to its abbreviation. Let's check to see which of those state names and city names exist in your Word2Vec vocabulary:

```
>>> vocab = pd.np.concatenate([us.city, us.st, us.state])
>>> vocab = np.array([word for word in vocab if word in wv.wv])
>>> vocab[:10]
```

Even when you only look at United States cities, you'll find a lot of large cities with the same name, like Portland, Oregon and Portland, Maine. So let's incorporate into your city vector the essence of the state where that city is located. To combine the meanings of words in Word2Vec, you add the vectors together. That's the magic of "Analogy reasoning."

Here's one way to add the Word2Vecs for the states to the vectors for the cities and put all these new vectors in a big DataFrame. We use either the full name of a state or just the abbreviations (whichever one is in your Word2Vec vocabulary).

Listing 6.10. Augment city word vectors with US state word vectors

```
>>> city_plus_state = []
>>> for c, state, st in zip(us.city, us.state, us.st):
...     if c not in vocab:
...         continue
...     row = []
...     if state in vocab:
...         row.extend(wv[c] + wv[state])
...     else:
...         row.extend(wv[c] + wv[st])
...     city_plus_state.append(row)
>>> us_300D = pd.DataFrame(city_plus_state)
```

Depending on your corpus, your word relationship can represent different

attributes, such as geographical proximity or cultural or economic similarities. But the relationships heavily depend on the training corpus, and they will reflect the corpus.



Word vectors are biased!

Word vectors learn word relationships based on the training corpus. If your corpus is about finance then your "bank" word vector will be mainly about businesses that hold deposits. If your corpus is about geology the your "bank" word vector will be trained on associations with rivers and streams. And if your corpus is mostly about a matriarchal society with women bankers and men washing clothes in the river, then your word vectors would take on that gender bias.

The following example shows the gender bias of a word model trained on Google News articles. If you calculate the distance between "man" and "nurse" and compare that to the distance between "woman" and "nurse", you'll be able to see the bias.

```
>>> word_model.distance('man', 'nurse')  
0.7453  
>>> word_model.distance('woman', 'nurse')  
0.5586
```

Identifying and compensating for biases like this is a challenge for any NLP practitioner that trains her models on documents written in a biased world.

The news articles used as the training corpus share a common component, which is the semantical similarity of the cities. Semantically similar locations in the articles seems to interchangeable and therefore the word model learned that they are similar. If you would have trained on a different corpus, your word relationship might have differed.

Cities that are similar in size and culture are clustered close together despite being far apart geographically, such as San Diego and San Jose, or vacation destinations such as Honolulu and Reno.

Fortunately you can use conventional algebra to add the vectors for cities to

the vectors for states and state abbreviations. As you discovered in chapter 4, you can use tools such as the principal components analysis (PCA) to reduce the vector dimensions from your 300 dimensions to a human-understandable 2D representation. PCA enables you to see the projection or "shadow" of these 300D vectors in a 2D plot. Best of all, the PCA algorithm ensures that this projection is the best possible view of your data, keeping the vectors as far apart as possible. PCA is like a good photographer that looks at something from every possible angle before composing the optimal photograph.

You don't even have to normalize the length of the vectors after summing the city + state + abbrev vectors, because PCA takes care of that for you.

We saved these "augmented" city word vectors in the `nlpia` package so you can load them to use in your application. In the following code, you use PCA to project them onto a 2D plot.

Listing 6.11. Bubble chart of US cities

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=2)      #1
>>> us_300D = get_data('cities_us_wordvectors')
>>> us_2D = pca.fit_transform(us_300D.iloc[:, :300])    #2
```

Figure 6.8 shows the 2D projection of all these 300-D word vectors for US cities:

Figure 6.8. Google News Word2Vec 300-D vectors projected onto a 2D map using PCA



Note

Low semantic distance (distance values close to zero) represent high similarity between words. The semantic distance, or "meaning" distance, is determined by the words occurring nearby in the documents used for training. The Word2Vec vectors for two terms are *close* to each other in word vector space if they are often used in similar contexts (used with similar words nearby). For example "San Francisco" is *close* to "California" because they often occur nearby in sentences and the distribution of words used near them are similar. A large distance between two terms expresses a low likelihood of shared context and shared meaning (they are semantically dissimilar), such as

"cars" and "peanuts".

If you'd like to explore the city map shown in figure 6.8, or try your hand at plotting some vectors of your own, listing 6.12 shows you how. We built a wrapper for Plotly's offline plotting API that should help it handle DataFrames where you've "denormalized" your data. The Plotly wrapper expects a DataFrame with a row for each sample and column for features you'd like to plot. These can be categorical features (such as time zones) and continuous real-valued features (such as city population). The resulting plots are interactive and useful for exploring many types of machine learning data, especially vector-representations of complex things such as words and documents.

Listing 6.12. Bubble plot of US city word vectors

```
>>> import seaborn
>>> from matplotlib import pyplot as plt
>>> from nlpia.plots import offline_plotly_scatter_bubble
>>> df = get_data('cities_us_wordvectors_pca2_meta')
>>> html = offline_plotly_scatter_bubble(
...     df.sort_values('population', ascending=False)[:350].copy(
...         .sort_values('population'),
...         filename='plotly_scatter_bubble.html',
...         x='x', y='y',
...         size_col='population', text_col='name', category_col='tim
...         xscale=None, yscale=None, # 'log' or None
...         layout={}, marker={'sizeref': 3000}
...         {'sizemode': 'area', 'sizeref': 3000}
```

To produce the 2D representations of your 300-D word vectors, you need to use a dimension reduction technique. We used PCA. To reduce the amount of information lost during the compression from 300-D to 2D, reducing the range of information contained in the input vectors helps. So you limited your word vectors to those associated with cities. This is like limiting the domain or subject matter of a corpus when computing TF-IDF (term frequency - inverse document frequency) or BOW vectors.

For a more diverse mix of vectors with greater information content, you'll probably need a nonlinear embedding algorithm such as t-SNE (t-Distributed Stochastic Neighbor Embedding). We talk about t-SNE and other neural net

techniques in later chapters. t-SNE will make more sense once you've grasped the word vector embedding algorithms here.

6.4.12 Making Connections

In this section, we are going to construct what is known as a *graph*.^[250] The *graph* data structure is ideal for representing relations in data. At the core, a *graph* can be characterized as having *entities* (*nodes* or *vertices*) that are connected together through *relationships* or *edges*. Social networks are great examples of where the *graph* data structure is ideal to store the data. We will be using a particular type of *graph* in this section, an *undirected graph*. This type of *graph* is one where the *relationships* do not have a direction. An example of this non-directed relationship could be a friend connection between two people on Facebook, since neither can be the friend of the other without reciprocation. Another type of *graph* is the *directed graph*. This type of *graph* has relationships that go one way. This type of relationship can be seen in the example of Followers or Following on Twitter. You can follow someone without them following you back, and thus you can have Followers without having to reciprocate the relationship.

To visualize the relationships between ideas and thoughts in this chapter you can create an *undirected graph* with connections (edges) between sentences that have similar meaning. You'll use a force-directed layout engine to push all the similar concepts nodes together into clusters. But first you need some sort of embedding for each sentence. Sentences are designed to contain a single thought How would you use word embeddings to create an embedding for a sentence?

You can apply what you learned about word embeddings from previous sections to create sentence embeddings. You will just average all the embeddings for each word in a sentence to create a single 300-D embedding for each sentence.

Extreme summarization

What does a sentence embedding or thought vector actually contain? That depends on how you create it. You've already seen how to use SpaCy and

nessvec to create word embeddings. You can create sentence embeddings by averaging all the word embeddings for a sentence;

```
from nessvec.files import load_glove
```

Extract natural language from the NLPIA manuscript

The first step is to take this unstructured chapter text and turn it into structured data so that the natural language text can be separated from the code blocks and other "unnatural" text. You can use the Python package called Asciidoc3 to convert any *AsciiDoc* (.adoc) text file into HTML. With the HTML text file, we can use the *BeautifulSoup* to extract the text.

Listing 6.13. HTML Convert AsciiDoc files to HTML with Asciidoc3

```
>>> import subprocess
>>> import os
>>> from bs4 import BeautifulSoup
>>> chapt6_adoc = 'Chapter 06 -- Reasoning with Word Vectors (Wor
>>> chapt6_html = 'Chapter 06 -- Reasoning with Word Vectors (Wor
>>> subprocess.run(args=['asciidoc3', '-a', '-n', '-a', 'icons',
<lineArrow></lineArrow> chapt6_adoc])    #1
>>> if os.path.exists(chapt6_html) and os.path.getsize(chapt6_htm
...     chapter6_html = open(chapt6_html, 'r').read()
...     bsoup = BeautifulSoup(chapter6_html, 'html.parser')
...     text = bsoup.get_text()
```

Now that we have our text from this chapter, we will run the text through the English model from *spaCy* to get our sentence embeddings. *spaCy* will default to simply averaging the *token* vectors. [251] In addition to getting the sentence vectors, we also want to get the *noun phrases* [252] [253] from each sentence that will be the labels for our sentence vectors.

Listing 6.14. Getting Sentence Embeddings and Noun Phrases with spaCy

```
>>> import spacy
>>> nlp = spacy.load('en_core_web_md')
>>> config = {'punct_chars': None}
>>> nlp.add_pipe('sentencizer', config=config)
>>> doc = nlp(text)
>>> sentences = []
```

```

>>> noun_phrases = []
>>> for sent in doc.sents:
...     sent_noun_chunks = list(sent.noun_chunks)
...     if sent_noun_chunks:
...         sentences.append(sent)
...         noun_phrases.append(max(sent_noun_chunks))
>>> sent_vecs = []
>>> for sent in sentences:
...     sent_vecs.append(sent.vector)

```

Now that we have sentence vectors and noun phrases, we are going to normalize (the 2-norm) [\[254\]](#) the sentence vectors. Normalizing the data in the 300-dimensional vector gets all the values on the same scale while still retaining what differentiates them. [\[255\]](#)

Listing 6.15. Normalizing the Sentence Vector Embeddings with NumPy

```

>>> import numpy as np
>>> for i, sent_vec in enumerate(sent_vecs):
...     sent_vecs[i] = sent_vec / np.linalg.norm(sent_vec)

```

With the sentence vectors normalized, we can get the *similarity matrix* (also called an *affinity matrix*). [\[256\]](#)

Listing 6.16. Getting the Similarity/Affinity Matrix

```

>>> np_array_sent_vecs_norm = np.array(sent_vecs)
>>> similarity_matrix = np_array_sent_vecs_norm.dot(np_array_sent_

```

The similarity matrix is calculated by taking the *dot product* between the normalized matrix of sentence embeddings (n by 300 dimensions) with the transpose of itself. This gives an n by n (n = number of sentences in the document) dimensional matrix with the top triangle and the bottom triangle of the matrix being equal. The logic of this is that any vectors pointing in a similar direction will give a weighted sum of their values (dot product) that is close to 1 when they are similar, since the vectors are normalized and all have the same magnitude, but in different directions; think of a sphere in hyper space—a hypersphere with n-dimensions (an *n-sphere*). [\[257\]](#) These weighted sums will be the value of the undirected edges in the graph, and the nodes are the indexes from the similarity matrix. For example: index_i is one node, and

`index_j` is another node (where 'i' represents rows and 'j' represents columns in the matrix).

With the similarity matrix, we can now create an undirected graph with the data. The code below uses a library called Networkx [\[258\]](#) to create the *undirected graph* data structure. This data structure is a dictionary of dictionaries of dictionaries. [\[259\]](#) So the graph is a multi-nested dictionary. The nested dictionaries allow for quick lookups with a sparse data storage.

Listing 6.17. Creating the Undirected Graph

```
>>> import re
>>> import networkx as nx
>>> similarity_matrix = np.triu(similarity_matrix, k=1)    #1
>>> iterator = np.nditer(similarity_matrix, flags=['multi_index'])
>>> node_labels = dict()
>>> G = nx.Graph()
>>> pattern = re.compile(r'[\w\s]*[""]?[\w\s]+[-?][\w\s]*[""]?')
>>> for edge in iterator:
...     key = 0
...     value = ''
...     if edge > 0.95:    #3
...         key = iterator.multi_index[0]
...         value = str(noun_phrases[iterator.multi_index[0]])
...         if pattern.fullmatch(value) and (value.lower()).rstrip():
...             node_labels[key] = value
...         G.add_node(iterator.multi_index[0])
...         G.add_edge(iterator.multi_index[0], iterator.multi_index[1])
```

With the shiny new graph (network) you've assembled, you can now use `matplotlib.pyplot` to visualize it.

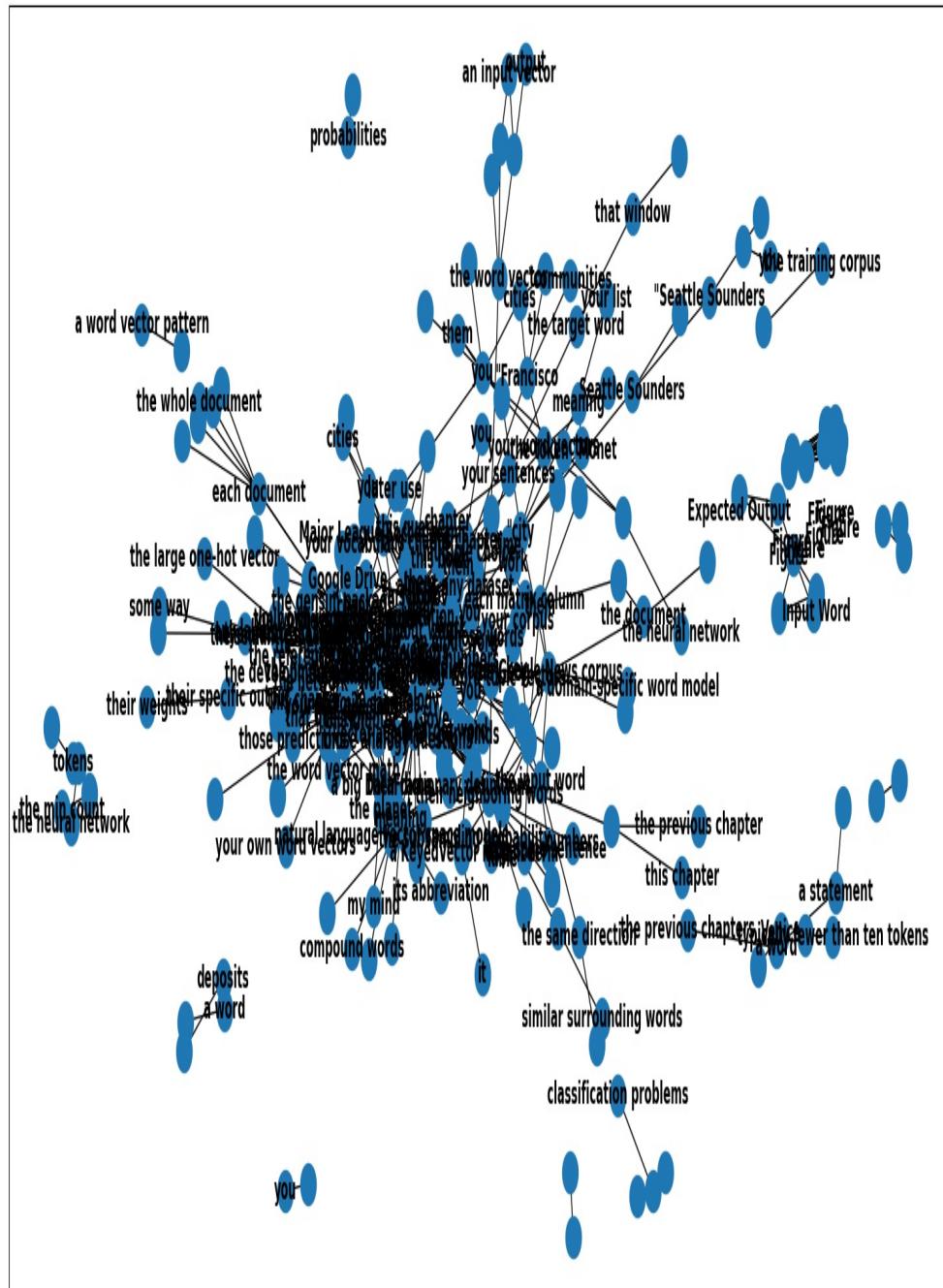
Listing 6.18. Plotting the Undirected Graph

```
>>> import matplotlib.pyplot as plt
>>> plt.subplot(1, 1, 1)    #1
>>> pos = nx.spring_layout(G, k=0.15, seed=42)    #2
>>> nx.draw_networkx(G, pos=pos, with_labels=True, labels=node_labels)
>>> plt.show()
```

Finally you can see your *undirected graph* show the clusters of concepts in the natural language of this book! Each *node* represents the average word

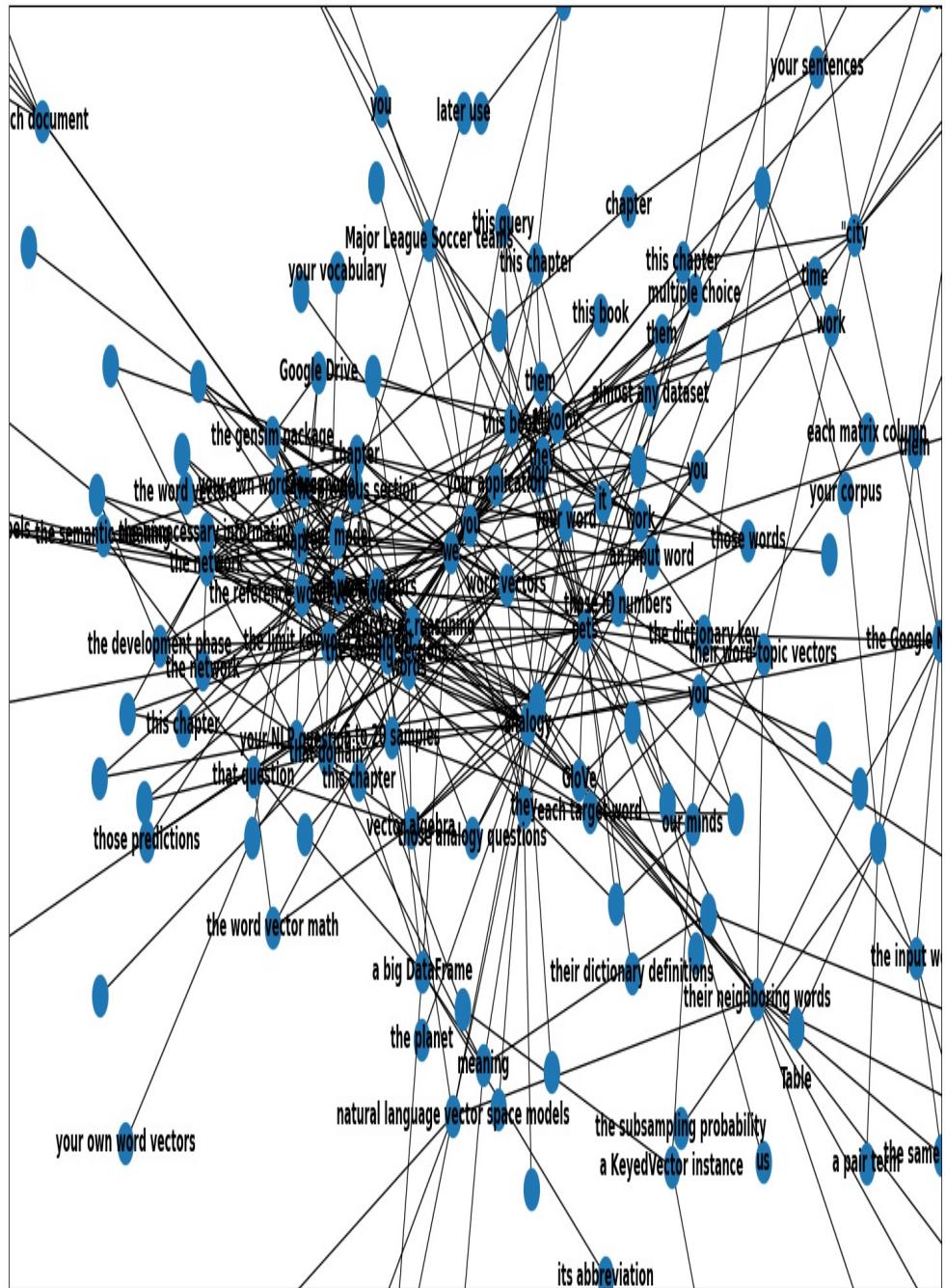
embedding for a sentence in this chapter. And the *edges* (or lines) represent the connections between the meaning of those sentences that mean similar things. Looking at the plot, you can see the central big cluster of nodes (sentences) are connected the most. And there are other smaller clusters of nodes further out from the central cluster for topics such as sports and cities.

Figure 6.9. Connecting concepts to each other with word embeddings



The dense cluster of concepts in the center should contain some information about the central ideas of this chapter and how they are related. Zooming in you can see these passages are mostly about words and numbers to represent words, because that's what this chapter is about.

Figure 6.10. Undirected Graph Plot of Chapter 6 Center Zoom-in



The end of this chapter includes some exercises that you can do to practice what we have covered in this section.

6.4.13 Unnatural words

Word embeddings such as 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 languages other than English.

Embedding works also for pictorial languages such as traditional Chinese and Japanese (Kanji) or the mysterious hieroglyphics 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 *Caesar cipher* [\[260\]](#) such as ROT13 or a *substitution cipher* [\[261\]](#) are both vulnerable to vector-based reasoning with Word2Vec. You don't even need a decoder ring (shown in figure 6.9). You just need a large collection of messages or n -grams that your Word2Vec embedder can process to find co-occurrences of words or symbols.

Figure 6.11. Decoder rings



Word2Vec has even been used to glean information and relationships from unnatural words or ID numbers such as college course numbers (CS-101), model numbers (Koala E7270 or Galaga Pro), and even serial numbers, phone numbers, and zip codes. [262] To get the most useful information about the relationship between ID numbers like this, you'll need a variety of sentences that contain those ID numbers. And if the ID numbers often contain a structure where the position of a symbol has meaning, it can help to tokenize these ID numbers into their smallest semantic packet (such as words or syllables in natural languages).

[219] "Efficient Estimation of Word Representations in Vector Space" Sep 2013, Mikolov, Chen, Corrado, and Dean (<https://arxiv.org/pdf/1301.3781.pdf>).

[220] See the web page titled "Unsupervised Feature Learning and Deep Learning Tutorial" (<http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/>).

[221] See the PDF "Linguistic Regularities in Continuous Space Word Representations" by Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig (<https://www.aclweb.org/anthology/N13-1090>).

[222] See Radim Řehůrek's interview of Tomas Mikolov (https://rare-technologies.com/rrp#episode_1_tomas_mikolov_on_ai).

[223] See "ICRL2013 open review" (<https://openreview.net/forum?id=idpCdOWtqXd60¬eId=C8Vn84fqSG8qa>).

[224] You can find the code for generating these interactive 2D word plots in <http://mng.bz/M5G7>.

[225] See Part III. "Tools for thinking about Meaning or Content" p 59 and chapter 15 "Daddy is a doctor" p. in the book "Intuition Pumps and Other Tools for Thinking" by Daniel C. Dennett

[226] See the web page titled "GitHub - 3Top/word2vec-api: Simple web service providing a word embedding model"
(<https://github.com/3Top/word2vec-api#where-to-get-a-pretrained-model>).

[227] Original Google 300D Word2Vec model on Google Drive
(<https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUlSS21pQmM>)

[228] See the web page titled "GitHub - facebookresearch/fastText: Library for fast text representation and classification."
(<https://github.com/facebookresearch/fastText>).

[229] Wikipedia on Bayard Rustin
(https://en.wikipedia.org/wiki/Bayard_Rustin) a civil right leader and Larry Dane Brimner (https://en.wikipedia.org/wiki/Larry_Dane_Brimner) an author of more than 150 children's books

[230] paperswithcode.com meta study
(<https://paperswithcode.com/methods/category/static-word-embeddings>)

[231] Spelling corrector code and explanation by Peter Norvig
(<https://norvig.com/spell-correct.html>)

[232] The paper "Deep Contextualized Word Representations" paper by the Allen Institute (<https://arxiv.org/pdf/1802.05365v2.pdf>) describes how to

created Embeddings from bidirectional Language Models (ELMo)

[233] See the official ELMo website
(<https://allenai.org/allennlp/software/elmo>)

[234] Sense2Vec - A fast and accurate method for word sense disambiguation in neural network embeddings, Trask et al.:
(<https://arxiv.org/pdf/1511.06388.pdf>)

[235] The publication by the team around Tomas Mikolov (<https://arxiv.org/pdf/1310.4546.pdf>) provides more details.

[236] See the README file at <http://gitlab.com/tangibleai/nlpia2> for installation instructions.

[237] Google hosts the original model trained by Mikolov on Google Drive [here](#).

[238] *Surfaces and Essences: Analogy as the Fuel and Fire of Thinking* by Douglas Hofstadter and Emmanuel Sander.

[239] Detector Morse, by Kyle Gorman and OHSU on pypi and at <https://github.com/cslu-nlp/DetectorMorse>

[240] Explosion.ai blog post (<https://explosion.ai/blog/bloom-embeddings>)

[241] Stanford GloVe Project (<https://nlp.stanford.edu/projects/glove/>).

[242] See chapter 5 and Appendix C for more details on SVD.

[243] *GloVe: Global Vectors for Word Representation* by Jeffrey Pennington, Richard Socher, and Christopher D. Manning:
<https://nlp.stanford.edu/pubs/glove.pdf>

[244] Gensim's comparison of Word2Vec and GloVe performance:
https://rare-technologies.com/making-sense-of-Word2Vec/#glove_vs_word2vec

[245] Enriching Word Vectors with Subword Information, Bojanowski et al.:
<https://arxiv.org/pdf/1607.04606.pdf>

[246] See the web page titled "fastText/pretrained-vectors.md at master"
(<https://github.com/facebookresearch/fastText/blob/main/docs/pretrained-vectors.md>).

[247] SpaCy medium language model docs
(https://spacy.io/models/en#en_core_web_md)

[248] Nessvec source code (<https://gitlab.com/tangibleai/nessvec>) and tutorial videos (<https://proai.org/nessvec-videos>)

[249] Niel Chah's word2vec4everything repository
(<https://github.com/nchah/word2vec4everything>)

[250] See this Wiki page titled, 'Graph (abstract data type)':
[https://en.wikipedia.org/wiki/Graph_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Graph_(abstract_data_type))

[251] spaCy's vector attribute for the Span object defaults to the average of the token vectors: <https://spacy.io/api/span#vector>

[252] See the Wiki page titled, 'Noun phrase':
https://en.wikipedia.org/wiki/Noun_phrase

[253] spaCy's Span.noun_chunks: https://spacy.io/api/span#noun_chunks

[254] See the Wiki page title, 'Norm (mathematics)—Euclidean norm':
[https://en.wikipedia.org/wiki/Norm_\(mathematics\)#Euclidean_norm](https://en.wikipedia.org/wiki/Norm_(mathematics)#Euclidean_norm)

[255] See the web page titled, 'Why Data Normalization is necessary for Machine Learning models': <http://mng.bz/aJ2z>

[256] See this web page titled, 'Affinity Matrix': <https://deepai.org/machine-learning-glossary-and-terms/affinity-matrix>

[257] See the Wiki page titled, 'n-sphere': <https://en.wikipedia.org/wiki/N-sphere>

[258] See the NetworkX web page for more information: <https://networkx.org/>

[259] See the NetworkX documentation for more details:

<https://networkx.org/documentation/stable/reference/introduction.html#data-structure>

[260] See the web page titled "Caesar cipher - Wikipedia" (https://en.wikipedia.org/wiki/Caesar_cipher).

[261] See the web page titled "Substitution cipher - Wikipedia" (https://en.wikipedia.org/wiki/Substitution_cipher).

[262] See the web page titled "A non-NLP application of Word2Vec – Towards Data Science" (<https://medium.com/towards-data-science/a-non-nlp-application-of-word2vec-c637e35d3668>).

6.5 Summary

- Word vectors and vector-oriented reasoning can solve some surprisingly subtle problems like analogy questions and nonsynonymy relationships between words.
- To keep your word vectors current and improve their relevance to the current events and concepts you are interested in you can retrain and fine tune your word embeddings with gensim or PyTorch.
- The nessvec package is a fun new tool for helping you find that word on the tip of your tongue or visualize the "character sheet" of a word.
- Word embeddings can reveal some surprising hidden meanings of the names of people, places, businesses and even occupations
- A PCA projection of word vectors for cities and countries can reveal the cultural closeness of places that are geographically far apart.
- The key to turning latent semantic analysis vectors into more powerful word vectors is to respect sentence boundaries when creating your n -grams
- Machines can easily pass the word analogies section of standardized tests with nothing more than pretrained word embeddings

6.6 Review

1. Use pretrained word embeddings to compute the strength, agility, and intelligence of Dota 2 heroes based only on the natural language summary.^[263]
2. Visualize the graph of connections between concepts in another chapter of this book (or any other text) that you'd like to understand better.
3. Try combining graph visualizations of the word embeddings for all the chapters of this book.
4. Give examples for how word vectors enable at least two of Hofstadter's eight elements of intelligence
5. Fork the nessvec repository and create your own visualizations or nessvector "character sheets" for your favorite words or famous people. Perhaps the "mindfulness", "ethicalness", "kindness", or "impactfulness" of your heroes? Humans are complex and the words used to describe them are multidimensional.

^[263] The nessvec and nlpia2 packages contain FastText, GloVe and Word2vec loaders (<https://gitlab.com/tangibleai/nessvec>). nlpia2 contains ch06_dota2_wiki_heroes.hist.py (https://gitlab.com/tangibleai/nlpia2/-/blob/main/src/nlpia2/etl/ch06_dota2_wiki_heroes.hist.py) dota2-heroes.csv (<https://gitlab.com/tangibleai/nlpia2/-/raw/main/.nlpia2-data/dota2-heroes.csv?inline=false>)

7 Finding kernels of knowledge in text with Convolutional Neural Networks (CNNs)

This chapter covers

- Understanding neural networks for NLP
- Finding patterns in sequences
- Building a CNN with PyTorch
- Training a CNN
- Training embeddings
- Classifying text

In this chapter you will unlock the misunderstood superpowers of convolution for Natural Language Processing. This will help your machine understand words by detecting patterns in sequences of words and how they are related to their neighbors.

Convolutional Neural Networks (CNNs) are all the rage for *computer vision* (image processing). But few businesses appreciate the power of CNNs for NLP. This creates an opportunity for you in your NLP learning and for entrepreneurs that understand what CNNs can do. For example, in 2022 Cole Howard and Hannes Hapke (coauthors on the first edition of this book) used their NLP CNN expertise to help their startup automate business and accounting decisions. [\[264\]](#) And deep learning deep thinkers in academia like Christopher Manning and Geoffrey Hinton use CNNs to crush the competition in NLP. You can too.

So why haven't CNNs caught on with industry and big tech corporations? Because they are too good—too efficient. CNNs don't need the massive amounts of data and compute resources that is central to Big Tech's monopoly power in AI. They are interested in models that "scale" to huge datasets, like reading the entire Internet. Researchers with access to big data

focus on problems and models that leverage their competitive advantage with data, "the new oil."[\[265\]](#) It's hard to charge people much money for a model anyone can train and run on their own laptop.

Another more mundane reason CNNs are overlooked is that properly configured and tuned CNNs for NLP are hard to find. I wasn't able to find a single reference implementation of CNNs for NLP in the PyTorch, Keras, or Tensorflow. And the unofficial implementations seemed to transpose the CNN channels used for image processing to creation convolutions in embedding dimensions rather than convolution in time. You'll soon see why that is a bad idea. But don't worry, you'll soon see the mistakes that others have made and you'll be building CNNs like a pro. Your CNNs will be more efficient and performant than anything coming out of the blogosphere.

Perhaps you're asking yourself why should you learn about CNNs when the shiny new thing in NLP, *transformers*, are all the rage. You've probably heard of *GPT-J*, *GPT-Neo*, *PaLM* and others. After reading this chapter you'll be able to build better, faster, cheaper NLP models based on CNNs while everyone else is wasting time and money on Giga-parameter transformers. You won't need the unaffordable compute and training data that large transformers require.[\[266\]](#) [\[267\]](#) [\[268\]](#)

- **PaLM:** 540B parameters
- **GPT-3:** 175B parameters
- **T5-11B:** 11B parameters (FOSS, outperforms GPT-3)
- **GPT-J:** 6B parameters (FOSS, outperforms GPT-3)
- **CNNs** (in this Chapter): less than 200k parameters

Yes, in this chapter you're going to learn how to build CNNs models that are a million times smaller and faster than the big transformers you read about in the news. And CNNs are often the best tool for the job.

7.1 Patterns in sequences of words

Individual words worked well for you in the previous chapters. You can say a lot with individual words. You just need to chose the right words or find the keywords in a passage of text and that can usually capture the meaning. And

the order doesn't matter too much for the kinds of problems you solved in previous chapters. If you throw all the words from a job title such as "Junior Engineer" or "Data Scientist" into a bag of words (BOW) vector, the jumbled up BOW contains most of the information content of the original title. That's why all the previous examples in this book worked best on short phrases or individual words. That's why keywords are usually enough to learn the most important facts about a job title or get the gist of a movie title. And that's why it's so hard to choose just a few words to summarize a book or job with its title. For short phrases, the occurrence of words is all that matters.

When you want to express a complete thought, more than just a title, you have to use longer sequences of words. And the order matters.

Before NLP, and even before computers, humans have used a mathematical operation called *convolution* to detect patterns in sequences. For NLP, convolution is used to detect patterns that span multiple words and even multiple sentences. The original convolutions were hand crafted on paper with a quill pen, or even a cuneiform on a clay tablet! Once computers were invented, researchers and mathematicians would hand-craft the math to match what they wanted to achieve for each problem. Common hand-crafted kernels for image processing include Laplacian, Sobel, and Gaussian filters. And in digital signal processing similar to what is used in NLP, low pass and high pass convolution filters can be designed from first principles. If you're a visual learner or are into computer vision it might help you grasp convolution if you check out heatmap plots of the kernels used for these convolutional filters on Wikipedia. [\[269\]](#) [\[270\]](#) [\[271\]](#) These filters might even give you ideas for initializations of your CNN filter weights to speed learning and create more explainable deep learning language models.

But that gets tedious after a while, and we no longer even consider hand-crafted filters to be important in computer vision or NLP. Instead, we use statistics and neural networks to automatically *learn* what patterns to look for in images and text. Researchers started with linear fully connected networks (multi-layer perceptrons). But these had a real problem with over-generalization and couldn't recognize when a pattern of words moved from the beginning to the end of the sentence. Fully-connect neural networks are not scale and translation invariant should look like to do their job. But then

David Rumelhart invented and Geoffrey Hinton popularized the backpropagation approach that helped CNNs and deep learning bring the world out of a long AI winter.^[272] ^[273] This approach birthed the first practical CNNs for computer vision, time series forecasting and NLP.

Figuring out how to combine convolution with neural networks to create CNNs was just the boost neural networks needed. CNNs now dominate computer vision. And for NLP, CNNs are still the most efficient models for many advanced natural language processing problems. For example, spaCy switched to CNNs for version 2.0. CNNs work great for *named entity recognition* (NER) and other word tagging problems.^[274] And CNNs in your brain seem to be responsible for your ability to recognize language patterns that are too complex for other animals.

^[264] Digits technology description (<https://digits.com/technology>)

^[265] Wired Magazine popularized the concept of data as the new oil in a 2014 article by that title (<https://www.wired.com/insights/2014/07/data-new-oil-digital-economy/>)

^[266] Google AI blog post on Pathways Language Model, or PaLM, (<https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html>)

^[267] "How you can use GPT-J" by Vincent Meuller (<https://towardsdatascience.com/how-you-can-use-gpt-j-9c4299dd8526>)

^[268] "T5 - A Detailed Explanation" by Qiurui Chen (<https://medium.com/analytics-vidhya/t5-a-detailed-explanation-a0ac9bc53e51>)

^[269] "Digital image processing" on Wikipedia (https://en.wikipedia.org/wiki/Digital_image_processing#Filtering)

^[270] "Sobel filter" on Wikipedia (https://en.wikipedia.org/wiki/Sobel_operator)

[271] "Gaussian filter" (https://en.wikipedia.org/wiki/Gaussian_filter)

[272] May 2015, *nature*, "Deep Learning" by Hinton, LeCunn, and Benjio (<https://www.nature.com/articles/nature14539>)

[273] "A Brief History of Neural Nets and Deep Learning" by Andrey Kurenkov (<https://www.skynettoday.com/overviews/neural-net-history>)

[274] SpaCy NER documentation (<https://spacy.io/universe/project/video-spacys-ner-model>)

7.2 Scale and Translation Invariance

The main advantage of CNNs over previous NLP algorithms is that they can recognize patterns in text no matter where those patterns occur in the text (*translation invariance*) and how spread out they are (*scale invariance*). TF-IDF vectors don't have any way of recognizing and generalizing from patterns in your text. And fully connected neural networks over-generalize from particular patterns at particular locations in text.

As far back as the 1990s famous researchers like Yann LeCun, Yoshua Bengio, and Geoffrey Hinton were using convolution for computer vision and OCR (optical character recognition). [275] They got this idea from our brains. Neural networks are often referred to as "neuromorphic" computing because they mimic or simulate what happens in our brains. Neural networks simulate in software what brains (networks of biological neurons) do in wetware. And because CNNs are based on brains, they can be used for all kinds of "off-label" NLP applications: voice, audio, text, weather, and time series. NLP CNNs are useful for any series of symbols or numerical vectors (embeddings). This intuition empowers you to apply your NLP CNNs to a wide variety of problems that you will run into at your job - such as financial time series forecasting and weather forecasting.

Convolutions in our brains give us the ability to process and understand verbal communication in a *scale invariant* and *translation invariant* way. The scale invariance of convolution means you can understand others even if they stretch out the patterns in their words over a long time by speaking slowly or

adding a lot of filler words. And translation invariance means you can understand peoples' intent whether they lead with the good news or the bad news. You've probably gotten pretty good at handling feedback from your parents, teachers, and bosses whether it is authentic constructive criticism or even if the "meat" is hidden inside a "praise sandwich."^[276] Perhaps because of the subtle ways we use language and how important it is in culture and memory, convolution is built into our brains. We are the only species to have convolution networks built into our brains. And some people have as many as 3 layers of convolutions happening within the part of the brain that processes voice, called "Heschl's gyrus" (HG).^[277]

You'll soon see how to incorporate the power of translation and scale invariant convolutional filters into your own neural networks. You will use CNNs to classify questions and toots (Mastodon ^[278] posts) and even the beeps and boops of Morse code. Your machine will soon be able to tell whether a question is about a person, thing, historical date, or general concept. You'll even try to see if a question classifier can tell if someone is asking you out on a date. And you might be surprised to learn that CNNs can detect subtle differences between catastrophes you might read about online: catastrophic birdsite post vs a real world disaster.

^[275] LeCun, Y and Bengio, Y "Convolutional Networks for Images, Speech, and Time-series" (<https://www.iro.umontreal.ca/~lisa/poiteurs/handbook-convo.pdf>)

^[276] Sometimes "feedback sandwich" or "sh-t sandwich."

^[277] "An anatomical and functional topography of human auditory cortical areas" by Michelle Moerel et al (<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4114190/>)

^[278] Mastodon is a community-owned, ad-free social network: <https://joinmastodon.org/>

7.3 Convolution

The concept of *convolution* is not as complicated as it sounds. The math is

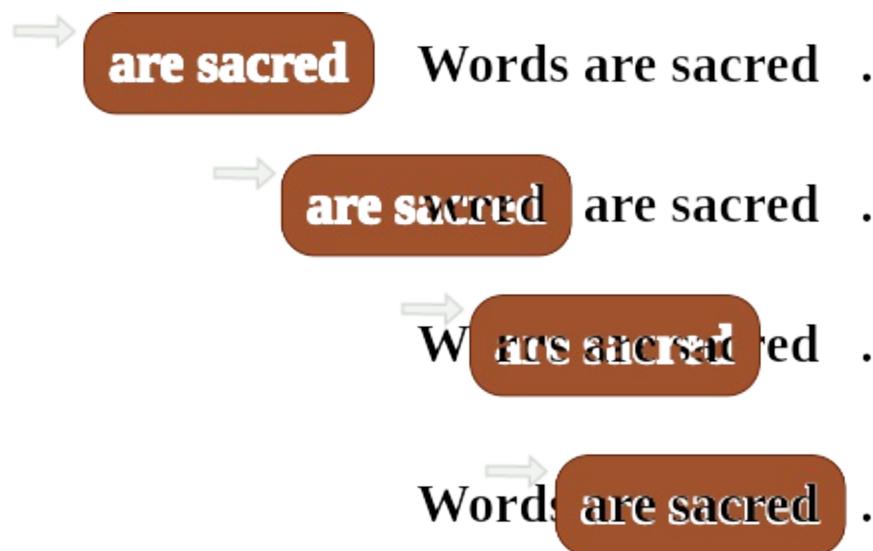
almost the same as for calculating the correlation coefficient. Correlation helps you measure the covariance or similarity between a pattern and a signal. In fact its purpose is the same as for correlation, pattern recognition. Correlation allows you to detecting the similarity between a series of numbers and some other series of numbers representing the pattern you're looking to match.

7.3.1 Stencils for natural language text

Have you ever seen a lettering stencil? A lettering stencil is a piece of cardboard or plastic with the outline of printed letters cut out. When you want paint words onto something, such as a storefront sign, or window display, you can use a stencil to make your sign come out looking just like printed text. You use a stencil like movable masking tape to keep you from painting in the wrong places. But in this example you're going to use the stencil in reverse. Instead of painting words with your stencil, you're going to detect patterns of letters and words with a stencil. Your NLP stencil is an array of weights (floating point numbers) called a *filter* or *kernel*.

So imagine you create a lettering stencil for the nine letters (and one *space* character) in the text "are sacred". And imagine it was exactly the size and shape of the text in this book that you are reading right now.

Figure 7.1. A real-life stencil



Now, in your mind, set the stencil down on top of the book so that it covers the page and you can only see the words that "fit" into the stencil cutout. You have to slide that stencil across the page until the stencil lines up with this pair of words in the book. At that point you'd be able to see the words spelled out clearly through the stencil or mask. The black lettering of the text would fill the holes in the stencil. And the amount of black that you see is a measure of how good the match is. If you used a white stencil, the words "are sacred" would shine through and would be the only words you could see.

If you used a stencil this way, sliding it across text to find the maximum match between your pattern and a piece of text, you'd be doing *convolution* with a stencil! When talking about deep learning and CNNs the stencil is called a *kernel* or *filter*. In CNNs the *kernel* is an array of floating point numbers rather than a cardboard cutout. And the kernel is designed to match a general pattern in the text. Your text has also been converted to an array of numerical values. And convolution is process of sliding that kernel across your numerical representation of text to see what pops out.

Just a decade or so ago, before CNNs, you would have had to hand-craft your kernels to match whatever patterns you could dream up. But with CNNs you don't have to program the kernels at all, except to decide how wide the kernels are - how many letters or words you think will capture the patterns you need. Your CNN optimizer will fill in the weights within your kernel. As you train a model, the optimizer will find the best array of weights that matches the patterns that are most predictive of the target variable in your NLP problem. The back propagation algorithm will incrementally adjust the weights bit by bit until they match the right patterns for your data.

You need to add a few more steps to your mental model of stencils and kernels to give you a complete understanding of how CNNs work. A CNN needs to do 3 things with a kernel (stencil) to incorporate it into a natural language processing pipeline.

1. Measure the amount of match or similarity between the kernel and the text
2. Find the maximum value of the kernel match as it slides across some text
3. Convert the maximum value to a binary value or probability using an

activation function

You can think of the amount of blackness that pops through your stencil as a measure of the amount of match between your stencil and the text. So step 1 for a CNN, is to multiply the weights in your kernel by the numerical values for a piece of text and adding up all those products to create a total match score. This is just the dot product or correlation between the kernel and that particular window of text.

Step 2 is to slide your window across the text and do the dot product of step 1 again. This kernel window sliding, multiplying, and summing is called convolution. Convolutions turn one sequence of numbers into another sequence of numbers that's about the same size as the original text sequence. Depending on the details of how you do this sliding and multiplying (convolution) you can end up with a slightly shorter or longer sequence of numbers. But either way, the convolution operation outputs a sequence of numerical values, one for every possible position of the kernel in your text.

Step 3 is to decide whether the text contains a good match somewhere within it. For this your CNN converts the sequence of values output by convolution into a single value. The end result is a single value representing the likelihood that the kernel's pattern was somewhere in the text. Most CNNs are designed to take the maximum value of this sequence of numbers as a measure of a match. This approach called *max pooling* because it collects or pools all of the values from the convolution into a single maximum value.



Note

If the patterns that you are looking for are spread out over multiple different locations within a passage of text, then you may want to try *mean pooling* for some of your kernels.

You can see how convolution enables your CNN to extract patterns that depend on the order of words. And this allows CNN kernels to recognize subtleties in the meaning of natural language text that are lost if you only use BOW (bag-of-words) representations of text.

Words are sacred. If you get the right ones in the right order you can nudge the world a little.

-- Tom Stoppard *The Real Thing*

In the first few chapters you treated words as sacred by learning how best to tokenize text into words and then compute vector representations of individual words. Now you can combine that skill with convolution to give you the power to "nudge the world a little" with your next chatbot on Mastodon.[\[279\]](#)

7.3.2 A bit more stenciling

Remember the lettering stencil analogy? Reverse lettering stencils would not be all that useful for NLP because cardboard cutouts can only match the "shape" of words. You want to match the meaning and grammar of how words are used in a sentence. So how can you upgrade your reverse stencil concept to make it more like what you need for NLP? Suppose you want your stencil to detect (adjective, noun) 2-grams, such as "right word" and "right order" in the quote by Tom Stoppard. Here's how you can label the words in a portion of the quote with their parts of speech.

```
>>> import pandas as pd
>>> import spacy
>>> nlp = spacy.load('en_core_web_md')      #1

>>> text = 'right ones in the right order you can nudge the world
>>> doc = nlp(text)
>>> df = pd.DataFrame([
...     {k: getattr(t, k) for k in 'text pos_'.split()}
...     for t in doc])

    text  pos_
0    right   ADJ
1    ones    NOUN
2      in    ADP
3      the    DET
4    right   ADJ
5   order    NOUN
6     you   PRON
7     can    AUX
8   nudge   VERB
```

```

9      the   DET
10    world  NOUN

```

Just as you learned in chapter 6 you want to create a vector representation of each word so that the text can be converted to numbers for use in the CNN.

```
>>> pd.get_dummies(df, columns=['pos_'], prefix='', prefix_sep=''
```

| | text | ADJ | ADP | AUX | DET | NOUN | PRON | VERB |
|----|-------|-----|-----|-----|-----|------|------|------|
| 0 | right | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | ones | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | in | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | the | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 | right | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | order | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | you | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 7 | can | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 8 | nudge | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 9 | the | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 | world | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Now your stencil or kernel will have to be expanded a bit to span two of the 7-D one-hot vectors. You will create imaginary cutouts for the 1's in the one-hot encoded vectors so that the pattern of holes matches up with the sequence of parts of speech you want to match. Your adjective-noun stencil has holes in the first row and the first column the adjective at the beginning of a 2-gram. You will need a hole in the second row and fifth column for the noun as the second word in the 2-gram. As you slide your imaginary stencil over each pair of words it will output a boolean `True` or `False` depending on whether the stencil matches the text or not.

The first pair of words will create a match:

```
0, 1  (right, ones)  (ADJ, NOUN)  _True_
```

Moving the stencil to cover the second 2-gram, it will output `False` because the two gram starts with a noun and ends with a fails to beep

```
1, 2  (ones, in)  (NOUN, ADP)  False
```

Continuing with the remaining words we end up with this 9-element map for the 10-word phrase.

| Span | Pair | Is match? |
|-------|----------------|-----------------|
| 0, 1 | (right, ones) | True (1) |
| 1, 2 | (ones, in) | False (0) |
| 2, 3 | (in, the) | False (0) |
| 3, 4 | (the, right) | False (0) |
| 4, 5 | (right, order) | True (1) |
| 5, 6 | (order, you) | False (0) |
| 6, 7 | (you, can) | False (0) |
| 7, 8 | (can, nudge) | False (0) |
| 8, 9 | (nudge, the) | False (0) |
| 9, 10 | (the, world) | False (0) |

Congratulations. What you just did was convolution. You transformed smaller chunks of an input text, in this case 2-grams, to reveal where there was a match for the pattern you were looking for. It's usually helpful to add

padding to your token sequences. And to clip your text at a maximum length. This ensures that your output sequence is always the same length, no matter how long your text is your kernel.

Convolution, then, is

- a transformation...
- of input that may have been padded...
- to produce a map...
- of where in the input certain conditions existed (e.g. two consecutive adverbs)

Later in the chapter you will use the terms *kernel* and *stride* to talk about your stencil and how you slide it across the text. In this case your *stride* was one and the kernel size was two. And for the part-of-speech vectors, your kernel was designed to handle 7-D embedding vectors. Had you used the same kernel size of two but stepped it across the text with a stride of two, then you would get the following output:

| Span | Pair | Is match? |
|------|----------------|-----------------|
| 0, 1 | (right, ones) | True (1) |
| 2, 3 | (in, the) | False (0) |
| 4, 5 | (right, order) | True (1) |
| 6, 7 | (you, can) | False (0) |
| 8, 9 | (nudge, the) | False (0) |

In this case you got lucky with your stride because the two adjective-noun pairs were an even number of words apart. So your kernel successfully detected both matches for your pattern. But you would only get luck 50% of the time with this configuration. So it is much more common to have a stride of one and kernel sizes of two or more.

7.3.3 Correlation vs. convolution

In case you've forgotten, listing 7.1 should remind you what correlation looks like in Python. (You can also use `scipy.stats.pearsonr`).

Listing 7.1. Python implementation of correlation

```
>>> def corr(a, b):
...     """ Compute the Pearson correlation coefficient R """
...     a = a - np.mean(a)
...     b = b - np.mean(b)
...     return sum(a * b) / np.sqrt(sum(a*a) * sum(b*b))

... a = np.array([0, 1, 2, 0, 1, 2, 0, 1, 2])
... b = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])

print(corr(a, b))
0.31622776601683794

print(corr(a, a))
1.0
```

However, correlation only works when the series are the same length. And you definitely want to create some math that can work with patterns that are shorter than the sequence of numbers representing your text. That's how mathematicians came up with the concept of convolution. They split the longer sequence into smaller ones that are the same length as the shorter one and then apply the correlation function to each of these pairs of sequences. That way convolution can work for any 2 sequences of numbers no matter how long or short they are. So in NLP we can make our pattern (called a *kernel*) as short as we need to. And the series of tokens (text) can be as long as you like. You compute correlation over a sliding window of text to create a sequence of correlation coefficients that represent the meaning of the text.

7.3.4 Convolution as a mapping function

CNNs (in our brains and in machines) are the "mapping" in a map-reduce algorithm. It outputs a new sequence that is shorter than the original sequence, but not short enough. That will come later with the *reduce* part of the pipeline. Pay attention to the size of the outputs of each convolutional layer.

The math of convolution allows you to detect patterns in text no matter where (or when) they occur in that text. We call an NLP algorithm "time invariant" if it produces feature vectors that are the same no matter where (when) a particular pattern of words occurs. Convolution is a time-invariant operation, so it's perfect for text classification and sentiment analysis and NLU. Time invariance is a big advantage of convolution over other approaches you've used so far. Your CNN output vector gives you a consistent representation of the thought expressed by a piece of text no matter where in the text that thought is expressed. Unlike word embedding representations, convolution will pay attention to the meaning of the order of the vectors and won't smush them all together into a pointless average.

Another advantage of convolution is that it outputs a vector representation of your text that is the same size no matter how long your text is. Whether your text is a one-word name or a ten thousand word document, a convolution across that sequence of tokens would output the same size vector to represent the meaning of that text. Convolution creates embedding vectors that you can use to make all sorts of predictions with, just like you did with word embeddings in chapter 6. But now these embeddings will work on sequences of words, not just individual words. Your embedding, your vector representation of meaning, will be the same size no matter whether the text you're processing is the three words "I love you" or much longer: "I feel profound and compersive love for you." The feeling or sentiment of love will end up in the same place in both vectors despite the word love occurring at different locations in the text. And the meaning of the text is spread over the entire vector creating what is called a "dense" vector representation of meaning. When you use convolution, there are no gaps in your vector representation for text. Unlike the sparse TF-IDF vectors of earlier chapters, the dimensions of your convolution output vectors are all packed meaning for

every single bit of text you process.

7.3.5 Python convolution example

You're going to start with a pure python implementation of convolution. This will give you a mental model of the math for convolution, and most importantly, of the shapes of the matrices and vectors for convolution. And it will help you appreciate the purpose of each layer in a convolutional neural network. For this first convolution you will hard-code the weights in the convolution kernel to compute a 2-point moving average. This might be useful if you want to extract some machine learning features from daily cryptocurrency prices in Robinhood. Or perhaps it would be better to imagine you trying to solve a solvable problem like doing feature engineering of some 2-day averages on the reports of rainfall for a rainy city like Portland, Oregon. Or even better yet, imagine you are trying to build a detector that detects a dip in the part-of-speech tag for an adverb in natural language text. Because this is a hard-coded kernel, you won't have to worry about training or fitting your convolution to data just yet.

You are going to hard-code this convolution to detect a pattern in a sequence of numbers just like you hard-coded a regular expression to recognize tokens in a sequence of characters in Chapter 2. When you hard-code a convolutional filter, you have to know what patterns you're looking for so you can put that pattern into the coefficients of your convolution. This works well for easy-to-spot patterns like dips in a value or brief spikes upward in a value. These are the kinds of patterns you'll be looking for in Morse code "text" later in this chapter. In section 3 of this chapter you will learn how to build on this skill to create a convolutional neural network in PyTorch that can *learn* on its own which patterns to look for in your text.

In computer vision and image processing you would need to use a 2-D convolutional filter so you can detect both vertical and horizontal patterns, and everything in-between. For natural language processing you only need 1-dimensional convolutional filters. You're only doing convolution in one dimension, the time dimension, the position in your sequence of tokens. You can store the components of your embedding vectors, or perhaps other parts of speech, in channels of a convolution. More on that later, once you're done

with the pure Python convolution. Here's the Python for perhaps the simplest possible useful 1-D convolution.

Listing 7.4 shows you how to create a 1-D convolution in pure python for a hard-coded kernel ([.5, .5]) with only two weights of .5 in it.

This kernel is computing the rolling or moving average of two numbers in a sequence of numbers. For natural language processing, the numbers in the input sequence represent the occurrence (presence or absence) of a token in your vocabulary. And your token can be anything, like the part-of-speech tag that we used to mark the presence or absence (occurrence) of adverbs in listing. Or the input could be the fluctuating numerical values of a dimension in your word embeddings for each token.

This moving average filter can detect the occurrence of two things in a row because (.5 * 1 + .5 * 1) is 1. A 1 is how your code tells you it has found something. Convolution is great at detecting *patterns* like this that other NLP algorithms would miss. Rather than looking for two occurrences a word, you are going to look for two aspects of meaning in a row. And you've just learned all about the different aspects of meaning in the last chapter, the dimensions of word vectors. For now you're just looking for a single aspect of words, their part of speech. You are looking for one particular part of speech, adverbs. You're looking for two adverbs in a row.

The right word may be effective, but no word was ever as effective as a rightly timed pause.

-- Mark Twain

Can you spot the two adverbs in a row? I had to cheat and use SpaCy in order to find this example. Subtle patterns of meaning like this are very hard for a human to consciously notice. But measuring the the *adverbiness* of text is just a matter of math for a convolutional filter. And convolution will work in parallel for all the other aspects of meaning that you might be looking for. In fact, once you're done with this first example, you will run convolution on *all* of the dimensions of words. Convolution works best when you use the word embeddings from the previous chapter that keep track of all the dimensions of words in vectors.

Not only will convolution look at all the dimensions of meaning in words but also all the *patterns* of meaning in all those dimensions of words. A convolutional neural network (CNN) looks at your desired output (target variable) to find all the patterns in all dimensions of word embeddings that influence your target variable. For this example you're defining an adverb sentence as one that contains two adverbs consecutively within a sentence. This is just to help you see the math for a very simple problem. Adverbiness is just one of many features you need to engineer from text in machine learning pipelines. A CNN will automate that engineering for you by learning just the right combination of adverbiness, nounness, stopwordness, and lots of other nesses. For now you'll just do it all by hand for this one adverbiness feature. The goal is to understand the kinds of patterns a CNN can learn to recognize in your data.

Listing 7.2 shows how to tag the quote with parts of speech tags using SpaCy and then create a binary series to represent the one aspect of the words you are searching for, adverbiness.

Listing 7.2. Tag a quote with parts of speech

```
>>> nlp = spacy.load('en_core_web_md')
>>>
>>> quote = "The right word may be effective, but no word was eve
...     " as effective as a rightly timed pause."
>>> tagged_words = {
...     t.text: [t.pos_, int(t.pos_ == 'ADV')]      #1
...     for t in nlp(quote)}
>>>
>>> df_quote = pd.DataFrame(tagged_words, index=['POS', 'ADV'])
>>> print(df_quote)

          The   right   word   may   be   ...   a   rightly   timed   pause
POS      DET       ADJ      NOUN    AUX    AUX    ...    DET       ADV      VERB      NOUN      PUN
ADV        0        0        0        0        0    ...        0        1        0        0        0
```

Now you have your sequence of ADV ones and zeros so you can process it with convolution to match the pattern you're looking for.

Listing 7.3. Define your input sequence for convolution

```
>>> inpt = list(df_quote.loc['ADV'])
```

```
>>> print(inpt)
[0, 0, 0, ... 0, 1, 1, 0, 0...]
```

Wow, this cheating worked too well! We can clearly see there are two adverbs in a row somewhere in the sentence. Let's use our convolution filter to find where exactly.

Listing 7.4. Convolution in pure python

```
>>> kernel = [.5, .5]                                #1
>>>
>>> output = []
>>> for i in range(len(inpt) - 1):                  #2
...     z = 0
...     for k, weight in enumerate(kernel):            #3
...         z = z + weight * inpt[i + k]
...     output.append(z)
>>>
>>> print(f'inpt:\n{inpt}')
>>> print(f'len(inpt): {len(inpt)}')
>>> print(f'output:\n{[int(o) if int(o)==o else o for o in output}')
>>> print(f'len(output): {len(output)}')

inpt:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0., 1, 1., 0, 0, 0., 1., 0, 0, 0]
len(inpt): 20
output:
[0, 0, 0, 0, 0, 0, 0, 0, 0, .5, 1, .5, 0, 0, .5, .5, 0, 0]
len(output): 19
```

You can see now why you had to stop the `for` loop 1 short of the end of the input sequence. Otherwise our kernel with 2 weights in it would have overflowed off the end of the input sequence. You may have seen this kind of software pattern called "map reduce" elsewhere. And you can see how you might to use the Python built-in functions `map()` and `filter()` to implement the code in listing 7.4.

You can create a moving average convolution that computes the adverbiness of a text according to our 2-consecutive-adverb definition if you use the `sum` function as your *pooling* function. If you want it to compute an unweighted moving average you then just have to make sure your kernel values are all `1 / len(kernel)` so that they sum to 1 and are all equal.

Listing 7.5 will create a line plot to help you visualize the convolution output and the original `is_adv` input on top of each other.

Listing 7.5. Line plot of input (`is_adv`) and output (adverbiness)

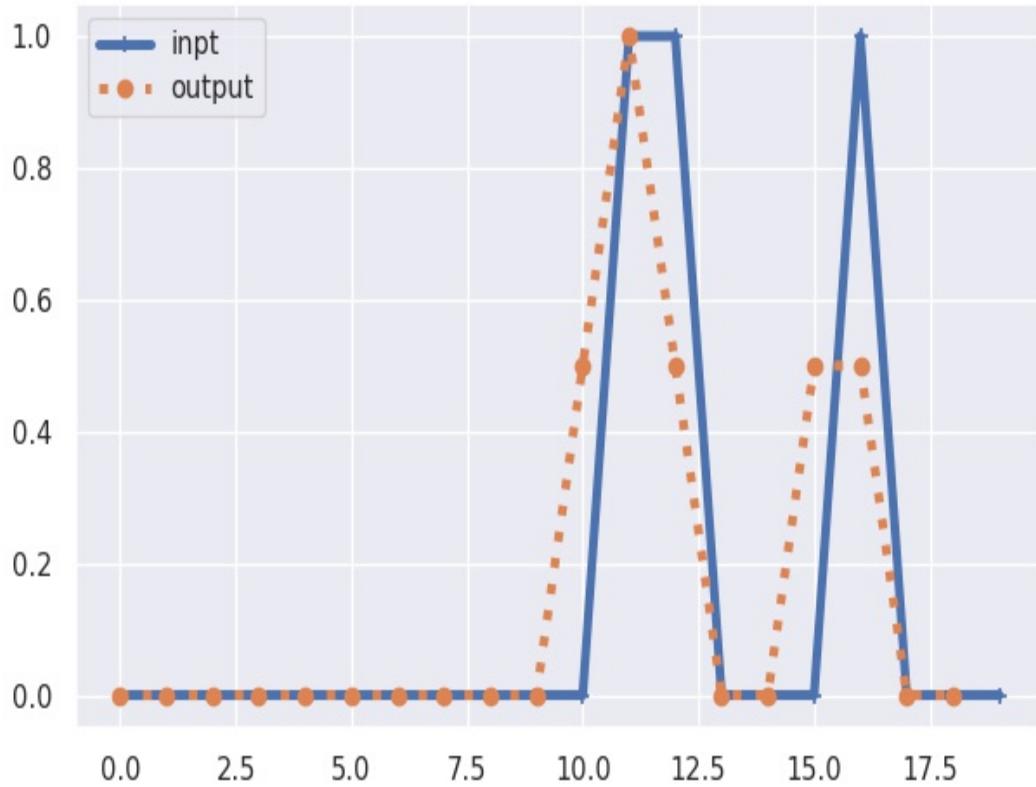
```
>>> import pandas as pd
>>> from matplotlib import pyplot as plt
>>> plt.rcParams['figure.dpi'] = 120      #1

>>> import seaborn as sns
>>> sns.set_theme('paper')      #2

>>> df = pd.DataFrame([inpt, output], index=['inpt', 'output']).T
>>> ax = df.plot(style=['+-', 'o:'], linewidth=3)
```

Did you notice how the output sequence for this convolution by a size 2 kernel produced output that was one shorter than the input sequence? Figure 7.2 shows a line plot of the input and output of this moving average convolution. When you multiply two numbers by .5 and add them together, you get the average of those two numbers. So this particular kernel ([.5, .5]) is a very small (two-sample) moving average filter.

Figure 7.2. Line plot of `is_adv` and adverbiness convolution



Looking at figure 7.2 you might notice that it looks a bit like the moving average or smoothing filters for financial time series data or daily rainfall values. For a 7-day moving average of your GreenPill token prices, you would use a size 7 convolution kernel with values of one seventh (0.142) for each day of the week. [\[280\]](#) A size 7 moving average convolution would just smooth your spikes in adverbiness even more, creating a much more curved signal in your line plots. But you'd never achieve a 1.0 adverbiness score on any organic quotes unless you carefully crafted a statement yourself that contained seven adverbs in a row.

You can generalize your python script in listing 7.6 to create a convolution function that will work even when the size of the kernel changes. This way you can reuse it in later examples.

Listing 7.6. Generalized convolution function

```

>>> def convolve(inpt, kernel):
...     output = []
...     for i in range(len(inpt) - len(kernel) + 1):      #1
...         output.append(
...             sum(
...                 [
...                     inpt[i + k] * kernel[k]
...                     for k in range(len(kernel))           #2
...                 ]
...             )
...         )
...     return output

```

The `convolve()` function you created here sums the input multiplied by the kernel weights. You could also use the Python `map()` function to create a convolution. And you used the Python `sum()` function to *reduce* the amount of data in your output. This combination makes the convolution algorithm a *map reduce* operation that you may have heard of in your computer science or data science courses.



Important

Map-reduce operations such as convolution are highly parallelizable. Each of the kernel multiplications by a window of data could be done simultaneously in parallel. This parallelizability is what makes convolution such a powerful, efficient, and successful way to process natural language data.

7.3.6 PyTorch 1-D CNN on 4-D embedding vectors

You can see how 1-D convolution is used to find simple patterns in a sequence of tokens. In previous chapters you used regular expressions to find patterns in a 1-D sequence of characters. But what about more complex patterns in grammar that involve multiple different aspects of the meaning of words? For that you will need to use word embeddings (from chapter 6) combined with a *convolutional neural network*. You want to use PyTorch to take care of all the bookkeeping of all these linear algebra operations. You'll keep it simple with this next example by using 4-D one-hot encoded vectors for the parts of speech of words. Later you'll learn how to use 300-D GloVe vectors that keep track of the meaning of words in addition to their

grammatical role.

Because word embeddings or vectors capture all the different components of meaning in words, they include parts of speech. Just as in the adverb quote example earlier, you will match a grammatical pattern based on the parts of speech of words. But this time your words will have a 3-D part of speech vector representing the parts of speech noun, verb, and adverb. And your new CNN can detect a very specific pattern, an adverb followed by a verb then a noun. Your CNN is looking for the "rightly timed pause" in the Mark Twain quote. Refer back to Listing 7.2 if you need help creating a DataFrame containing the POS tags for the "rightly timed pause" quote.

```
>>> tags = 'ADV ADJ VERB NOUN'.split()
>>> tagged_words = [
...     [tok.text] + [int(tok.pos_ == tag) for tag in tags]      #1
...     for tok in nlp(quote)]
>>>
>>> df = pd.DataFrame(tagged_words, columns=['token'] + tags).T
>>> print(df)
```

| | The | right | word | may | be | ... | a | rightly | timed | pause | . |
|------|-----|-------|------|-----|----|-----|---|---------|-------|-------|---|
| ADV | 0 | 0 | 0 | 0 | 0 | ... | 0 | 1 | 0 | 0 | 0 |
| ADJ | 0 | 1 | 0 | 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 |
| VERB | 0 | 0 | 0 | 0 | 0 | ... | 0 | 0 | 1 | 0 | 0 |
| NOUN | 0 | 0 | 1 | 0 | 0 | ... | 0 | 0 | 0 | 1 | 0 |

Figure 7.3. Sentence tagged with parts of speech

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

token The right word may be effective , but no word was ever as effective as a rightly timed pause .

ADV 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0

ADJ 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0

VERB 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

NOUN 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0

To keep things efficient, PyTorch does not accept arbitrary Pandas or numpy objects. Instead you must convert all input data to `torch.Tensor` containers with `torch.float` or `torch.int` data type (`dtype`) objects inside.

Listing 7.7. Convert a DataFrame to a tensor with the correct size

```
>>> import torch  
>>> x = torch.tensor(df.iloc[1:]).astype(float).values, dtype=torch.float32  
>>> x = x.unsqueeze(0) #2
```

Now you construct that pattern that we want to search for in the text: adverb, verb, then noun. You will need to create a separate filter or kernel for each part of speech that you care about. Each kernel will be lined up with the others to find the pattern you're looking for in all aspects of the meaning of the words simultaneously.

Before you had only one dimension to worry about, the adverb tag. Now

you'll need to work with all 4 dimensions of these word vectors to get the pattern right. And you need to coordinate four different "features" or channels of data. So for a 3-word, 4-channel kernel we need a 4x3 matrix. Each row represents a channel (part of speech tag), and each column represents a word in the sequence. The word vectors are 4-D column vectors.

```
>>> kernel = pd.DataFrame(  
...     [[1, 0, 0.],  
...      [0, 0, 0.],  
...      [0, 1, 0.],  
...      [0, 0, 1.]], index=tags)  
>>> print(kernel)
```

You can see that this DataFrame is just an exact copy of the sequence of vectors you want to match in your text samples. Of course you were only able to do this because you knew what you were looking for in this one toy example. In a real neural network the deep learning optimizer will use back propagation to *learn* the sequences of vectors that are most helpful in predicting your target variable (the label).

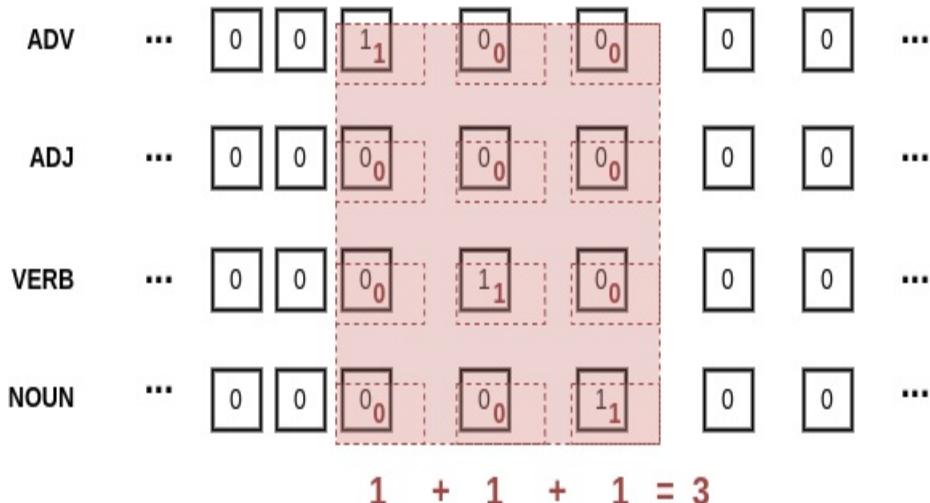
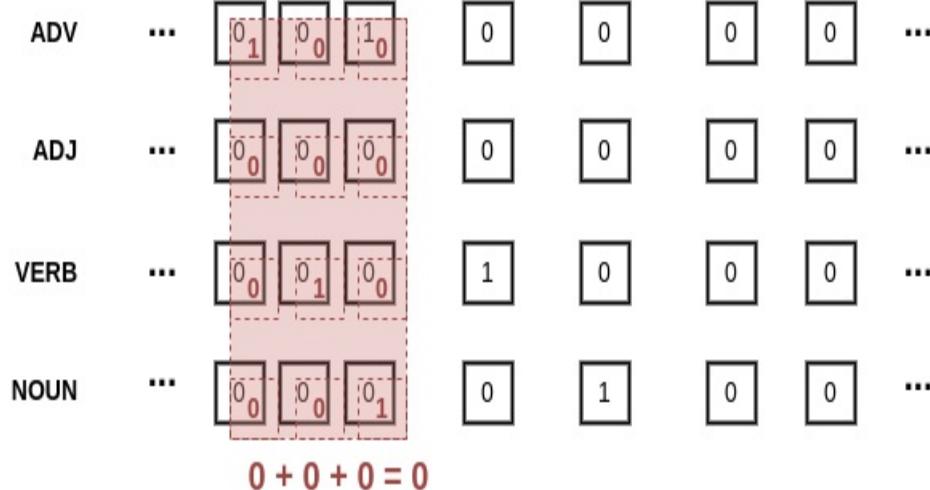
How is it possible for a machine to match patterns? What is the math that causes a kernel to always match the pattern that it contains? In Figure 7.4 you can do the math yourself for a couple strides of the filter across your data. This will help you see how all this works and why it's so simple and yet so powerful.

Figure 7.4. Check the convolution pattern match yourself

kernel
(pattern)

... as a rightly timed pause . <pad> ...

| | | |
|---|---|---|
| 1 | 0 | 0 |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 1 |



Have you checked the math in Figure 7.4? Make sure you do this before you let PyTorch do the math, to embed this pattern of math in your neural network so you can do it in the future if you ever need to debug problems with your CNN.

In PyTorch or any other deep learning framework designed to process multiple samples in parallel you have to unsqueeze the kernel to add a dimension to hold additional samples. Your unsqueezed kernel (weight

matrix) needs to be the same shape as your batch of input data. The first dimension is for the samples from your training or test datasets that are being input to the convolutional layer. Normally this would be the output of an embedding layer and would already be sized appropriately. But since you are hard-coding all the weights and input data to get to know how the Conv1d layer works, you will need to unsqueeze the 2-D tensor matrix to create a 3-D tensor cube. Since you only have the one quote you want to push forward through the convolution the dataset you only need a size of 1 in the first dimension.

Listing 7.8. Load hard-coded weights into a Conv1d layer

```
>>> kernel = torch.tensor(kernel.values, dtype=torch.float32)
>>> kernel = kernel.unsqueeze(0) # ...
>>> conv = torch.nn.Conv1d(in_channels=4,
...                      out_channels=1,
...                      kernel_size=3,
...                      bias=False)
>>> conv.load_state_dict({'weight': kernel})
>>> print(conv.weight)

tensor([[[[1., 0., 0.],
          [0., 0., 0.],
          [0., 1., 0.],
          [0., 0., 1.]]]])
```

Finally you're ready to see if your hand-crafted kernel can detect a sequence of adverb, verb, noun in this text.

Listing 7.9. Running a single example through a convolutional layer

```
>>> y = np.array(conv.forward(x).detach()).squeeze()
>>> df.loc['y'] = pd.Series(y)
>>> df
      0      1      2      3      4     ...     15      16      17      1
token  The  right  word   may    be    ...     a  rightly  timed  paus
ADV      0      0      0      0      0    ...     0      1      0
ADJ      0      1      0      0      0    ...     0      0      0
VERB     0      0      0      1      0    ...     0      0      1
NOUN     0      0      1      0      0    ...     0      0      0
y       1.0     0.0    1.0    0.0    0.0    ...    0.0     3.0    0.0    Na
```

Figure 7.5. Conv1d output predicting rightly timed pause

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|

token The right word may be effective , but no word was ever as effective as a rightly timed pause .

ADV 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0

ADJ 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0

VERB 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0

NOUN 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0

y 1 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 3 0 NaN NaN

The y value reaches a maximum value of 3 where all 3 values of 1 in the kernel line up perfectly with the three 1's forming the same pattern within the part-of-speech tags for the sentence. Your kernel correctly detected the adverb, verb, noun sequence at the end of the sentence. The value of 3 for your convolution output rightly lines up with the word "rightly", the 16th word in the sequence. This is where the sequence of 3 words is located which match your pattern at positions 16, 17, and 18. And it makes sense that the output would have a value of three, because each of the three matched parts of speech had a weight of one in your kernel, summing to a total of three matches.

Don't worry, you'll never have to hand-craft a kernel for a convolutional neural network ever again... unless you want to remind yourself how the math is working so you can explain it to others.

7.3.7 Natural examples

In the optical world of eyes and cameras, convolution is everywhere. When you look down at the surface of the ocean or a lake with polarizing sunglasses, the lenses do convolution on the light to filter out the noise. The lenses of polarized glasses help fishermen filter out the scattered light and see beneath the surface of the water to find fish.

And for a wilder example, consider a zebra standing behind a fence. The stripes on a zebra can be thought of as a visual natural language. A zebra's stripes send out a signal to predators and potential mates about the health of that zebra. And the convolution that happens when a zebra is running among grass or bamboo or tree trunks can create a shimmering effect that makes Zebras difficult to catch.

In figure 7.6 you can think of the cartoon fence as a kernel of alternating numerical values. And the zebra in the background is like your data with alternating numerical values for the light and dark areas in its stripes. And convolution is symmetric because multiplication and addition are commutative operations. So if you prefer you can think of the zebra stripes as the filter and a long length of fence as the data.

Figure 7.6. Zebra behind a fence [\[281\]](#)



Imagine the zebra in figure 7.6 walking behind the fence or the fence sliding in front of the zebra. As the zebra walks, the gaps in the fence will periodically line up with the zebra's stripes. This will create a pattern of light and dark as we move the fence (kernel) or the zebra. It will become dark in places where the zebra's black strips line up with the gaps in the brown fence. And the zebra will appear brighter where the white parts of its coat line up with the fence gaps so they can shine through. So if you want to recognize alternating values of black and white or alternating numerical values you can use alternating high (1) and low values (0) in your kernel.

If you don't see zebras walking behind fences very often, maybe this next analogy will be better. If you spend time at the beach you can imagine the surf as a natural mechanical convolution over the bottom of the ocean. As waves pass over the sea floor and approach the beach they rise or fall depending on what is hidden underneath the surface such as sandbars and

large rocks or reefs. The sand bars and rocks are like components of word meaning that you are trying to detect with your convolutional neural network. This cresting of the waves over the sand bars is like the multiplication operation of convolution passing in waves over your data.

Now imagine that you've dug a hole in the sand near the edge of the water. As the surf climbs the shore, depending on the height of the waves, some of the surf will spill into your little pool. The pool or moat in front of your sand castle is like the reduce or sum operation in a convolution. In fact you will see later that we use an operation called "max pooling" which behaves very much like this in a convolutional neural network. Max pooling helps your convolution measure the "impact" of a particular pattern of words just as your hole in the sand accumulates the impact of the surf on the shore. If nothing else, this image of surf and sand castles will help you remember the technical term *max pooling* when you see it later in this chapter.

[279] Mastodon is a FOSS ad-free microblogging platform similar to Twitter with an open standard API for retrieving NLP datasets (<https://mastodon.social>)

[280] GreenPill is a regenerative economics initiative that encourages crypto investors to contribute to public goods (<https://greenpill.party>).

[281] GDFL (GNU Free Documentation License) pt.wikipedia.org
https://pt.wikipedia.org/wiki/Zebra#/media/Ficheiro:Zebra_standing_alone_cr

7.4 Morse code

Before ASCII text and computers, and even telephones, there was another way to communicate natural language: *Morse code*. [282] Morse code is a text encoding that substitutes dots and dashes for natural language letters and words. These dots and dashes become long and short beeping tones on a telegraph wire or over the radio. Morse code sounds like the beeping in a really really slow dial-up Internet connection. Play the audio file used in the Python example later in this section to hear it for yourself. [283] Amateur radio operators send messages around the world by tapping on a single key. Can you imagine typing text on a computer keyboard that has only one key like

the Framework laptop spacebar in Figure 7.7?!

Figure 7.7. A single key laptop keyboard

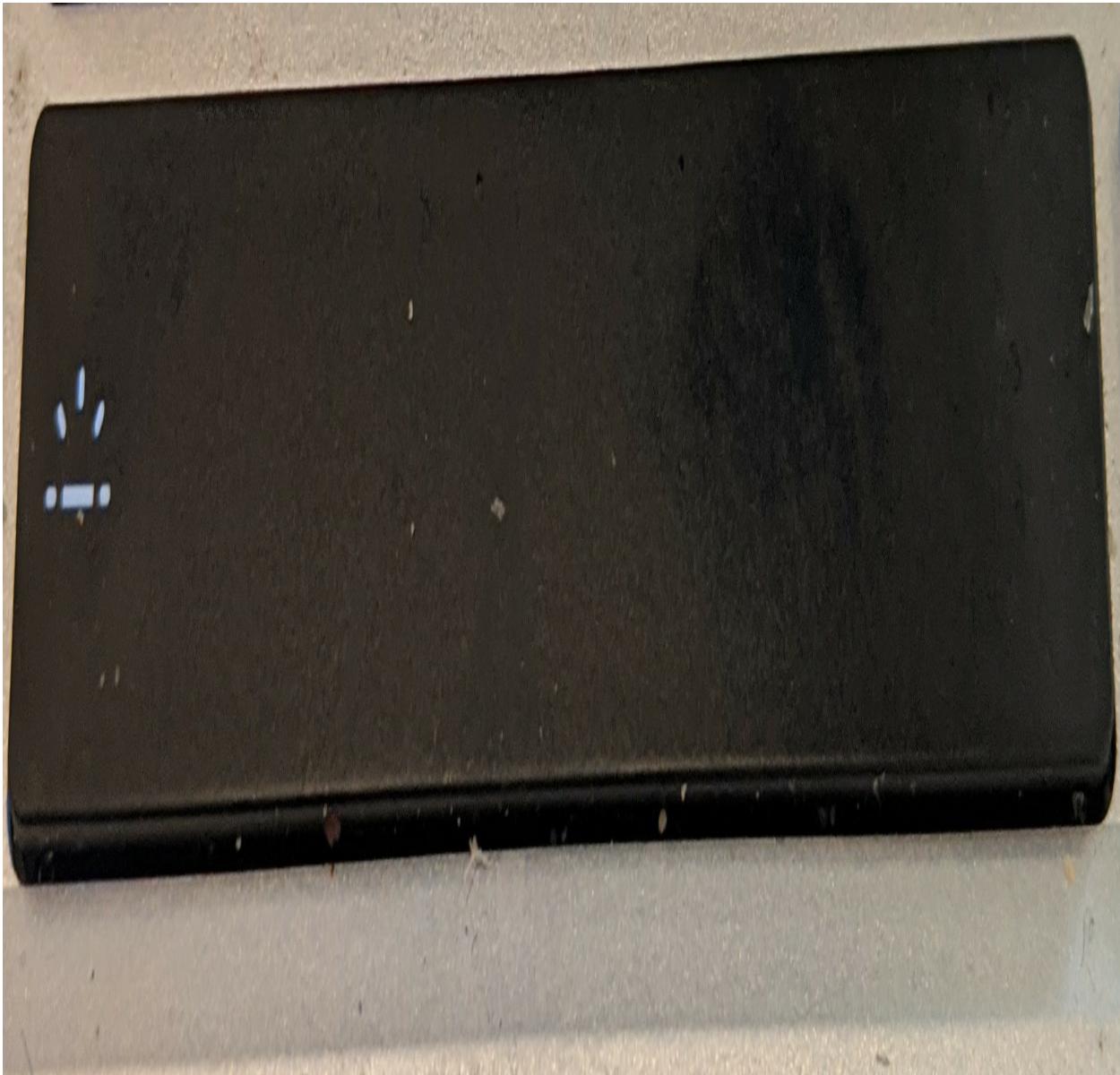


Figure 7.8 shows what an actual Morse code key looks like. Just like the key on a computer keyboard or the fire button on a game controller, the Morse code key just closes an electrical contact whenever the button is pressed.

Figure 7.8. An antique Morse code key



Morse code is a language designed to be tapped out on a single key like this. It was used a lot in the age of telegraph, before telephones made it possible to send voice and data over wires. To visualize Morse code on paper people draw dots and dashes to represent short and long taps the key. You press the key down briefly to send out a dot, and you press it down a bit longer to send out a dash. There's nothing but silence when you aren't pressing the key at all. So it's a bit different than typing text. It's more like using your keyboard as the fire button on game controller. You can imagine a Morse code key like a video game laser or anything that sends out energy only while the key is pressed. You might even find a way to send secret messages in multiplayer games using your weapon as a telegraph.

Communicating with a single key on a computer keyboard would be nearly impossible if it weren't for Samuel Morse's work to create a new natural language. Morse did such a good job designing the language of Morse code, even ham-fisted amateur radio operators like me can use it in a pinch.[\[284\]](#) You're about to learn the 2 most important bits of the language so you can use it too in an emergency. Don't worry, you're only going to learn 2 letters of the language. That should be enough to give you a clearer understanding of convolution and how it works on natural languages.

Figure 7.9. Morse code dictionary

International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

| | |
|---|-----------|
| A | • - |
| B | - - . . . |
| C | - - . - . |
| D | - - . . |
| E | • |
| F | • . - - . |
| G | - - - . |
| H | • |
| I | • • |
| J | • - - - - |
| K | - - . - |
| L | • - - . . |
| M | - - - |
| N | - - . |
| O | - - - - |
| P | • - - - . |
| Q | - - - . - |
| R | • - - . |
| S | • . . . |
| T | - |

| | |
|---|-----------|
| U | • . - |
| V | • . . - |
| W | • - - |
| X | - - . . - |
| Y | - - . - - |
| Z | - - - . . |

| | |
|---|-------------|
| 1 | • - - - - |
| 2 | • . - - - |
| 3 | • . . - - |
| 4 | • . . . - |
| 5 | • |
| 6 | - - . . . |
| 7 | - - - . . . |
| 8 | - - - - . . |
| 9 | - - - - - . |
| 0 | - - - - - |

Morse code is still used today in situations when the radio waves are too noisy for someone to understand your voice. It's especially useful when you really, really, really need to get a message out. Sailors trapped in an air pocket within a sunken submarine or ship have used it to communicate with rescuers by banging out Morse code on the metal hull. And people buried under rubble after earthquakes or mining accidents will bang on metal pipes and girders to communicate with rescuers. If you know a bit of Morse code you might be able to have a two-way conversation with someone, just by banging out your words in Morse code.

Here's the example audio data for a secret message being broadcast in Morse code. You will process it in the next section using using a hand-crafted convolution kernel. For now you probably just want to play the audio track so you can hear what Morse code sounds like.

Listing 7.10. Download secret message

```
>>> from nlpi2.init import maybe_download  
  
>>> url = 'https://upload.wikimedia.org/wikipedia/' \  
     'commons/7/78/1210secretmorsecode.wav'  
>>> filepath = maybe_download(url)    #1  
>>> print(filepath)  
  
/home/hobs/.nlpi2-data/1210secretmorsecode.wav
```

Of course your .nlpi2-data directory will be located in your \$HOME directory rather than mine. That's where you'll find all the data used in these examples. Now you can load the wav file to create an array of numerical values for the audio signal that you can process later with convolution.

7.4.1 Decoding Morse with convolution

If you know a little Python you can build a machine that can interpret Morse code for you so you won't have to memorize all those dots and dashes in the morse code dictionary of figure 7.9. Could come in handy during the zombie apocalypse or "The Big One" (Earthquake in California). Just make sure you hang onto a computer or phone that can run Python.

Listing 7.11. Load the secret Morse code wav file

```
>>> from scipy.io import wavfile  
  
>>> sample_rate, audio = wavfile.read(filepath)  
>>> print(f'sample_rate: {sample_rate}')  
>>> print(f'audio:\n{audio}')  
  
sample_rate: 4000  
audio:  
[255  0 255 ...  0 255  0]
```

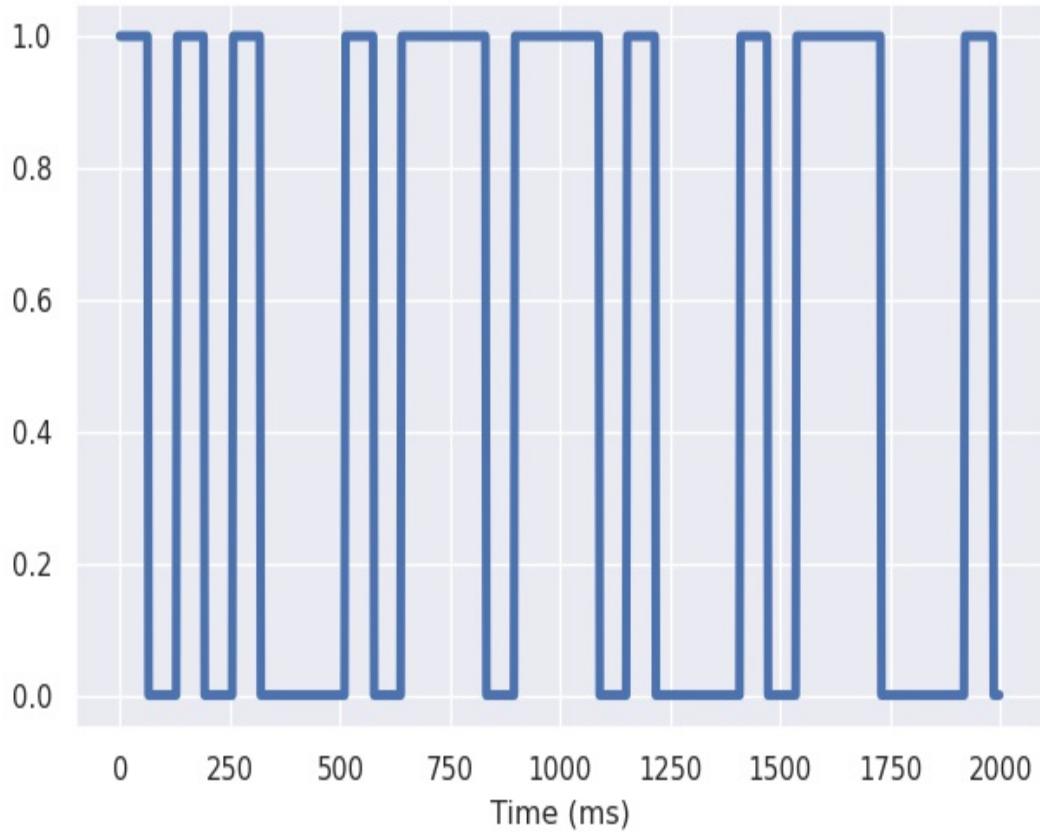
The audio signal in this wav file oscillates between 255 and 0 (max and min uint8 values) when there is a beep tone. So you need to rectify the signal using `abs()` and then normalize it so the signal will be 1 when a tone is playing and 0 when there is no tone. You also want to convert the sample numbers to milliseconds and downsample the signal so it's easier to examine individual values and see what's going on. Listing 7.12 centers, normalizes, and downsamples the audio data and extracts the first two seconds of this audio data.

Listing 7.12. Normalize and downsample the audio signal

```
>>> pd.options.display.max_rows = 7  
  
>>> audio = audio[:sample_rate * 2] #1  
>>> audio = np.abs(audio - audio.max() / 2) - .5 #2  
>>> audio = audio / audio.max() #3  
>>> audio = audio[:sample_rate // 400] #4  
>>> audio = pd.Series(audio, name='audio')  
>>> audio.index = 1000 * audio.index / sample_rate #5  
>>> audio.index.name = 'time (ms)'  
>>> print(f'audio:\n{audio}')
```

Now, you can plot your shiny new Morse code dots and dashes with `audio.plot()`.

Figure 7.10. Square waves morse code secret message



Can you see where the dots are in figure 7.10? The dots are 60 milliseconds of silence (signal value of 0) followed by 60 milliseconds of tone (signal value of 1) and then 60 seconds of silence again (signal value of 0).

To detect a dot with convolution you want to design a kernel that matches this pattern of low, high, low. The only difference is that for the low signal, you need to use a negative one rather than a zero, so the math adds up. You want the output of the convolution to be a value of one when a dot symbol is detected.

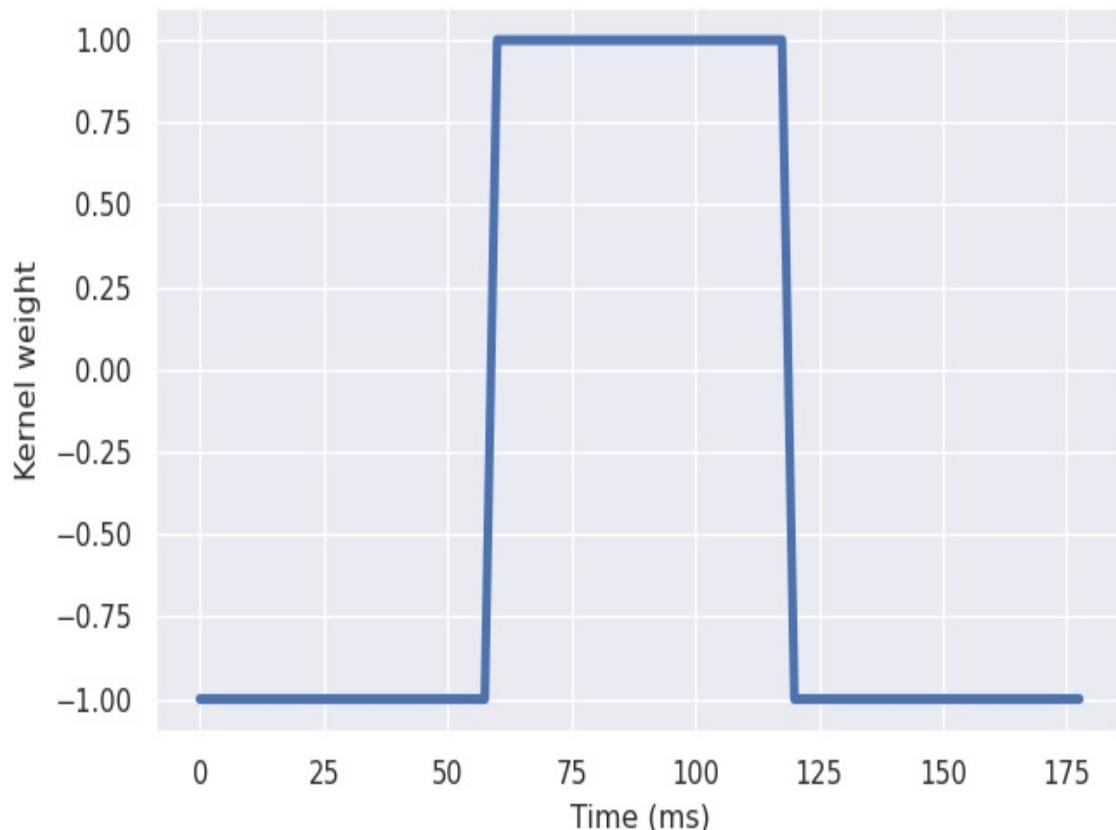
Lising 7.12 shows how to build dot-detecting kernel.

Listing 7.13. Dot detecting kernel

```
>>> kernel = [-1] * 24 + [1] * 24 + [-1] * 24
>>> kernel = pd.Series(kernel, index=2.5 * np.arange(len(kernel)))
```

```
>>> kernel.index.name = 'Time (ms)'  
>>> ax = kernel.plot(linewidth=3, ylabel='Kernel weight')
```

Figure 7.11. Morse code dot detecting kernel



You can try out your hand-crafted kernel by convolving it with the audio signal to see if it is able to detect the dots. The goal is for the convolved signal to be high, close to one, near the occurrences of a dot symbol, the short blips in the audio. You also want your dot detecting convolution to return a low value (close to zero) for any dash symbols or silence that comes before or after the dots.

Listing 7.14. Dot detector convolved with the secret message

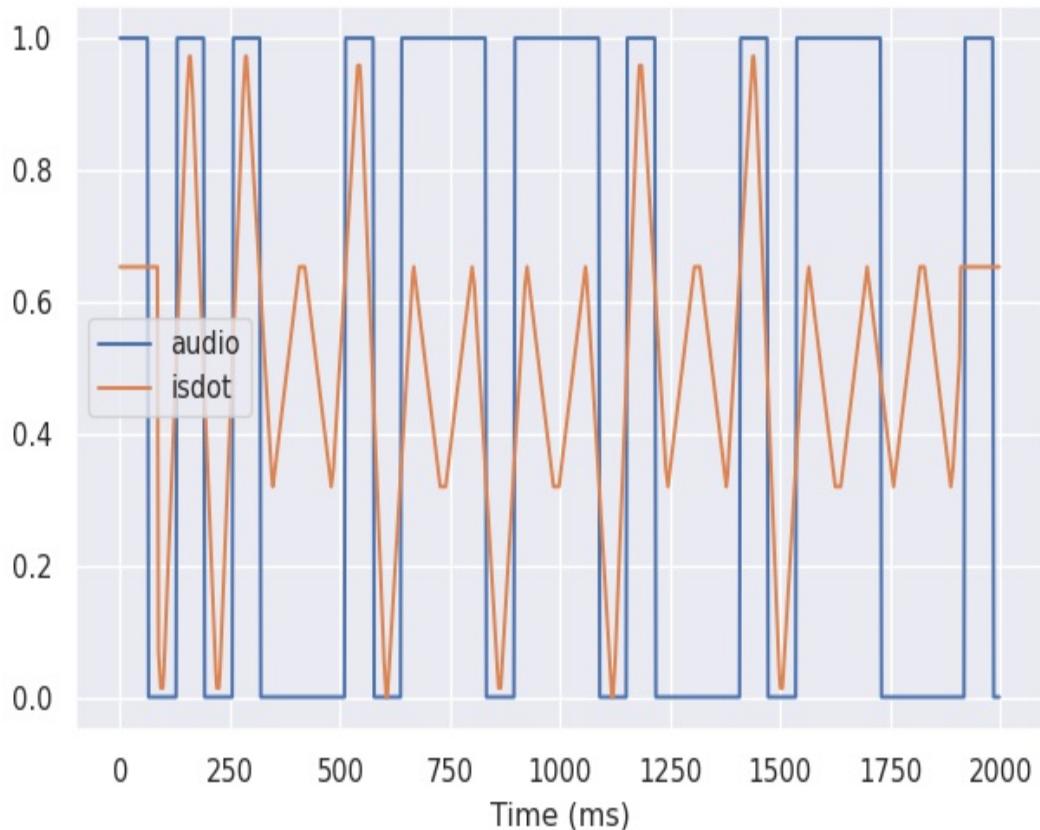
```
>>> kernel = np.array(kernel) / sum(np.abs(kernel))      #1  
>>> pad = [0] * (len(kernel) // 2)                      #2  
>>> isdot = convolve(audio.values, kernel)
```

```

>>> isdot = np.array(pad[:-1] + list(isdot) + pad)      #3
>>> df = pd.DataFrame()
>>> df['audio'] = audio
>>> df['isdot'] = isdot - isdot.min()
>>> ax = df.plot()

```

Figure 7.12. Hand-crafted dot detecting convolution



Looks like the hand-crafted kernel did all right! The convolution output is close to one only in the middle of the dot symbols.

Now that you understand how convolution works, feel free to use the `np.convolve()` function. It works faster and gives you more options for the mode of handling the padding.

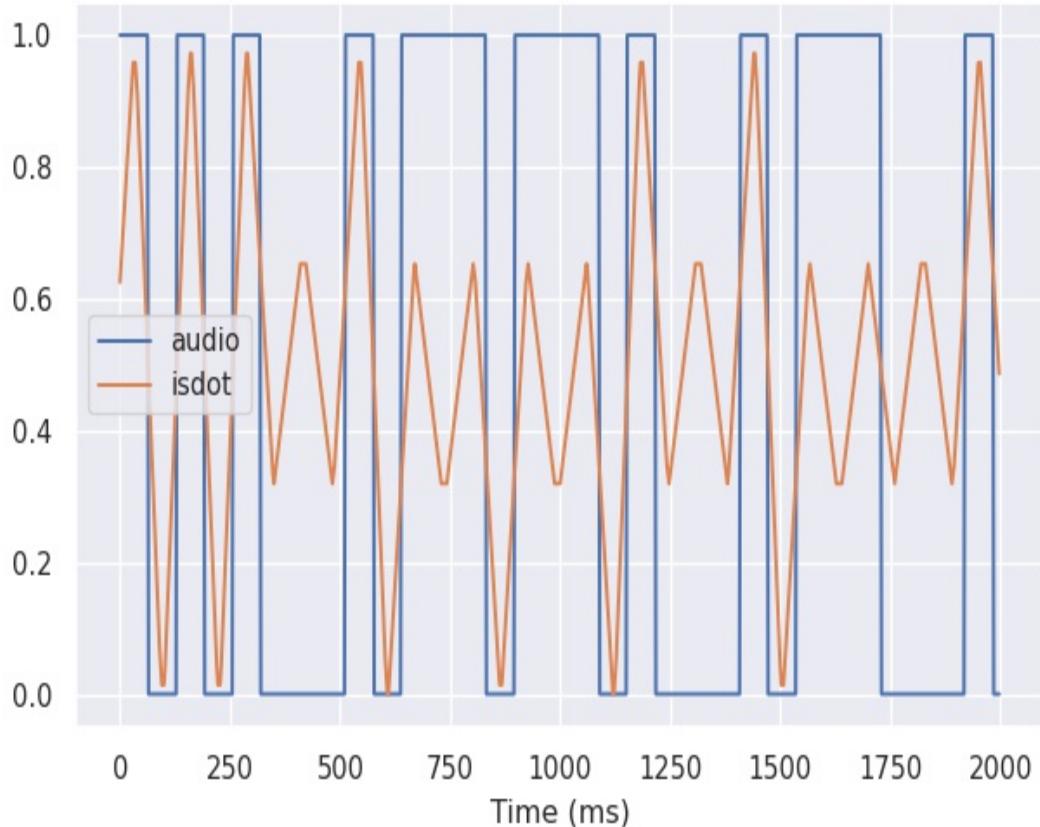
Listing 7.15. Numpy convolve

```

>>> isdot = np.convolve(audio.values, kernel, mode='same')      #1
>>> df['isdot'] = isdot - isdot.min()
>>> ax = df.plot()

```

Figure 7.13. Numpy convolution



Numpy convolution gives you three possible modes for doing the convolution, in order of increasing output length:

1. **valid**: Only output $\text{len(kernel)} - 1$ values for the convolution as our pure python `
2. **same**: Output a signal that is the same length as the input by extrapolating the signal beyond the beginning and end of the array.
3. **full**: Output signal will have more sample than the input signal.

The numpy convolution set to 'same' mode seems to work better on our

Morse code audio signal. So you'll want to check that your neural network library uses a similar mode when performing convolution within your neural network.

That was a lot of hard work building a convolutional filter to detect a single symbol in a Morse code audio file. And it wasn't even a single character of natural language text, just one third of the letter "S"! Fortunately all you laborious hand-crafting is over. It's possible to use the power of back-propagation within neural networks to *learn* the right kernels to detect all the different signals important to your problem.

[282] "Morse code" article on Wikipedia
(https://en.wikipedia.org/wiki/Morse_code)

[283] Wikipedia commons secret message wave file
(<https://upload.wikimedia.org/wikipedia/commons/7/78/1210secretmorsecode>)

[284] "Ham" was originally a pejorative term for ham-fisted Morse code "typists" (https://en.wikipedia.org/wiki/Amateur_radio#Ham_radio)

7.5 Building a CNN with PyTorch

Figure 7.14 shows you how text flows into a CNN network and then outputs a embedding. Just as with previous NLP pipelines, you need to tokenize your text first. Then you identify the set of all the tokens used in your text. You ignore the tokens you don't want to *count* and assign an integer index to each word in your vocabulary. The input sentence has 4 tokens so we start with a sequence of 4 integer indices, one for each token.

CNNs usually use word embeddings rather than one-hot encodings to represent each word. You initialize a matrix of word embeddings that has the same number of rows as words in your vocabulary and 300 columns if you want to use 300-D embeddings. You can set all your initial word embeddings to zero or some small random values. If you want to do knowledge transfer and use pretrained word embeddings, you then look up your tokens in GloVe, Word2vec, fastText or any word embeddings you like. And you insert these vectors into your matrix of embeddings at the matching row

based on your vocabulary index.

For this four-token sentence you then look up the appropriate word embedding get a sequence of 4 embedding vectors once you have looked up each embedding in your word embedding matrix. You also get additional padding token embeddings that are typically set to zeros so they don't interfere with the convolution. If you used the smallest GloVe embeddings, your word embeddings are 50 dimensional, so you end up with a 50×4 matrix of numerical values for this single short sentence.

Your convolutional layer can process each of these 50 dimensions with a 1-D convolutional kernel to squeeze this matrix of information about your sentence a bit. If you used a kernel of size (length) of two, and a stride of two, you would end up with a matrix of size 50×3 to represent the sequence of four 50-D word vectors.

A *pooling layer*, typically max pooling, is used to reduce the size of the output even further. A max pooling layer with 1-D kernel will compress your sequence of three 50-D vectors down to a single 50-D vector. As the name implies, max pooling will take the largest most impactful output for each channel (dimension) of meaning in your sequence of vectors. Max pooling is usually pretty effective because it allows your convolution to find the most important dimensions of meaning for each n-gram in your original text. With multiple kernels they can each specialize on a separate aspect of the text that is influencing your target variable.

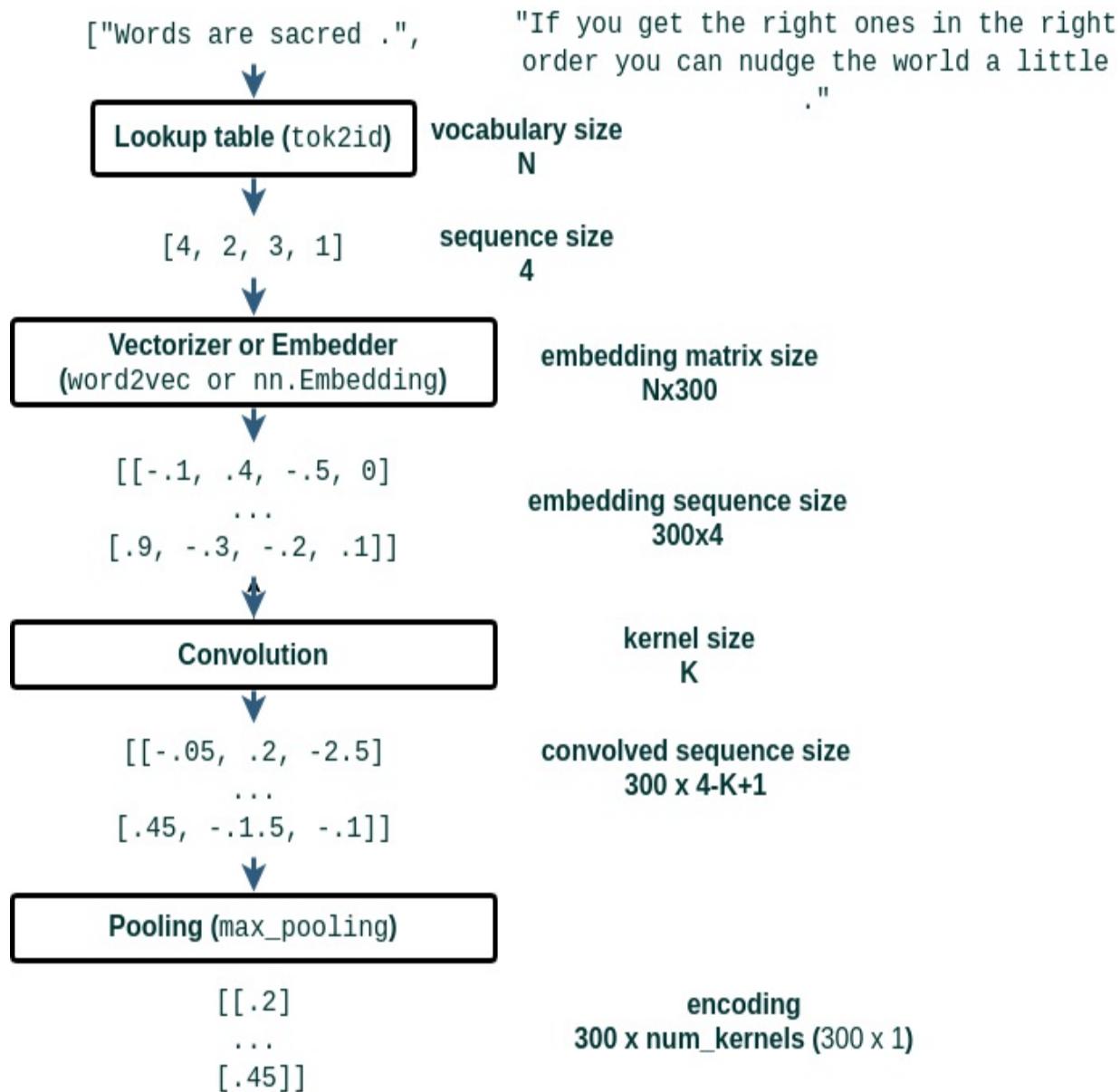


Note

You should call the output of a convolutional layer an "encoding" rather than an "embedding". Both words are used to describe high dimensional vectors, but the word "encoding" implies processing over time or in a sequence. The convolution math happens over time in your sequences of word vectors, whereas "embedding" vectors are the result of processing of a single unchanging token. Embeddings don't encode any information about the order or sequence of words. Encodings are more complete representations of the meaning of text because they account for the order of words in the same way that your brain does.

The encoding vector output by a CNN layer is a vector with whatever size (length) you specify. The length (number of dimensions) of your encoding vector doesn't depend in any way on the length of your input text.

Figure 7.14. CNN processing layers [\[285\]](#)



You're going to need all your skills from the previous chapters to get the text in order so it can be input into your neural network. The first few stages of your pipeline in figure 7.14 are the tokenization and case folding that you did in previous chapters. You will use your experience from the previous

examples to decide which words to ignore, such as stopwords, punctuation, proper nouns, or really rare words.

Filtering out and ignoring words based on an arbitrary list of stopwords that you handcraft is usually a bad idea, especially for neural nets such as CNNs. Lemmatizing and stemming is also usually not a good idea. The model will know much more about the statistics of your tokens than you could ever guess at with your own intuition. Most examples you see on Kaggle and DataCamp and other data science websites will encourage you to hand craft these parts of your pipeline. You know better now.

You aren't going to handcraft your convolution kernels either. You are going to let the magic of backpropagation take care of that for you. A neural network can learn most of the parameters of your model, such as which words to ignore and which words should be lumped together because they have similar meaning. In fact, in chapter 6 you learned to represent the meanings of words with embedding vectors that capture exactly how they are similar to other words. You no longer have to mess around with lemmatization and stemming, as long as you have enough data to create these embeddings.

7.5.1 Clipping and Padding

CNN models require a consistent length input text so that all the output values within the encoding are at consistent positions within that vector. This ensures that the encoding vector your CNN outputs always has the same number of dimensions no matter how long, or short your text is. Your goal is to create vector representations of both a single character string and a whole page of text. Unfortunately a CNN can't work with variable length text, so many of the words and characters will have to be "clipped" off at the end of your string if your text is too long for your CNN. And you need to insert filler tokens, called *padding*, to fill in the gaps in strings that are too short for your CNN.

Remember that the convolution operation reduces the length of the input sequence by the same amount no matter how long it is. Convolution will always reduce the length of the input sequence by one less than the size of

your kernel. And any pooling operation, such as max pooling, will also consistently reduce the length of the input sequence. So if you didn't do any padding or clipping, long sentences would produce longer encoding vectors than shorter sentences. And that won't work for an encoding, which needs to be size-invariant. You want your encoding vectors to always be the same length no matter the size of your input.

This is a fundamental properties of vectors, that they have the same number of dimensions for the entire *vector space* that you are working in. And you want your NLP pipeline to be able to find a particular bit of meaning at the same location, or vector dimension, no matter where that sentiment occurred in a piece of text. Padding and clipping ensures that your CNN is location (time) and size (duration) invariant. Basically your CNN can find patterns in the meaning of text no matter where those patterns are in the text, as long as those patterns are somewhere within the maximum length that your CNN can handle.

You can chose any symbol you like to represent the padding token. Many people use the token "<PAD>", because it doesn't exist in any natural language dictionary. And most English speaking NLP engineers will be able to guess what "<PAD>" means. And your NLP pipeline will see that these tokens are repeated a lot at the end of many strings. This will help it create the appropriate "filler" sentiment within the embedding layer. If you're curious about what filler sentiment looks like, load your embedding vectors and compare the your embedding for "<PAD>" to the embedding for "blah" as in "blah blah blah". You just have to make sure that you use a consistent token and tell your embedding layer what token you used for your padding token. It's common to make this the first token in your id2token or vocab sequence so it has an index and id value of 0.

Once you've let everybody know what your padding token is, you now need to actually decide on a consistent padding approach. Just as in computer vision, you can pad either side of your token sequence, the beginning or the end. And you can even split the padding and put half at the beggining and half at the beginning. Just don't insert them between words. That would interfere with the convolution math. And make sure you add the total number of padding tokens required to create the correct length sequences for your

CNN.

In listing Listing 7.16 you will load "birdsite" (microblog) posts that have been labeled by Kaggle contributors with their news-worthiness. Later you'll use your CNN model to predict whether CNN (Cable News Network) would be likely to "pick up" on the news before it spreads on its own in the "miasma."



Important

We intentionally use words that nudge you towards prosocial, authentic, mindful behavior. The dark patterns that permeate the Internet have nudged creative powerhouses in the tech world to create an alternate, more authentic universe with it's own vocabulary.

"Birdsite": What "fedies" call Twitter

"Fedies": Users of federated social media apps that protect your well-being and privacy

"Fediverse" Alternate universe of federated social media apps (Mastodon, PeerTube)

"Nitter" is a less manipulative frontend for Twitter

"Miasma" is Neil Stephenson's name for a dystopian Internet

Listing 7.16. Load news posts

```
df = pd.read_csv(HOME_DATA_DIR / 'news.csv')
df = df[['text', 'target']]      #1
print(df)
```

| | | text | target |
|------|---|------|--------|
| 0 | Our Deeds are the Reason of this #earthquake M... | | 1 |
| 1 | Forest fire near La Ronge Sask. Canada | | 1 |
| 2 | All residents asked to 'shelter in place' are ... | | 1 |
| ... | | ... | ... |
| 7610 | M1.94 [01:04 UTC]?5km S of Volcano Hawaii. ht... | | 1 |
| 7611 | Police investigating after an e-bike collided ... | | 1 |

```
7612 The Latest: More Homes Razed by Northern Calif...
[7613 rows x 2 columns]
```

1

You can see in the examples above that some microblog posts push right up against the character limit of birdsite. Others get the point across with fewer words. So you will need to pad, or fill, these shorter texts so all of the examples in your dataset have the same number of tokens. If you plan to filter out really frequent words or really rare words later in your pipeline, your padding function needs to fill in those gaps too. So listing 7.17 tokenizes these texts and filters out a few of the most common tokens that it finds.

Listing 7.17. Most common words for your vocabulary

```
import re
from collections import Counter
from itertools import chain
HOME_DATA_DIR = Path.home() / '.nlpia2-data'

counts = Counter(chain(*[
    re.findall(r'\w+', t.lower()) for t in df['text']]))      #1
vocab = [tok for tok, count in counts.most_common(4000)[3:]]  #2

print(counts.most_common(10))
[('t', 5199), ('co', 4740), ('http', 4309), ('the', 3277), ('a',
('in', 1986)]
```

You can see that the token "t" occurs almost as many times (5199) as there are posts (7613). This looks like part of a url created by a url shortener often used to track microbloggers on this app. You should ignore the first three url-like tokens if you want your CNN to focus on just the meaning of the words in the content that a human would likely read. If your goal is to build a CNN that reads and understands language like a human, you would create a more sophisticated tokenizer and token filter to strip out any text that humans don't pay attention to, such as URLs and geospatial coordinates.

Once you have your vocabulary and tokenizer dialed in, you can build a padding function to reuse whenever you need it. If you make your pad() function general enough, as in listing 7.18, you can use it on both string tokens and integer indexes.

Listing 7.18. Multipurpose padding function

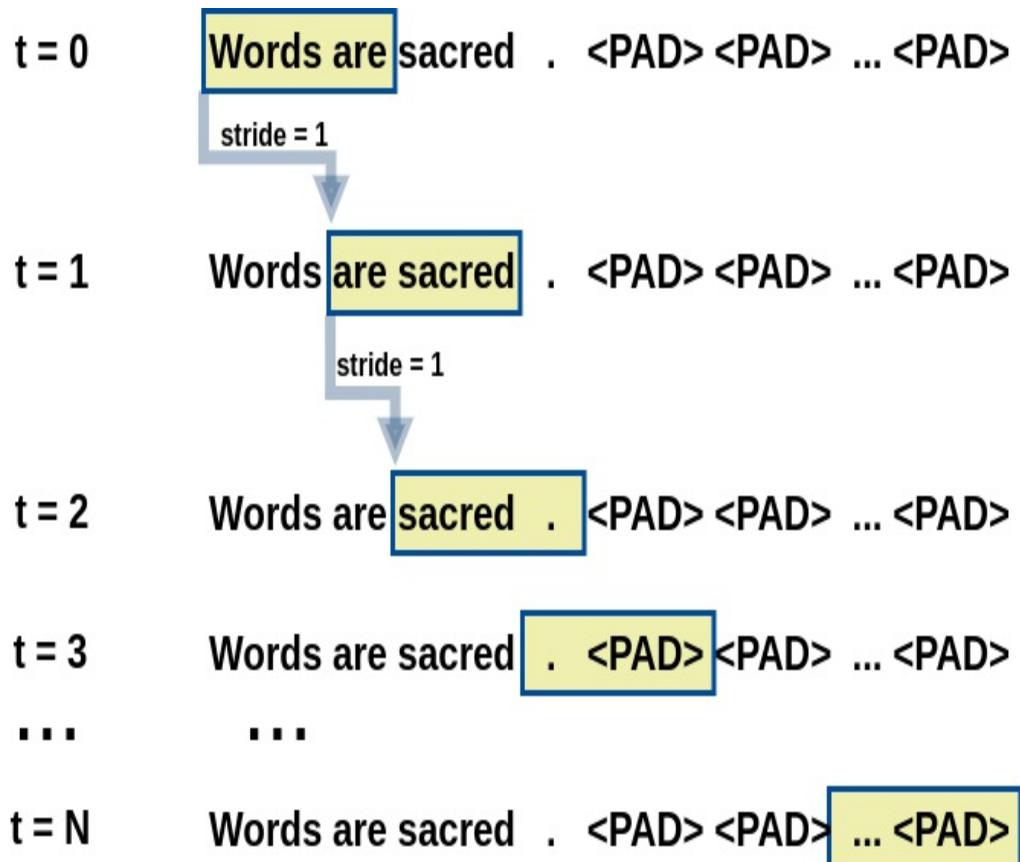
```
def pad(sequence, pad_value, seq_len):  
    padded = list(sequence)[:seq_len]  
    padded = padded + [pad_value] * (seq_len - len(padded))  
    return padded
```

We have one last preprocessing step to do for CNNs to work well. You want to include your token embeddings that you learned about in chapter 6.

7.5.2 Better representation with word embeddings

Imagine you are running a short bit of text through your pipeline. Figure 7.15 shows what this would look like before you've turned your word sequence into numbers (or vectors, hint hint) for the convolution operation.

Figure 7.15. Convolution striding



Now that you have assembled a sequence of tokens, you need to represent

their meaning well for your convolution to be able to compress and encode all that meaning. For the fully-connected neural networks we used in chapter 5 and 6 you could use one-hot encoding. But one-hot encoding creates extremely large, sparse matrices and you can do better than that now. You learned a really powerful way to represent words in chapter 6: word embeddings. Embeddings are much more information-rich and dense vector representation of your words. A CNN, and almost any other deep learning or NLP model, will work better when you represent words with embeddings. Figure 7.11 shows you how to do that.

Figure 7.16. Word embeddings for convolution

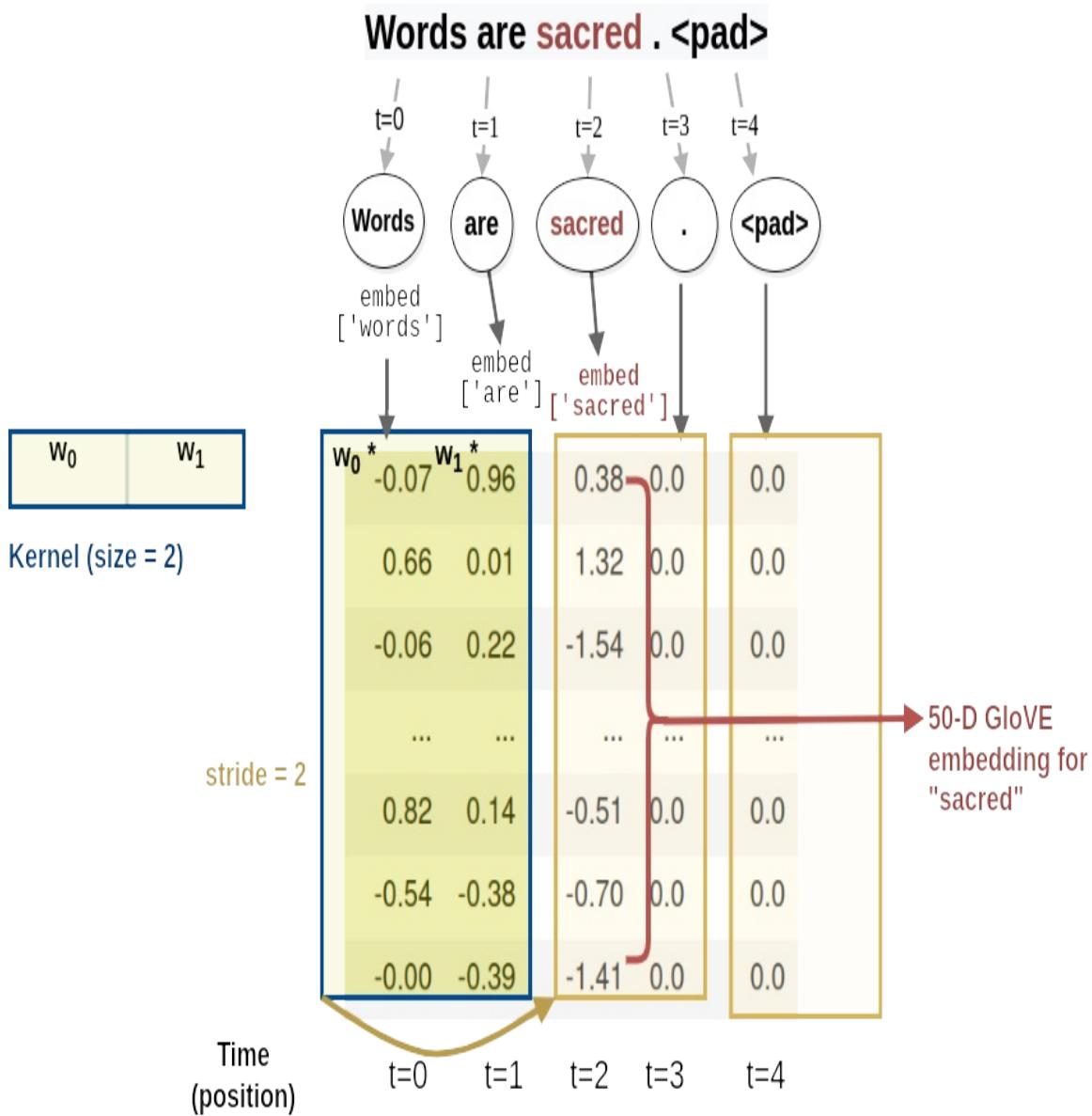


Figure 7.16 shows what the `nn.Embedding` layer in PyTorch is doing behind the scenes. To orient you on how the 1-D convolution slides over your data, the diagram shows 3 steps of a two-length kernel stepping through your data. But how can a 1-D convolution work on a sequence of 300-D GloVe word embeddings? You just have to create a convolution kernel (filter) for each dimension you want to find the patterns in. This means that each dimension of your word vectors is a channel in the convolution layer.

Unfortunately, many blog posts and tutorials may mislead you about the

proper size for a convolutional layer. Many PyTorch beginners assume that the output of an Embedding layer can flow right into a convolution layer without any resizing. Unfortunately this would create a 1-D convolution along the dimensions of the word embeddings rather than the sequence of words. So you will need to transpose your Embedding layer outputs so that the channels (word embedding dimensions) line up with the convolutional channels.

PyTorch has an `nn.Embedding` layer you can use within all your deep learning pipelines. If you want your model to learn the embeddings from scratch you only need to tell PyTorch the number of embeddings you need, which is the same as your vocabulary size. The embedding layer also needs you to tell it the number of dimension to allocate for each embedding vector. Optionally you can define the padding token index id number.

Listing 7.19. Learn your embeddings from scratch

```
from torch import nn

embedding = nn.Embedding(
    num_embeddings=2000,           #1
    embedding_dim=64,             #2
    padding_idx=0)
```

The embedding layer will be the first layer in your CNN. That will convert your token IDs into their own unique 64-D word vectors. And backpropagation during training will adjust the weights in each dimension for each word to match 64 different ways that words can be used to talk about news-worthy disasters. These embeddings won't represent the complete meaning of words the way the FastText and GloVe vectors did in chapter 6. These embeddings are good for only one thing, determining if a Tweet contains newsworthy disaster information or not.

Finally you can train your CNN to see how well it will do on an extremely narrow dataset like the Kaggle disaster tweets dataset. Those hours of work crafting a CNN will pay off with super-fast training time and impressive accuracy.

Listing 7.20. Learn your embeddings from scratch

```

from nlpia2.ch07.cnn.train79 import Pipeline      #1

pipeline = Pipeline(
    vocab_size=2000,
    embeddings=(2000, 64),
    epochs=7,
    torch_random_state=433994,      #2
    split_random_state=1460940,
)

pipeline = pipeline.train()

Epoch: 1, loss: 0.66147, Train accuracy: 0.61392, Test accuracy:
Epoch: 2, loss: 0.64491, Train accuracy: 0.69712, Test accuracy:
Epoch: 3, loss: 0.55865, Train accuracy: 0.73391, Test accuracy:
Epoch: 4, loss: 0.38538, Train accuracy: 0.76558, Test accuracy:
Epoch: 5, loss: 0.27227, Train accuracy: 0.79288, Test accuracy:
Epoch: 6, loss: 0.29682, Train accuracy: 0.82119, Test accuracy:
Epoch: 7, loss: 0.23429, Train accuracy: 0.82951, Test accuracy:

```

After only 7 passes through your training dataset you achieved 79% accuracy on your test set. And on modern laptop CPU this should take less than a minute. And you kept the overfitting to a minimum by minimizing the total parameters in your model. The CNN uses very few parameters compared to the embedding layer.

What happens if you continue the training for a bit longer?

Listing 7.21. Continue training

```

pipeline.epochs = 13      #1
pipeline = pipeline.train()

Epoch: 1, loss: 0.24797, Train accuracy: 0.84528, Test accuracy:
Epoch: 2, loss: 0.16067, Train accuracy: 0.86528, Test accuracy:
...
Epoch: 12, loss: 0.04796, Train accuracy: 0.93578, Test accuracy:
Epoch: 13, loss: 0.13394, Train accuracy: 0.94132, Test accuracy:

```

Oh my, that looks fishy. That's a lot of overfitting - 94% on the training set and 78% on the test set. The training set accuracy kept climbing and eventually got well above 90%. By the 20th epoch the model achieved 94% accuracy on the training set. It's better than even expert humans. Read through a few examples yourself without looking at the label. Can you get

94% of them correct? Here are the first four, after tokenization, ignoring out-of-vocabulary words, and adding padding.

```
pipeline.indexes_to_texts(pipeline.x_test[:4])  
['getting in the poor girl <PAD> <PAD> ...',  
'Spot Flood Combo Cree LED Work Light Bar Offroad Lamp Full ...'  
'ice the meltdown <PAD> <PAD> <PAD> <PAD> ...',  
'and burn for bush fires in St http t co <PAD> <PAD> ...']
```

If you answered ["disaster", "not", "not", "disaster"] then you got all 4 of these right. But keep going. Can you get nineteen out of twenty correct? That's what you'd have to do to beat the training set accuracy of this CNN. It's no surprise this is a hard problem and your CNN is getting only 79% accuracy on the test set. After all, bots are filling Twitter with disaster-sounding tweets all the time. And sometimes even real humans get sarcastic or sensationalist about world events.

What could be causing this overfitting? Are there too many parameters? Too much "capacity" in the neural net? Here's a good function for displaying the parameters in each layer of your PyTorch neural networks.

```
def describe_model(model):      #1  
    state = model.state_dict()  
    names = state.keys()  
    weights = state.values()  
    params = model.parameters()  
    df = pd.DataFrame([  
        dict(  
            name=name,  
            learned_params=int(p.requires_grad) * p.numel(),      #  
            all_params=p.numel(),      #3  
            size=p.size(),  
        )  
        for name, w, p in zip(names, weights, params)  
    ])  
    df = df.set_index('name')  
    return df  
  
describe_model(pipeline.model)      #4
```

| name | learned_params | all_params | size |
|------|----------------|------------|------|
|------|----------------|------------|------|

| | | | | |
|---------------------|--------|--------|------------|----|
| embedding.weight | 128064 | 128064 | (2001, 64) | #1 |
| linear_layer.weight | 1856 | 1856 | (1, 1856) | |
| linear_layer.bias | 1 | 1 | (1,) | |

When you have overfitting you can use pretrained models in your pipeline to help it generalize a bit better.

7.5.3 Transfer learning

Another enhancement that can help your CNN models it to use pretrained word embeddings such as GloVe. And it's not cheating, because these models have been trained in a self-supervised way, without any labels from your disaster tweets dataset. You can transfer all the learning these GloVe vectors contain from the training that Stanford gave them on all of Wikipedia and other larger corpora. This way your model can get a head start learning a vocabulary of words about disasters by using the more general meaning of words. You just need to size your embedding layer to make room for the size GloVe embeddings you want to initialize your CNN with.

Listing 7.22. Make room for GloVE embeddings

```
from torch import nn

embedding = nn.Embedding(
    num_embeddings=2000,          #1
    embedding_dim=50,            #2
    padding_idx=0)
```

That's it. Once PyTorch knows the number of embeddings and their dimensions it can allocate RAM to hold the embedding matrix for `num_embedding` rows and `embedding_dim` columns. This would train your embeddings from scratch at the same time it is training the rest of your CNN. Your domain-specific vocabulary and embeddings would be customized for your corpus. But training your embeddings from scratch doesn't take advantage of the fact that words share meaning across many domains.

If you want your pipeline to be "cross-fit" you can use embedding trained in other domains. This "cross training" of word embeddings is called *transfer learning*. This gives your Embedding layer a head start on learning the

meaning of words by using pretrained word embeddings trained on a much broader corpus of text. For that, you will need to filter out all the words used in other domains so that the vocabulary for your CNN pipeline is based only on the words in your dataset. Then you can load the embeddings for those words into your `nn.Embedding` layer.

Listing 7.23. Load embeddings and align with your vocabulary

```
from nessvec.files import load_vecs_df

glove = load_vecs_df(HOME_DATA_DIR / 'glove.6B.50d.txt')
zeroes = [0.] * 50
embed = []
for tok in vocab:                      #1
    if tok in glove.index:
        embed.append(glove.loc[tok])
    else:
        embed.append(zeros)           #2
embed = np.array(embed)

print(f'embed.shape: {embed.shape}')
print(f'vocab:\n{pd.Series(vocab)}')

embed.shape: (4000, 50)
pd.Series(vocab):
0              a
1              in
2              to
...
3831      43rd
3832  beginners
3833     lover
Length: 3834, dtype: object----
```

You have taken the top 4000 most frequent tokens from the tweets. Of those 4000 words, 3834 are available in the small GloVe embeddings vocabulary. So you filled in those missing 166 tokens with zero vectors for their unknown embeddings. Your model will learn what these words mean and compute their embeddings as you train the Embedding layer within your neural network.

Now that you have a consistent way of identifying tokens with an integer, you can load a matrix of GloVe embeddings into your `nn.Embedding` layer.

Listing 7.24. Initialize your embedding layer with GloVe vectors

```
embed = torch.Tensor(embed)                                #1
print(f'embed.size(): {embed.size()}')
embed = nn.Embedding.from_pretrained(embed, freeze=False)   #2
print(embed)
```

Detecting meaningful patterns

How you say something, the order of the words, makes a big difference. You combine words to create patterns that mean something significant to you, so that you can convey that meaning to someone else.

If you want your machine to be a meaningful natural language processor, it will need to be able to detect more than just the presence or absence of particular tokens. You want your machine to detect meaningful patterns hidden within word sequences. [\[286\]](#)

Convolutions are the filters that bring out meaningful patterns from words. And the best part is, you don't have no longer have to hard-code these patterns into the convolutional kernel. The training process will search for the best possible pattern-matching convolutions for your particular problem. Each time you propagate the error from your labeled dataset back through the network (backpropagation), the optimizer will adjust the weights in each of your filters so that they get better and better at detecting meaning and classifying your text examples.

7.5.4 Robustifying your CNN with dropout

Most neural networks are susceptible to adversarial examples that trick them into outputting incorrect classifications or text. And sometimes neural networks are susceptible to changes as straight forward as synonym substitution, misspellings, or insertion of slang. Sometimes all it takes is a little "word salad"—nonsensical random words—to distract and confuse an NLP algorithm. Humans know how to ignore noise and filter out distractors, but machines sometimes have trouble with this.

Robust NLP is the study of approaches and techniques for building machines

that are smart enough to handle unusual text from diverse sources.^[287] In fact, research into robust NLP may uncover paths toward artificial general intelligence. Humans are able to learn new words and concepts from just a few examples. And we generalize well, not too much and not too little. Machines need a little help. And if you can figure out the "secret sauce" that makes us humans good at this, then you can encode it into your NLP pipelines.

One popular technique for increasing the robustness of neural networks is *random dropout*. *Random dropout*, or just *dropout*, has become popular because of its ease and effectiveness. Your neural networks will almost always benefit from a dropout layer. A dropout layer randomly hides some of the neurons outputs from the neurons listening to them. This causes that pathway in your artificial brain to go quiet and forces the other neurons to learn from the particular examples that are in front of it during that dropout.

It's counter-intuitive, but dropout helps your neural network to spread the learning around. Without a dropout layer, your network will focus on the words and patterns and convolutional filters that helped it achieve the greatest accuracy boost. But you need your neurons to diversify their patterns so that your network can be "robust" to common variations on natural language text.

The best place in your neural network to install a dropout layer is close to the end, just before you run the fully connected linear layer that computes the predictions on a batch of data. This vector of weights passing into your linear layer are the outputs from your CNN and pooling layers. Each one of these values represents a sequence of words, or patterns of meaning and syntax. By hiding some of these patterns from your prediction layer, it forces your prediction layer to diversify its "thinking." Though your software isn't really thinking about anything, it's OK to anthropomorphize it a bit, if it helps you develop intuitions about why techniques like random dropout can improve your model's accuracy.

[285] "A Unified Architecture for Natural Language Processing" by Ronan Collobert and Jason Weston
https://thetalkingmachines.com/sites/default/files/2018-12/unified_nlp.pdf

[286] International Association of Facilitators Handbook, <http://mng.bz/2aOa>

[287] Robin Jia's thesis on Robust NLP (https://robinjia.github.io/assets/pdf/robinjia_thesis.pdf) and his presentation with Kai-Wei Chang, He He and Sameer Singh (<https://robustnlp-tutorial.github.io>)

7.6 PyTorch CNN to process disaster tweets

Now comes the fun part. You are going to build a real world CNN that can distinguish real world news from sensationalism. Your model can help you filter out Tweets about the culture wars so you can focus on news from real war zones.

First you will see where your new convolution layers fit into the pipeline. Then you'll assemble all the pieces to train a CNN on a dataset of "disaster tweets." And if doom scrolling and disaster is not your thing, the CNN is easily adaptable to any labeled dataset of tweets. You can even pick a hashtag that you like and use that as your target label. Then you can find tweets that match that hashtag topic even when the tweeter doesn't know how to use hashtags.

7.6.1 Network architecture

Here are the processing steps and the corresponding shapes of the tensors for each stage of a CNN NLP pipeline. It turns out one of the trickiest things about building a new CNN is keeping track of the shapes of your tensors. You need to ensure that the shape of the outputs of one layer match the shape of the inputs for the next layer will be the same for this example as for previous examples.

1. Tokenization $\Rightarrow (N_{_}, \)$
2. Padding $\Rightarrow (N,)$
3. Embedding $\Rightarrow (M, N)$
4. Convolution(s) $\Rightarrow (M, N - K)$
5. Activation(s) $\Rightarrow (M, N - K)$
6. Pooling(s) $\Rightarrow (M, N - K)$

7. Dropout (optional) $\Rightarrow (M, N - K)$
8. Linear combination $\Rightarrow (L,)$
9. Argmax, softmax or thresholding $\Rightarrow (L,)$

And

- $N_{_}$ is the number of tokens in your input text.
- N is the number of tokens in your padded sequences.
- M is the number of dimensions in your word embeddings.
- K is the size of your kernel.
- L is the number of class labels or values you want to predict.

Your PyTorch model for a CNN has a few more hyperparameters than you had in chapters 5 and 6. However, just as before, it's a good idea to set up your hyperparameters within the *init* constructor of your `CNNTextClassifier` model.

Listing 7.25. CNN hyperparameters

```
class CNNTextClassifier(nn.Module):

    def __init__(self, embeddings):
        super().__init__()

        self.seq_len = 40                         #1
        self.vocab_size = 10000                     #2
        self.embedding_size = 50                   #3
        self.out_channels = 5                      #4
        self.kernel_lengths = [2, 3, 4, 5, 6]      #5
        self.stride = 1                            #6
        self.dropout = nn.Dropout(0)                #7
        self.pool_stride = self.stride             #8
        self.conv_out_seq_len = calc_out_seq_len(
            seq_len=self.seq_len,
            kernel_lengths=self.kernel_lengths,
            stride=self.stride,
        )
```

Just as for your hand-crafted convolutions earlier in this chapter, the sequence length is reduced by each convolutional operation. And the amount of shortening depends on the size of the kernel and the stride. The PyTorch documentation for a `Conv1d` layer provides this formula and a detailed

explanation of the terms.[\[288\]](#)

```
def calc_conv_out_seq_len(seq_len, kernel_len,
                           stride=1, dilation=1, padding=0):
    """
    L_out = (L_in + 2 * padding - dilation * (kernel_size - 1
        1 + _____
                           stride
    """
    return (
        1 + (seq_len +
              2 * padding - dilation * (kernel_len - 1) - 1
              ) //
        stride
    )
```

Your first CNN layer is an `nn.Embedding` layer that converts a sequence of word id integers into a sequence of embedding vectors. It has as many rows as you have unique tokens in your vocabulary (including the new padding token). And it has a column for each dimension of the embedding vectors. You can load these embedding vectors from GloVe or any other pretrained embeddings.

Listing 7.26. Initialize CNN embedding

```
self.embed = nn.Embedding(
    self.vocab_size,                                     #1
    self.embedding_size,                                 #2
    padding_idx=0)
state = self.embed.state_dict()
state['weight'] = embeddings                         #3
self.embed.load_state_dict(state)
```

Next you want to build the convolution and pooling layers. The output size of each convolution layer can be used to define a pooling layer whose kernel takes up the entire convolutional layer output sequence. This is how you accomplish "global" max pooling in PyTorch to produce a single maximum value for each convolutional filter (kernel) output. This is what NLP experts like Christopher Manning and Yoon Kim do in the research papers of theirs that achieved state-of-the-art performance.[\[289\]](#)[\[290\]](#)

Listing 7.27. Construct convolution and pooling layers

```

self.convolvers = []
self.poolers = []
total_out_len = 0
for i, kernel_len in enumerate(self.kernel_lengths):
    self.convolvers.append(
        nn.Conv1d(in_channels=self.embedding_size,
                  out_channels=self.out_channels,
                  kernel_size=kernel_len,
                  stride=self.stride))
    print(f'conv[{i}].weight.shape: {self.convolvers[-1].weight.s')
    conv_output_len = calc_conv_out_seq_len(
        seq_len=self.seq_len, kernel_len=kernel_len, stride=self.
    print(f'conv_output_len: {conv_output_len}')
    self.poolers.append(
        nn.MaxPool1d(kernel_size=conv_output_len, stride=self.str
    total_out_len += calc_conv_out_seq_len(
        seq_len=conv_output_len, kernel_len=conv_output_len,
        stride=self.stride)
    print(f'total_out_len: {total_out_len}')
    print(f'poolers[{i}]: {self.poolers[-1]}')
print(f'total_out_len: {total_out_len}')
self.linear_layer = nn.Linear(self.out_channels * total_out_len,
print(f'linear_layer: {self.linear_layer}')

```

Unlike the previous examples, you're going to now create multiple convolution and pooling layers. For this example we won't layer them up as is often done in computer vision. Instead you will concatenate the convolution and pooling outputs together. This is effective because you've limited the dimensionality of your convolution and pooling output by performing global max pooling and keeping the number of output channels much smaller than the number of embedding dimensions.

You can use print statements to help debug mismatching matrix shapes for each layer of your CNN. And you want to make sure you don't unintentionally create too many trainable parameters that cause more overfitting than you'd like: Your pooling outputs each contain a sequence length of 1, but they also contain 5 channels for the embedding dimensions combined together during convolution. So the concatenated and pooled convolution output is a 5x5 tensor which produces a 25-D linear layer for the output tensor that encodes the meaning of each text.

Listing 7.28. CNN layer shapes

```
conv[0].weight.shape: torch.Size([5, 50, 2])
conv_output_len: 39
total_pool_out_len: 1
poolers[0]: MaxPool1d(kernel_size=39, stride=1, padding=0, dilation=1,
    ceil_mode=False)
conv[1].weight.shape: torch.Size([5, 50, 3])
conv_output_len: 38
total_pool_out_len: 2
poolers[1]: MaxPool1d(kernel_size=38, stride=1, padding=0, dilation=1,
    ceil_mode=False)
conv[2].weight.shape: torch.Size([5, 50, 4])
conv_output_len: 37
total_pool_out_len: 3
poolers[2]: MaxPool1d(kernel_size=37, stride=1, padding=0, dilation=1,
    ceil_mode=False)
conv[3].weight.shape: torch.Size([5, 50, 5])
conv_output_len: 36
total_pool_out_len: 4
poolers[3]: MaxPool1d(kernel_size=36, stride=1, padding=0, dilation=1,
    ceil_mode=False)
conv[4].weight.shape: torch.Size([5, 50, 6])
conv_output_len: 35
total_pool_out_len: 5
poolers[4]: MaxPool1d(kernel_size=35, stride=1, padding=0, dilation=1,
    ceil_mode=False)
total_out_len: 5
linear_layer: Linear(in_features=25, out_features=1, bias=True)
```

And the end result is a rapidly overfitting language model and text classifier. Your model achieves a maximum test accuracy of 73% at epoch 55 and a maximum training set accuracy of 81% at the last epoch, epoch 75. You can accomplish even more overfitting by increasing the number of channels for the convolutional layers. You usually want to ensure your first training runs accomplish overfitting to ensure all your layers are configured correctly and to set an upper bound on the accuracy that is achievable on a particular problem or dataset.

```
Epoch: 1, loss: 0.76782, Train accuracy: 0.59028, Test accuracy: 0.59028
Epoch: 2, loss: 0.64052, Train accuracy: 0.65947, Test accuracy: 0.65947
Epoch: 3, loss: 0.51934, Train accuracy: 0.68632, Test accuracy: 0.68632
...
Epoch: 55, loss: 0.04995, Train accuracy: 0.80558, Test accuracy: 0.73000
Epoch: 65, loss: 0.05682, Train accuracy: 0.80835, Test accuracy: 0.73000
Epoch: 75, loss: 0.04491, Train accuracy: 0.81287, Test accuracy: 0.73000
```

By reducing the number of channels from 5 to 3 for each embedding you can reduce the total output dimensionality from 25 to 15. This will limit the overfitting but reduce the convergence rate unless you increase the learning coefficient:

```
Epoch:  1, loss: 0.61644, Train accuracy: 0.57773, Test accuracy:  
Epoch:  2, loss: 0.52941, Train accuracy: 0.63232, Test accuracy:  
Epoch:  3, loss: 0.45162, Train accuracy: 0.67202, Test accuracy:  
...  
Epoch: 55, loss: 0.21011, Train accuracy: 0.79200, Test accuracy:  
Epoch: 65, loss: 0.21707, Train accuracy: 0.79434, Test accuracy:  
Epoch: 75, loss: 0.20077, Train accuracy: 0.79784, Test accuracy:
```

7.6.2 Pooling

Pooling aggregates the data from a large tensor to compress the information into fewer values. This is often called a "reduce" operation in the world of "Big Data" where the map-reduce software pattern is common. Convolution and pooling lend themselves well to the map-reduce software pattern and can be parallelized within a GPU automatically using PyTorch. You can even use multi-server HPC (high performance computing) systems to speed up your training. But CNNs are so efficient, you aren't likely to need this kind of horsepower.

All the statistics you're used to calculating on a matrix of data can be useful as pooling functions for CNNs:

- min
- max
- std
- sum
- mean

The most common and most successful aggregations

7.6.3 Linear layer

The concatenated encodings approach gave you a lot of information about each microblog post. The encoding vector had 1856 values. The largest word

vectors you worked with in chapter 6 were 300 dimensions. And all you really want for this particular pipeline is the binary answer to the question "is it news worthy or not?"

Do you remember in chapter 6 how you had to when you were trying to get a neural network to predict "yes or no" questions about the occurrence or absence of particular words? Even though you didn't really pay attention to the answer to all those thousands of questions (one for each word in your vocabulary), it was the same problem you have now. So you can use the same approach, a `torch.nn.Linear` layer will optimally combine all the pieces of information together from a high dimensional vector to answer whatever question you pose it.

So you need to add a Linear layer with as many weights as you have encoding dimensions that are being output from your pooling layers.

Listing 7.26 shows the code you can use to calculate the size of the linear layer.

Listing 7.29. Compute the tensor size for the output of a 1D convolution

```
out_pool_total = 0
for kernel_len, stride in zip(kernel_lengths, strides):
    out_conv = (
        (in_seq_len - dilation * (kernel_len - 1) - 1) // stride)
    out_pool = (
        (out_conv - dilation * (kernel_len - 1) - 1) // stride) +
    out_pool_total += out_pool
```

7.6.4 Getting fit

Before you can train your CNN you need to tell it how to adjust the weights (parameters) with each batch of training data. You need to compute two pieces, the slopes of the weights relative to the loss function (the gradient) and an estimate of how far to try to descend that slope (the learning rate). For the single-layer perceptrons and even the logistic regressions of the previous chapters you were able to get away with using some general purpose optimizers like "Adam." And you can often set the learning rate to a fixed value for CNNs And those will work well for CNNs too. However, if you

want to speed up your training you can try to find an optimizer that's a bit more clever about how it adjusts all those parameters of your model. Geoffrey Hinton called this approach "rmsprop" because he uses the root mean square (RMS) formula to compute the moving average of the recent gradients. RMSprop aggregates an exponentially decaying window of the weights for each batch of data to improve the estimate of the parameter gradient (slopes) and speed up learning.[\[291\]](#) [\[292\]](#) It is usually a good bet for backpropagation within a convolutional neural network for NLP.

7.6.5 Hyperparameter Tuning

Explore the hyperparameter space to see if you can beat my performance. Fernando Lopez and others have achieved 80% validation and test set accuracy on this dataset using 1-D convolution. There's likely a lot of room to grow.

The nlpia2 package contains a command line script that accepts arguments for many of the hyperparameters you might want to adjust. Give it a try and see if you can find a more fertile part of the hyperspace universe of possibilities. You can see my latest attempt in listing 7.27

Listing 7.30. Command line script for optimizing hyperparameters

```
python train.py --dropout_portion=.35 --epochs=16 --batch_size=8
Epoch: 1, loss: 0.44480, Train accuracy: 0.58152, Test accuracy:
Epoch: 2, loss: 0.27265, Train accuracy: 0.63640, Test accuracy:
...
Epoch: 15, loss: 0.03373, Train accuracy: 0.83871, Test accuracy:
Epoch: 16, loss: 0.09545, Train accuracy: 0.84718, Test accuracy:
```

Did you notice the `win=True` flag in listing 7.27? That is an Easter Egg or cheat code I created for myself within my CNN pipeline. Whenever I discover a winning ticket in the "Lottery Ticket Hypothesis" game, I hard code it into my pipeline. In order for this to work, you have to keep track of the random seeds you use and the exact dataset and software you are using. If you can recreate all of these pieces, it's usually possible to recreate a particularly lucky "draw" to build on and improve later as you think of new architecture or parameter tweaks.

In fact, this winning random number sequence initialized the weights of the model so well that the test accuracy started off better than the training set accuracy. It took 8 epochs for the training accuracy to overtake the test set accuracy. After 16 passes through the dataset (epochs), the model is fit 5% better to the training set than the test set.

If you want to achieve higher test set accuracy and reduce the overfitting, you can try adding some regularization or increasing the amount of data ignored within the Dropout layer. For most neural networks, dropout ratios of 30% to 50% often work well to prevent overfitting without delaying the learning too long. A single-layer CNN doesn't benefit much from dropout ratios above 20%.

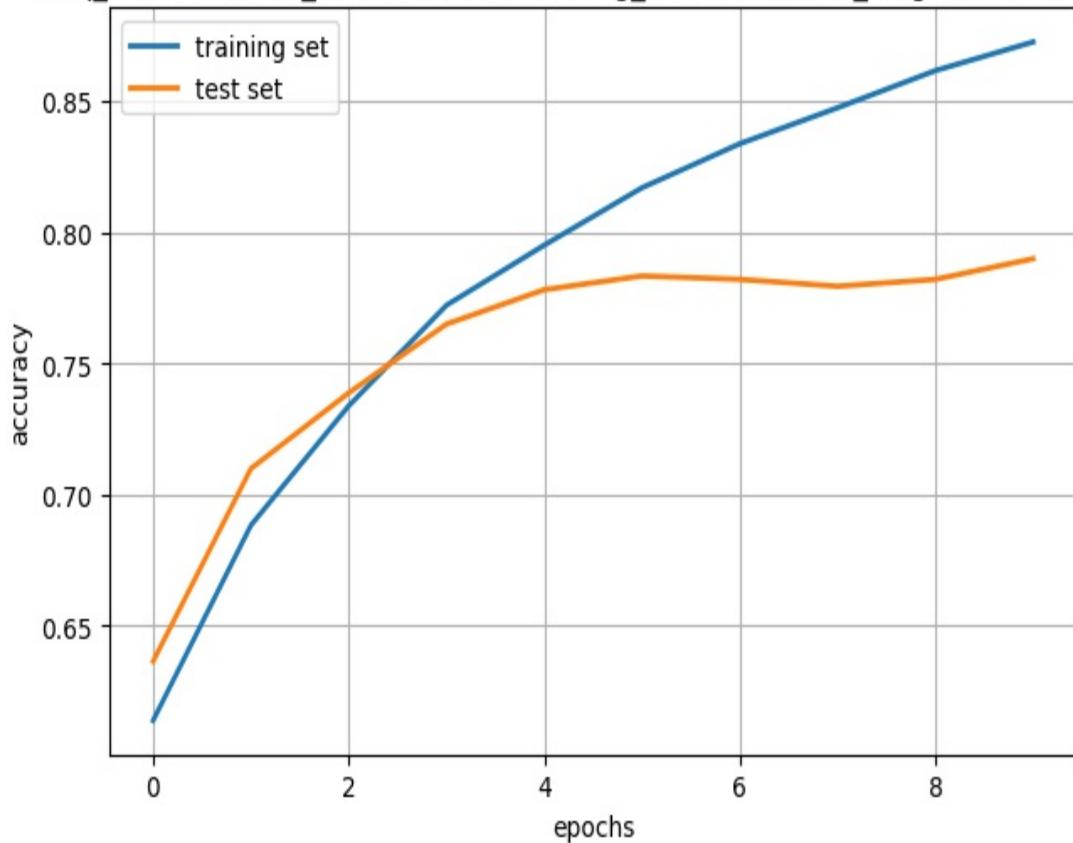
Listing 7.31. CNN hyperparameter tuning

| kernel_sizes | learning_rate | seq_len | case_sens | vocab_size | dropout | training_accuracy | test_accuracy |
|---------------|---------------|---------|-----------|------------|---------|-------------------|---------------|
| [2] | 0.0010 | 32 | False | 2000 | NaN | 0.5790 | 0.545 |
| [1 2 3 4 5 6] | 0.0010 | 40 | False | 2000 | NaN | 0.7919 | 0.710 |
| [2 3 4 5] | 0.0015 | 40 | False | 2000 | NaN | 0.8038 | 0.715 |
| [1 2 3 4 5 6] | 0.0010 | 40 | True | 2000 | NaN | 0.7685 | 0.752 |
| [2] | 0.0010 | 32 | True | 2000 | 0.2 | 0.8472 | 0.753 |
| [2 3 4 5] | 0.0010 | 32 | True | 2000 | 0.2 | 0.8727 | 0.790 |

Can you find a better combination of hyperparameters to improve this model's accuracy? Don't expect to achieve much better than 80% test set accuracy, because this is a hard problem. Even human readers can't reliably tell if a tweet represents a factual news-worthy disaster or not. After all, other humans (and bots) are composing these tweets in an attempt to fool readers. This is an adversarial problem. Even a small 1-layer CNN does a decent job.

Figure 7.17. Learning curve for the best hyperparamters we found

seq_len=32 vocab_size=2000 embedding_size=64 kernel_lengths=[2, 3, 4, 5]



The key to hyperparameter tuning is to conscientiously record each experiment and make thoughtful decisions about the hyperparameter adjustments you make for the next experiment. You can automate this decisionmaking with a Bayesian Optimizer. But in most cases you can develop your intuition and accomplish faster tuning of your hyperparameters if you use your biological neural network to accomplish the Bayesian optimization. And if you are curious about the affect of the transpose operation on the embedding layer, you can try it both ways to see which works best on your problem. But you probably want to follow the experts if you want to get state-of-the-art results on hard problems. Don't believe everything you read on the Internet, especially when it comes to CNNs for NLP.

[288] (<https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html>)

[289] Conv Nets for NLP by Christopher Manning (<http://mng.bz/1Meq>)

[290] "A Sensitivity Analysis of CNNs for Sentence Classification" by Ye Zhang and Brian Wallace (<https://arxiv.org/pdf/1510.03820.pdf>)

[291] Slide 14 "Four ways to speed up machine learning" from "Overview of mini-batch gradient descent" by Hinton
(https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf)

[292] Ph D thesis "Optimizing Neural Networks that Generate Images" by Tijmen Tieleman (https://www.cs.toronto.edu/~tijmen/tijmen_thesis.pdf)

7.7 Review

1. For a length 3 kernel and an input array of length 8 what is the length of the output?
2. What is the kernel for detecting an "S O S" distress signal (Save Our Souls, or Save Our Ship) within the secret message audio file used in this chapter?
3. What is the best training set accuracy you can achieve after tuning the hyperparameters for the news-worthiness microblog post problem?
4. How would you extend the model to accommodate an additional class? The news.csv file, provided in the nlpia2 package on gitlab contains famous quotes to give you another level of profundity to attempt to classify with your CNN.
5. Write 3 kernels, one each for detecting dots, dashes, and pauses. Write a pooling function that *counts* unique occurrences of these symbols.
BONUS: Create a system of functions that *translates* the secret message audio file into the symbols ". .", "- -", and " _".
6. Find some hyperparameters (don't forget about random seeds) that achieve better than 80% accuracy on the test set for the disaster tweets dataset.
7. Create a sarcasm detector using a word-based CNN using datasets and examples on Hugging Face (huggingface.co). Is it credible that several published papers claim 91% accuracy at detecting sarcasm from a single tweet, without context? [293] [294].

[293] 92% is claimed by Ivan Helin for their model on Hugging Face (<https://huggingface.co/helinivan/english-sarcasm-detector>)

[294] 91% is claimed in "A Deeper Look into Sarcastic Tweets Using a CNN" by Soujanya Poria et al. (<https://arxiv.org/abs/1610.08815>)

7.8 Summary

- A convolution is a windowed filter that slides over your sequence of words to compress it's meaning into an encoding vector.
- Hand-crafted convolutional filters work great on predictable signals such as Morse code, but you will need CNNs that learn their own filters for NLP.
- Neural networks can extract patterns in a sequence of words that other NLP approaches would miss.
- During training, if you sandbag your model a bit with a dropout layer you can keep it from overachieving (over fitting) on your training data.
- Hyperparameter tuning for neural networks gives you more room to exercise your creativity than conventional machine learning models.
- You can outperform 90% of bloggers at NLP competitions if your CNNs align the embedding dimension with the convolutional channels.
- Old-fashioned CNNs may surprise you with their efficiency at solving hard problems such as detecting newsworthy tweets.

8 Reduce, reuse, recycle your words (RNNs and LSTMs)

This chapter covers

- Unrolling recursion so you can understand how to use it for NLP
- Implementing word and character-based RNNs in PyTorch
- Identifying applications where RNNs are your best option
- Re-engineering your datasets for training RNNs
- Customizing and tuning your RNN structure for your NLP problems
- Understanding backprop (backpropagation) in time
- Combining long and short term memory mechanisms to make your RNN smarter

An *RNN* (Recurrent Neural Network) recycles tokens. Why would you want to recycle and reuse your words? To build a more sustainable NLP pipeline of course! ;) *Recurrence* is just another word for recycling. An RNN uses recurrence to allow it to remember the tokens it has already read and reuse that understanding to predict the target variable. And if you use RNNs to predict the next word, RNNs can generate, going on and on and on, until you tell them to stop. This sustainability or regenerative ability of RNNs is their super power.

It turns out that your NLP pipeline can predict the next tokens in a sentence much better if it remembers what it has already read and understood. But, wait, didn't a CNN "remember" the nearby tokens with a kernel or filter of weights? It did! But a CNN can only *remember* a limited window, that is a few words long. By recycling the machine's understanding of each token before moving to the next one, an RNN can remember something about *all* of the tokens it has read. This makes your machine reader much more sustainable, it can keep reading and reading and reading...for as long as you like.

But wait, isn't recursion dangerous? If that's the first thought that came to

you when you read recurrence, you're not alone. Anyone who has taken an algorithms class has probably broken a function, an entire program, or even taken down an entire web server, but using recurrence the wrong way. The key to doing recurrence correctly and safely is that you must always make sure your algorithm is *reducing* the amount of work it has to do with each recycling of the input. This means you need to delete something from the input before you call the function again with that input. For your NLP RNN this comes naturally as you *pop* (remove) a token off of the *stack* (the text string) before you feed that input back into your network.



Note

Technically "recurrence" and "recursion" are two different things. [\[295\]](#) But most mathematicians and computer scientists use both words to explain the same concept - recycling a portion of the output back into the input to perform an operation repeatedly in sequence. [\[296\]](#) But as with all natural language words, the concepts are fuzzy and it can help to understand them both when building *Recurrent Neural Networks*. As you'll see in the code for this chapter, an RNN doesn't have a function that calls itself recursively the way you normally think of recursion. The `.forward(x)` method is called in a `for` loop that is outside of the RNN itself.

RNNs are *neuromorphic*. This is a fancy way of saying that researchers are mimicing how they think brains work when they design artificial neural nets such as RNNs. You can use what you know about how your own brain works to come up with ideas for how to process text with artificial neurons. And your brain is recurrently processing the tokens that you are reading right now. So recurrence must be a smart, efficient way to use your brain resources to understand text.

As you read this text you are recycling what you already know about the previous words before updating your prediction of what's going to happen next. And you don't stop predicting until you reach the end of a sentence or paragraph or whatever you're trying to understand. Then you can pause at the end of a text and process all of what you've just read. Just like the RNNs in this chapter, the RNN in your brain uses that pause at the end to encode,

classify, and *get something out* of the text. And because RNNs are always predicting, you can use them to predict words that your NLP pipeline should say. So RNNs are great not only for reading text, but also for tagging and writing text.

RNNs are a game changer for NLP. They have spawned an explosion of practical applications and advancements in deep learning and AI.

8.1 What are RNNs good for?

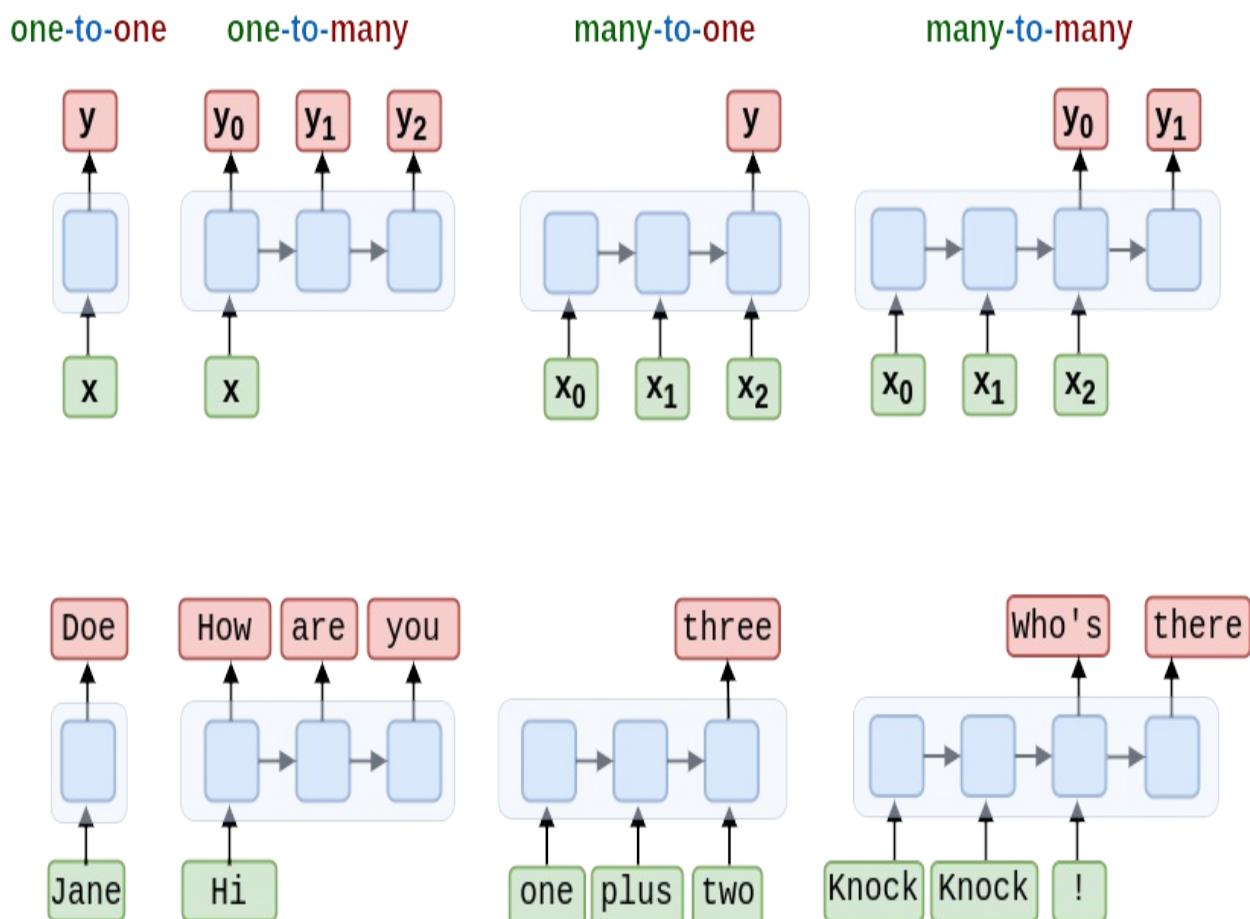
The previous deep learning architectures you've learned about are great for processing short bits of text - usually individual sentences. RNNs promise to break through that text length barrier and allow your NLP pipeline to ingest an infinitely long sequence of text. And not only can they process unending text, they can *generate* text for as long as you like. RNNs open up a whole new range of applications like generative conversational chatbots and text summarizers that combine concepts from many different places within your documents.

| Type | Description | Applications |
|--------------|---|--|
| One to Many | One input tensor used to generate a sequence of output tensors | Generate chat messages, answer questions, describe images |
| Many to One | sequence of input tensors gathered up into a single output tensor | Classify or tag text according to its language, intent, or other characteristics |
| Many to Many | a sequence of input tensors used to generate | Translate, tag, or anonymize the tokens within a sequence of |

| | |
|------------------------------|---|
| a sequence of output tensors | tokens, answer questions, participate in a conversation |
|------------------------------|---|

This is the superpower of RNNs, they process sequences of tokens or vectors. You are no longer limited to processing a single, fixed length vector. So you don't have to truncate and pad your input text to make your round text the right shape to fit into a square hole. And an RNN can generate text sequences that go on and on forever, if you like. You don't have to stop or truncate the output at some arbitrary maximum length that you decide ahead of time. Your code can dynamically decide when enough is enough.

Figure 8.1. Recycling tokens creates endless options



You can use RNNs to achieve state of the art performance on many of the

tasks you're already familiar with, even when your text is shorter than infinity ;).

- translation
- summarization
- classification
- question answering

And RNNs are one of the most efficient and accurate ways to accomplish some new NLP tasks that you will learn about in this chapter:

- generating new text such as paraphrases, summaries or even answers to questions
- tagging individual tokens
- diagramming the grammar of sentences like you did in English class
- creating language models that predict the next token

If you read through the RNNs that are the top of the leader board on Papers with Code [\[297\]](#) you can see that RNNs are the most efficient approach for many applications.

RNNs aren't just for researchers and academics. Let's get real. In the real world, people are using RNNs to:

- spell checking and correction
- autocomplete of natural language or programming language expressions
- classify sentences for grammar checking or FAQ chatbots
- classify questions or generate answers to those questions
- generate entertaining conversational text for chatbots
- named entity recognition (NER) and extraction
- classify, predict, or generate names for people, babies, and businesses
- classify or predict subdomain names (for security vulnerability scanning)

You can probably guess what most of those applications are about, but you're probably curious about that last one (subdomain prediction). A subdomain is that first part of a domain name in a URL, the `www` in `www` in `www.lesswrong.com` or `en` in `en.wikipedia.org`. Why would anyone would

want to predict or guess subdomains? Dan Meisler did a talk on the critical role that subdomain guessers play in his cybersecurity toolbox.[\[298\]](#) Once you know a subdomain, a hacker or pentester can scan the domain to find vulnerabilities in the server security.

And once you will soon be comfortable using RNNs to generate completely new words, phrases, sentences, paragraphs and even entire pages of text. It can be so much fun playing around with RNNs that you could find yourself accidentally creating applications that open up opportunities for completely new businesses.

- suggest company, product or domain names [\[299\]](#)
- suggest baby names
- sentence labeling and tagging
- autocomplete for text fields
- paraphrasing and rewording sentences
- inventing slang words and phrases

8.1.1 RNNs remember everything you tell them

Have you ever accidentally touched wet paint and found yourself "reusing" that paint whenever you touched something? And as a child you might have fancied yourself an impressionistic painter as you shared your art with the world by finger painting the walls around you. You're about to learn how to build a more mindful impressionistic word painter. In chapter 7 you imagined a lettering stencil as an analogy for processing text with CNNs. Well now, instead of sliding a word stencil across the words in a sentence your going roll a paint roller across them... while they're still wet!

Imagine painting the letters of a sentence with slow-drying paint and laying it on thick. And let's create a diverse rainbow of colors in your text. Maybe you're even supporting LBGTQ pride week by painting the crosswalks and bike lanes in North Park.

Figure 8.2. A rainbow of meaning

Wet Paint!

Now, pick up a clean paint roller and roll it across the letters of the sentence from the beginning of the sentence to the end. Your roller would pick up the paint from one letter and recycle it to lay it back down on top of the previous letters. Depending on how big your roller is, a small number of letters (or parts of letters) would be rolled on top of letters to the right. All the letters after the first one would be smeared together to create a smudgy stripe that only vaguely resembles the original sentence.

Figure 8.3. Pot of gold at the end of the rainbow



The smudge gathers up all the paint from the previous letters into a single compact representation of the original text. But is a useful, meaningful representation? For a human reader all you've done is create a multicolored mess. It wouldn't communicate much meaning to the humans reading it. This is why humans don't use this *representation* of the meaning of text for themselves. However, if you think about the smudge of characters you might be able to imagine how a machine might interpret it. And for a machine it is certainly much more dense and compact than the original sequence of characters.

In NLP we want to create compact, dense vector representations of text. Fortunately, that representation we're looking for is hidden on your paint

roller! As your fresh clean roller got smeared with the letters of your text it gathered up a *memory* of all the letters you rolled it across. This is analogous to the word embeddings you created in chapter 6. But this embedding approach would work on much longer pieces of text. You could keep rolling the roller forever across more and more text, if you like, squeezing more and more text into the compact representation.

In previous chapters your tokens were mostly words or word n-grams. You need to expand your idea of a token to include individual characters. The simplest RNNs use characters rather than words as the tokens. This is called a character-based RNN. Just as you had word and token embeddings in previous chapters you can think of characters too has having meaning. Now does it make more sense how this smudge at the end of the "Wet Paint!" lettering represents an embedding of all the letters of the text?

One last imaginary step might help you bring out the hidden meaning in this thought experiment. In your mind, check out that embedding on your paint roller. In your mind roll it out on a fresh clean piece of paper. Keep in mind the paper and your roller only big enough to hold a single letter. That will *output* a compact representation of the paint roller's memory of the text. And that output is hidden inside your roller until you decide to use it for something. That's how the text embeddings work in an RNN. The embeddings are *hidden* inside your RNN until you decide to output them or combine them with something else to reuse them. In fact this vector representation of your text is stored in a variable called `hidden` in many implementations of RNNs.



Important

RNN embeddings are different from the word and document embeddings you learned about in chapter 6 and 7. An RNN is gathering up meaning over time or text position. An RNN encodes meaning into this vector for you to reuse with subsequent tokens in the text. This is like the Python `str.encode()` function for creating a multi-byte representation of unicode text characters. The order that the sequence of tokens is processed matters a lot to the end result, the encoding vector. So you probably want to call RNN embeddings "encodings" or "encoding vectors" or "encoding tensors." This vocabulary

shift was encouraged by Garrett Lander on a project to do NLP on extremely long and complex documents, such as patient medical records or The Meuller Report.[\[300\]](#) This new vocabulary made it a lot easier for his team to develop a shared mental model of the NLP pipeline.

Keep your eye out for the hidden layer later in this chapter. The activation values are stored in the variable `h` or `hidden`. These activation values within this tensor are your embeddings up to that point in the text. It's overwritten with new values each time a new token is processed as your NLP pipeline is gathering up the meaning of the tokens it has read so far. In figure 8.4 you can see how this blending of meaning in an embedding vector is much more compact and blurry than the original text.

Figure 8.4. Gather up meaning into one spot



You could read into the paint smudge something of the meaning of the original text, just like in a Rorschach inkblot test. Rorschach inkblots are smudges of ink or paint on flashcards used to spark people's memories and test their thinking or mental health.[\[301\]](#) Your smudge of paint from the paint roller is a vague, impressionistic representation of the original text. And it's a much more compact representation of the text. This is exactly what you were trying to achieve, not just creating a mess. You could clean your roller, rinse and repeat this process on a new line of text to get a different smudge with different *meaning* for your neural network. Soon you'll see how each of these steps are analogous to the actual mathematical operations going on in an RNN layer of neurons.

Your paint roller has smeared many of the letters at the end of the sentence so that the last exclamation point at the end is almost completely unintelligible.

But that unintelligible bit at the end is exactly what your machine needs to understand the entire sentence within the limited surface area of the paint roller. You have smudged all the letters of the sentence together onto the surface of your roller. And if you want to see the message embedded in your paint roller, you just roll it out onto a clean piece of paper.

In your RNN you can accomplish this by outputting the hidden layer activations after you've rolled your RNN over the tokens of some text. The encoded message probably won't say much to you as a human, but it gives your paint roller, the machine, a hint at what the entire sentence said. Your paint roller gathered an impression of the entire sentence. We even use the word "gather" to express understanding of something someone says, as in "I gather from what you just said, that rolling paint rollers over wet paint are analogous to RNNs."

Your paint roller has compressed, or encoded the entire sentence of letters into a short smudgy impressionistic stripe of paint. In an RNN this smudge is a vector or tensor of numbers. Each position or dimension in the encoding vector is like a color in your paint smudge. Each encoding dimension holds an aspect of meaning that your RNN has been designed to keep track of. The impressions that the paint made on your roller (the hidden layer activations) were continuously recycled till you got to the end of the text. And then you reused all those smudges on your roller to create a new impression of the entire sentence.

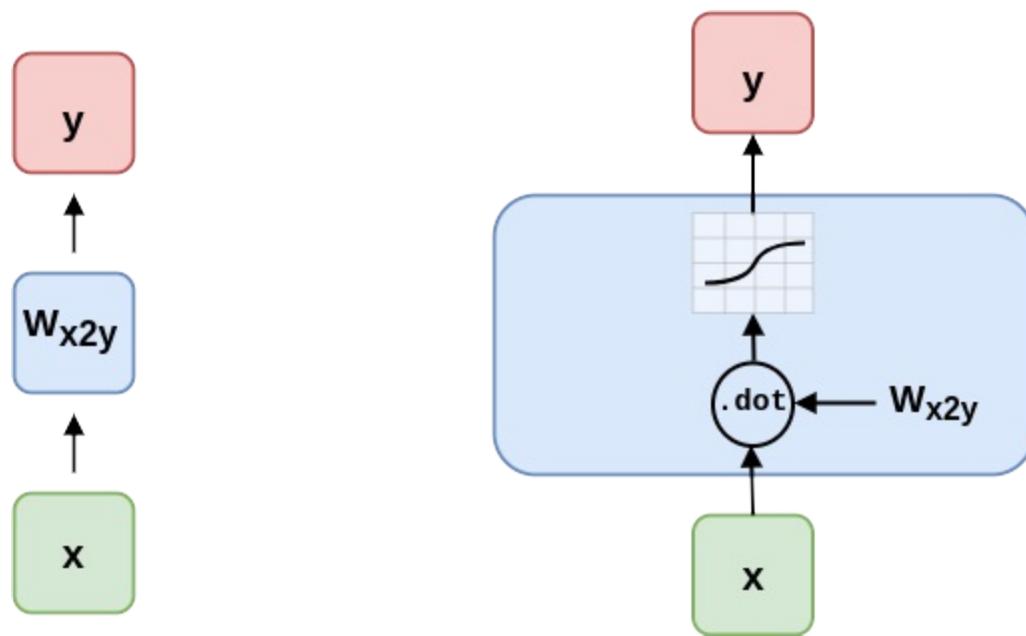
8.1.2 RNNs hide their understanding

The key change for an RNN is that it maintains a hidden embedding by recycling the meaning of each token as it reads them one at a time. This hidden vector of weights contains everything the RNN has understood up to the point in the text it is reading. This means you can't run the network all at once on the entire text you're processing. In previous chapters your model learns a function that maps one input to one output. But, as you'll soon see, an RNN learns a *program* that keeps running on your text until it's done. An RNN needs to read your text one token at a time.

An ordinary feedforward neuron just multiplies the input vector by a bunch

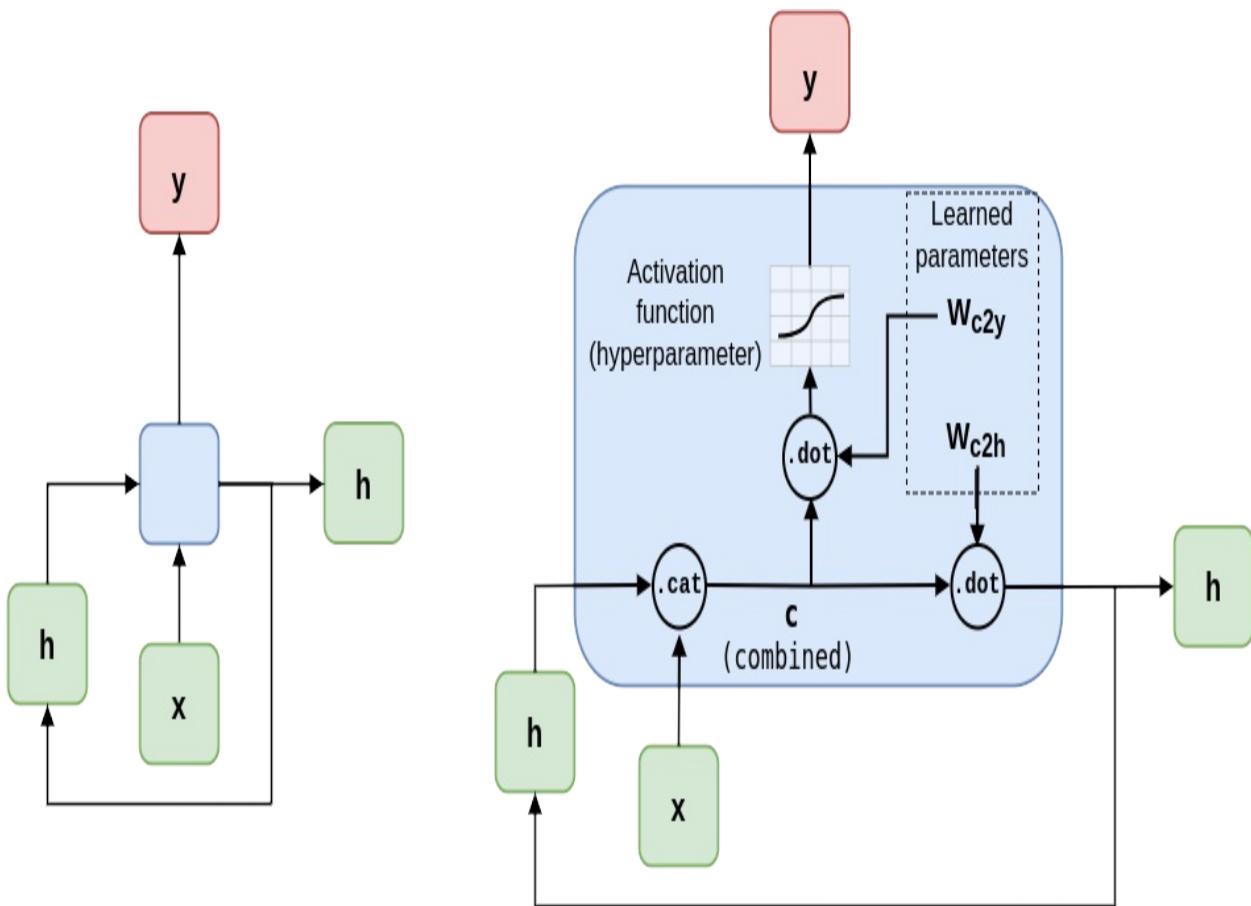
of weights to create an output. No matter how long your text is, a CNN or feedforward neural network will have to do the exact same number of multiplications to compute the output prediction. The neurons of a linear neural network all work together to compose a new vector to represent your text. You can see in Figure 8.5 that a normal feedforward neural network takes in a vector input (x), multiplies it by a matrix of weights (w), applies an activation function, and then outputs a transformed vector (y). Feedforward network layers transform can only transform one vector into another.

Figure 8.5. Ordinary feedforward neuron



With RNNs your neuron never gets to see the vector for the entire text. Instead an RNN must process your text one token at a time. In order to keep track of the tokens it has already read it records a hidden vector (h) that can be passed along to its future self - the exact same neuron that produced the hidden vector in the first place. In computer science terminology this hidden vector is called state. That's why Andrej Karpathy and other deep learning researchers get so excited about the effectiveness of RNNs. RNNs enable machines to finally learn Turing complete programs rather than just isolated functions.[\[302\]](#)

Figure 8.6. A neuron with recurrence



If you unroll your RNN it begins to look a lot like a chain... a Markov Chain, in fact. But this time your window is only one-token wide and you're reusing the output from the previous token, combined with the current token before rolling forward to the next token in your text. Fortunately you started doing something similar to this when you slid the CNN window or kernel across the text in chapter 7.

How can you implement neural network recurrence in Python? Fortunately you don't have to try to wrap around a recursive function call like you may have encountered in coding interviews. Instead, all you have to do is create a variable to store the hidden state separate from the inputs and outputs. And you need to have a separate matrix of weights to use for computing that hidden tensor. [8.1](#) implements a minimal RNN from scratch, without using PyTorch's RNN Module.

Listing 8.1. Recurrence in PyTorch

```

>>> from torch import nn

>>> class RNN(nn.Module):
...
...     def __init__(self,
...                  vocab_size, hidden_size, output_size):      #1
...         super().__init__()
...         self.W_c2h = nn.Linear(
...             vocab_size + hidden_size, hidden_size)      #2
...         self.W_c2y = nn.Linear(vocab_size + hidden_size, output_size)
...         self.softmax = nn.LogSoftmax(dim=1)
...
...     def forward(self, x, hidden):      #3
...         combined = torch.cat((x, hidden), axis=1)      #4
...         hidden = self.W_c2h(combined)      #5
...         y = self.W_c2y(combined)      #6
...         y = self.softmax(y)
...         return y, hidden      #7

```

You can see how this new RNN neuron now outputs more than one thing. Not only do you need to return the output or prediction, but you also need to output the hidden state tensor to be reused by the "future self" neuron.

8.1.3 RNNs remember everything you tell them

To see how RNNs retain a memory of all the tokens of a document you can unroll the neuron diagram in figure 8.7. You create copies of the neuron to show the "future selves" in the `for` loop that is iterating through your tokens. This is like unrolling a `for` loop, when you just copy and paste the lines of code in within the loop the appropriate number of times.

Figure 8.7. Unroll an RNN to reveal its hidden secrets

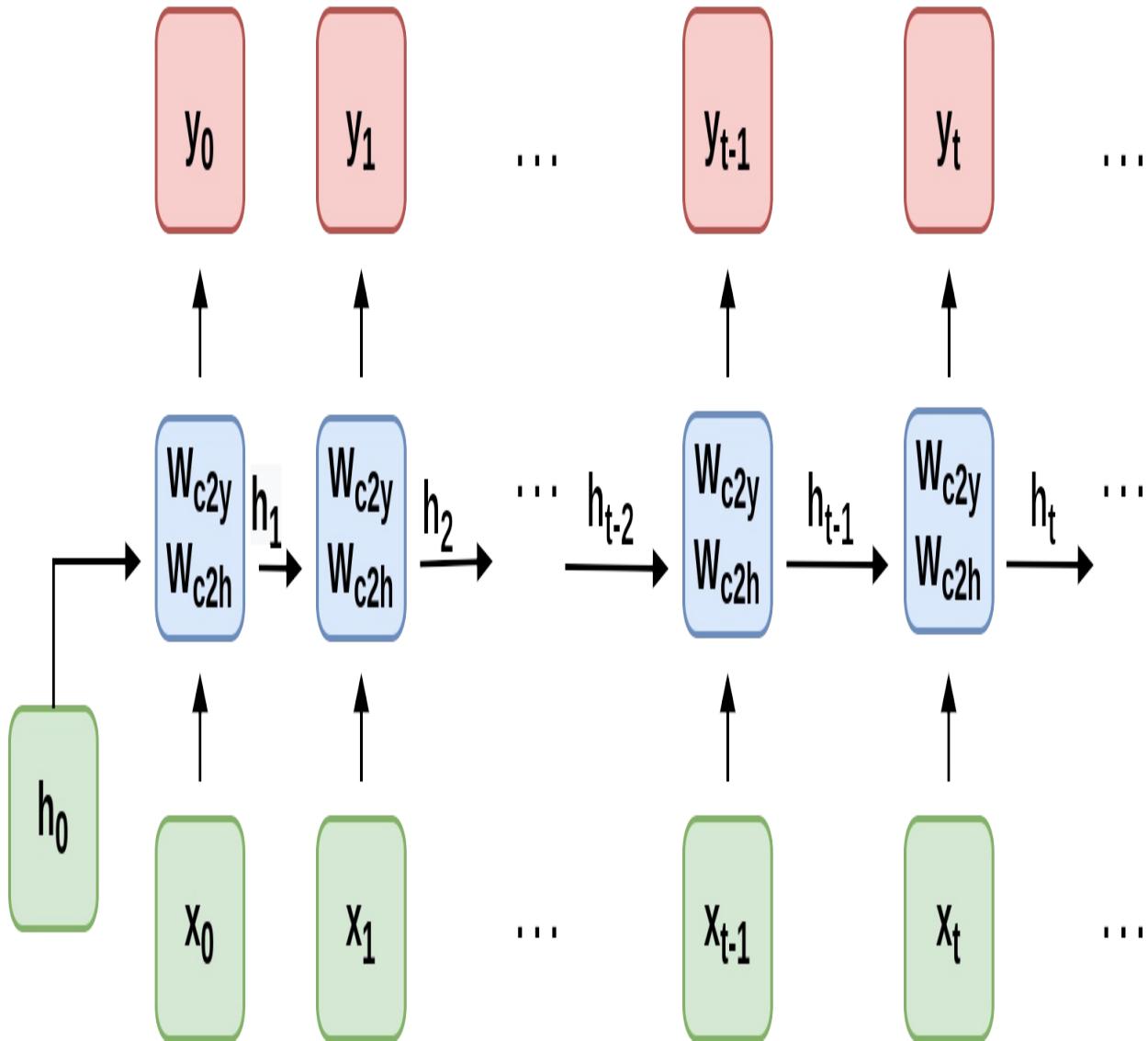


Figure 8.7 shows an RNN passes the hidden state along to the next "future self" neuron, sort of like Olympic relay runners passing the baton. But this baton is imprinted with more and more memories as it is recycled over and over again within your RNN. You can see how the tensors for the input tokens are modified many, many times before the RNN finally sees the last token in the text.

Another nice feature of RNNs is that you can tap into an output tensor anywhere along the way. This means you can tackle challenges like machine translation, named entity recognition, anonymization and deanonymization of text, and even unredaction of government documents. [\[303\]](#)

These two features are what make RNNs unique.

1. You can process as many tokens as you like in one text document.
2. You can output anything you need after each token is processed.

That first feature is not such a big deal. As you saw with CNNs, if you want to process long text, you just need to make room for them in your max input tensor size. In fact, the most advanced NLP models to date, *transformers*, create a max length limit and pad the text just like CNNs.

However, that second feature of RNNs is a really big deal. Imagine all the things you can do with a model that labels each and every token in a sentence. Linguists spend a lot of time diagramming sentences and labeling tokens. RNNs and deep learning have revolutionized the way linguistics research is done. Just look at some of the linguistic features that SpaCy can identify for each word in some example "hello world" text in listing [8.2](#).

Listing 8.2. SpaCy tags tokens with RNNs

```
>>> import pandas as pd
>>> from nlpia2.spacy_language_model import nlp

>>> tagged_tokens = list(nlp('Hello world. Goodbye now!'))
>>> interesting_tags = 'text dep_ head lang_ lemma_ pos_ sentimen
>>> pd.DataFrame([
...     [getattr(t, a) for a in interesting_tags]
...     for t in tagged_tokens],
...     columns=interesting_tags)
   text    dep_    head lang_  lemma_  pos_  sentiment shap
0  Hello    intj    world    en    hello  INTJ      0.0  Xxx
1  world    ROOT    world    en    world  NOUN      0.0   xx
2    .    punct    world    en        .  PUNCT      0.0
3  Goodbye    ROOT  Goodbye    en  goodbye  INTJ      0.0  Xxx
4   now  advmod  Goodbye    en      now    ADV      0.0     x
5     !    punct  Goodbye    en        !  PUNCT      0.0
```

It's all well and good to have all that information - all that output whenever you want it. And you're probably excited to try out RNNs on really long text to see how much it can actually remember.

[\[295\]](#) Mathematics forum StackExchange question about recurrence and

recursion (<https://math.stackexchange.com/questions/931035/recurrence-vs-recursive>)

[296] MIT Open Courseware lectures for CS 6.005 "Software Construction"
(<https://ocw.mit.edu/ans7870/6/6.005/s16/classes/10-recursion/>)

[297] Papers with Code query for RNN applications (<https://proai.org/pwc-rnn>)

[298] Daniel Miessler's Unsupervised Learning podcast #340
(<https://mailchi.mp/danielmiessler/unsupervised-learning-no-2676196>) and
the RNN source code (<https://github.com/JetP1ane/Affinis>)

[299] Ryan Stout's (<https://github.com/ryanstout>) BustAName app
(https://bustaname.com/blog_posts)

[300] Garrett Lander, Al Kari, and Chris Thompson contributed to our project
to unredact the Meuller report (<https://proai.org/unredact>)

[301] Rorsharch test wikipedia article
(https://en.wikipedia.org/wiki/Rorschach_test)

[302] "The unreasonable effectiveness of RNNs"
(<https://karpathy.github.io/2015/05/21/rnn-effectiveness>)

[303] Portland Python User Group presentation on unredacting the Meuller
Report (<https://proai.org/unredact>)

8.2 Predict someone's nationality from only their last name

To get you up to speed quickly on recycling, you'll start with the simplest possible token—the lowly character (letter or punctuation). You are going to build a model that can predict the nationality of last names, also called "surnames" using only the letters in the names to guide the predictions. This kind of model may not sound all that useful to you. You might even be

worried that it could be used to harm individuals from particular cultures.

Like you, the authors' LinkedIn followers were suspicious when we were mentioned we were training a model to predict the demographic characteristics of names. Unfortunately businesses and governments do indeed use models like this to identify and target particular groups of people, often with harmful consequences. But these models can also be used for good. We use them to help our nonprofit and government customers anonymize their conversational AI datasets. Volunteers and open source contributors can then train NLP models from these anonymized conversation datasets to identify healthcare or education content that can be helpful for users, while simultaneously protecting user privacy.

This multilingual dataset will give you a chance to learn how to deal with diacritics and other embellishments that are common for non-English words. To keep it interesting, you will remove these character embellishments and other give-aways in the unicode characters of multilingual text. That way your model can learn the patterns you really care about rather than "cheating" based on this leakage. The first step in processing this dataset is to *asciify* it - convert it to pure ASCII characters. For example, Unicode representation of the Irish name "O'Néàl" has an "accent accute" over the "e" and an "accent grave" over the "a" in the this name. And the apostrophe between the "O" and "N" can be a special directional apostrophe that could unfairly clue your model in to the nationality of the name, if you don't *asciify* it. You will also need to remove the cedilla embellishment that is often added to the letter "C" in Turkish, Kurdish, Romance and other alphabets.

```
>>> from nlpia2.string_normalizers import Asciifier  
>>> asciify = Asciifier()  
  
>>> asciify("O'Néàl")  
"O'Neal"  
>>> asciify("Çetin")  
'Cetin'
```

Now that you have a pipeline that "normalizes" the alphabet for a broad range of languages, your model will generalize better. Your model will be useful for almost any latin script text, even text transliterated into latin script from other alphabets. You can use this exact same model to classify any string in

almost any language. You just need to label a few dozen examples in each language you are interested in "solving" for.

Now let's see if you've created a *solvable problem*. A solvable machine learning problem is one where:

1. You can imagine a human answering those same questions
2. There exists a correct answer for the vast majority of "questions" you want to ask your model
3. You don't expect a machine to achieve accuracy much better than a well-trained human expert

Think about this problem of predicting the country or dialect associated with a surname. Remember we've removed a lot of the clues about the language, like the characters and embellishments that are unique to non English languages. Is it solvable?

Start with the first question above. Can you imagine a human could identify a person's nationality from their asciiified surname alone? Personally, I often guess wrong when I try to figure out where one of my students is from, based on their surname. I will never achieve 100% accuracy in real life and neither will a machine. So as long as you're OK with an imperfect model, this is a solvable problem. And if you build a good pipeline, with lots of labeled data, you should be able to create an RNN model that is at least as accurate as humans like you or I. It may even be more accurate than a well-trained linguistics expert, which is pretty amazing, when you think about it. This where the concept if AI comes from, if a machine or algorithm can do intelligent things, we call it AI.

Think about what makes this problem hard. There is no one-to-one mapping between surnames and countries. Even though surnames are generally shared between parents and children for generations, people tend to move around. And people can change their nationality, culture, and religion. All these things affect the names that are common for a particular country. And sometimes individuals or whole families decide to change their last name, especially immigrants, expats and spies. People have a lot of different reasons for wanting to blend in.^[304] That blending of culture and language is what makes humans so awesome at working together to achieve great things,

including AI. RNNs will give your nationality prediction model the same flexibility. And if you want to change your name, this model can help you craft it so that it invokes the nationality that you want people (and machines) to perceive of you.

Take a look at some random names from this dataset to see if you can find any character patterns that are reused in multiple countries.

Listing 8.3. Load the

```
>>> repo = 'tangibleai/nlpia2'      #1
>>> filepath = 'src/nlpia2/data/surname-nationality.csv.gz'
>>> url = f"https://gitlab.com/{repo}--/raw/main/{filepath}"
>>> df = pd.read_csv(url)        #2
>>> df[['surname', 'nationality']].sort_values('surname').head(9)
   surname    nationality
16760  Aalbers        Dutch
16829  Aalders        Dutch
35706  Aalsburg       Dutch
35707    Aalst        Dutch
11070    Aalto      Finnish
11052  Aaltonen      Finnish
10853     Aarab      Moroccan
35708    Aarle        Dutch
11410    Aarnio      Finnish
```

Take a quick look at the data before diving in. It seems the Dutch like their family names (surnames) to be at the beginning of the roll call. Several Dutch surnames begin with "Aa." In the US there are a lot of business names that start with "AAA" for similar reasons. And it seems that Moroccan, Dutch, and Finnish languages and cultures tend to encourage the use of the trigram "Aar" at the beginning of words. So you can expect some confusion among these nationalities. Don't expect to achieve 90% accuracy on a classifier.

You also want to count up the unique categories in your dataset so you know how many options your model will have to choose from.

Listing 8.4. Unique nationalities in the dataset

```
>>> df['nationality'].nunique()
37
>>> sorted(df['nationality'].unique())
```

```
['Algerian', 'Arabic', 'Brazilian', 'Chilean', 'Chinese', 'Czech',  
 'English', 'Ethiopian', 'Finnish', 'French', 'German', 'Greek',  
 'Honduran', 'Indian', 'Irish', 'Italian', 'Japanese', 'Korean',  
 'Malaysian', 'Mexican', 'Moroccan', 'Nepalese', 'Nicaraguan', 'N  
 'Palestinian', 'Papua New Guinean', 'Peruvian', 'Polish', 'Portu  
 'Russian', 'Scottish', 'South African', 'Spanish', 'Ukrainian',  
 'Venezuelan', 'Vietnamese']
```

In listing 8.4 you can see the thirty-seven unique nationalities and language categories that were collected from multiple sources. This is what makes this problem difficult. It's like a multiple-choice question where there are 36 wrong answers and only one correct answer. And these region or language categories often overlap. For example Algerian is considered to be an Arabic language, and Brazilian is a dialect of Portuguese. There are several names that are shared across these nationality boundaries. So it's not possible for the model to get the correct answer for all of the names. It can only try to return the right answer as often as is possible.

The diversity of nationalities and data sources helped us do name substitution to anonymize messages exchanged within our multilingual chatbots. That way we can share conversation design datasets in open source projects like the chatbots discussed in Chapter 12 of this book. RNN models are great for anonymization tasks, such as named entity recognition and generation of fictional names. They can even be used to generate fictional, but realistic social security numbers, telephone numbers, and other PII (Personally Identifiable Information). To build this dataset we augmented the PyTorch RNN tutorial dataset with names scraped from public APIs that contained data for underrepresented countries in Africa, South and Central America, and Oceania.

When we were building this dataset during our weekly mob programming on Manning's Twitch channel, Rochdi Khalid pointed out that his last name is Arabic. And he lives in Casablanca, Morocco where Arabic is an official language, along side French and Berber. This dataset is a mashup of data from a variety of sources.^[305] some of which create labels based on broad language labels such as "Arabic" and others are labeled with their specific nationality or dialect, such as Moroccan, Algerian, Palestinian, or Malaysian.

Dataset bias is one of the most difficult biases to compensate for, unless you

can find data for the groups you want to elevate. Besides public APIs you can also mine your internal data for names. Our anonymization scripts strip out names from multilingual chatbot dialog. We added those names to this dataset to ensure it is a representative sample of the kinds of users that interact with our chatbots. You can use this dataset for your own projects where you need a truly global slice of names from a variety of cultures.

Diversity has its challenges. As you might imagine some spellings of these transliterated names are reused across national borders and even across languages. Translation and transliteration are two separate NLP problems that you can solve with RNNs. The word "नमस्कार" can be *translated* to the English word "hello". But before your RNN would attempt to translate a Nepalese word it would *transliterate* the Nepalese word "नमस्कार" into the word "namaskāra" which uses only the Latin character set. Most multilingual deep learning pipelines utilize the Latin character set (Romance script alphabet) to represent words in all languages.



Note

Transliteration is when you translate the characters and spellings of words from one language's alphabet to another, make it possible to represent words using the Latin character set (Romance script alphabet) used in Europe and the Americas. A simple example is the removal or adding of the acute accent from the French character "é", as in "resume" and "école" (school). Transliteration is a lot harder for non Latin alphabets such as Nepalese.

Here's how you can calculate just how much overlap there is within each of your categories (nationalities).

```
>>> fraction_unique = {}
>>> for i, g in df.groupby('nationality'):
>>>     fraction_unique[i] = g['surname'].nunique() / len(g)
>>> pd.Series(fraction_unique).sort_values().head(7)
Portuguese          0.860092
Dutch              0.966115
Brazilian           0.988012
Ethiopian           0.993958
Mexican             0.995000
Nepalese            0.995108
```

| | |
|---------|------------|
| Chilean | 0 . 998000 |
|---------|------------|

In addition to the overlap *across* nationalities, the PyTorch tutorial dataset contained many duplicated names within nationalities. More than 94% of the Arabic names were duplicates, some of which are shown in listing 8.5. Other nationalities and languages such as English, Korean, and Scottish appear to have been deduplicated. Duplicates in your training set make your model fit more closely to common names than to less frequently occurring names. Duplicating entries in your datasets is a brute force way of "balancing" your dataset or enforcing statistics about the frequency of phrases to help it predict popular names and heavily populated countries more accurately. This technique is sometimes referred to as "oversampling the minority class" because it boosts the frequency and accuracy on underrepresented classes in your dataset.

If you're curious about the original surname data check out the PyTorch "RNN Classification Tutorial".[\[306\]](#) There were only 108 unique Arabic surnames among the 2000 Arabic examples in Arabic.txt.[\[307\]](#)

Listing 8.5. Surname oversampling

```
>>> arabic = [x.strip() for x in open('.nlpia2-data/names/Arabic.txt')]
>>> arabic = pd.Series(sorted(arabic))
0      Abadi
1      Abadi
2      Abadi
...
1995    Zogby
1996    Zogby
1997    Zogby
Length: 2000, dtype: object
```

This means that even a relative simple model like the one shown in the PyTorch tutorial should be able to correctly label as Arabic the popular names like Abadi and Zogby correctly. And you can anticipate your model's confusion matrix statistics by counting up the number of nationalities associated with each name in the dataset.

You are going to use a deduplicated dataset that you loaded in . We have counted up the duplicates to give you the statistics for these duplicates

without burdening you with downloading a bloated dataset. And you will use balanced sampling of countries to encourage your model to treat all categories and names equally. This means your model will predict rare names and rare countries just as accurately as popular names from popular countries. This balanced dataset will encourage your RNN to generalize from the linguistic features it sees in names. Your model will be more likely to recognize patterns of letters that are common among many different names, especially those that help the RNN distinguish between countries. We've included information on how to obtain accurate usage frequency statistics for names in the `nlpia2` repository on GitLab.^[308] You'll need to keep this in mind if you intend to use this model in the real world on a more random sample of names.

Listing 8.6. Name nationality overlap

```
>>> df.groupby('surname')
>>> overlap = {}
... for i, g in df.groupby('surname'):
...     n = g['nationality'].nunique()
...     if n > 1:
...         overlap[i] = {'nunique': n, 'unique': list(g['nationa
>>> overlap.sort_values('nunique', ascending=False)
          nunique           uniq
Michel      6  [Spanish, French, German, English, Polish, Dutc
Abel        5  [Spanish, French, German, English, Russia
Simon       5  [Irish, French, German, English, Dutc
Martin      5  [French, German, English, Scottish, Russia
Adam        5  [Irish, French, German, English, Russia
...
Best        2  [German, Englis
Katz        2  [German, Russia
Karl        2  [German, Dutc
Kappel      2  [German, Dutc
Zambrano    2  [Spanish, Italia
```

To help diversify this dataset and make it a little more representative of real world statistics, we added some names from India and Africa. And we compressed the dataset by counting the duplicates. The resulting dataset of surnames combines data from the PyTorch RNN tutorial with anonymized data from multilingual chatbots.^[309] In fact we use this name classification and generation model to anonymize names in our chatbot logs. This allows us

to *default to open* with both NLP datasets as well as software.[\[310\]](#)

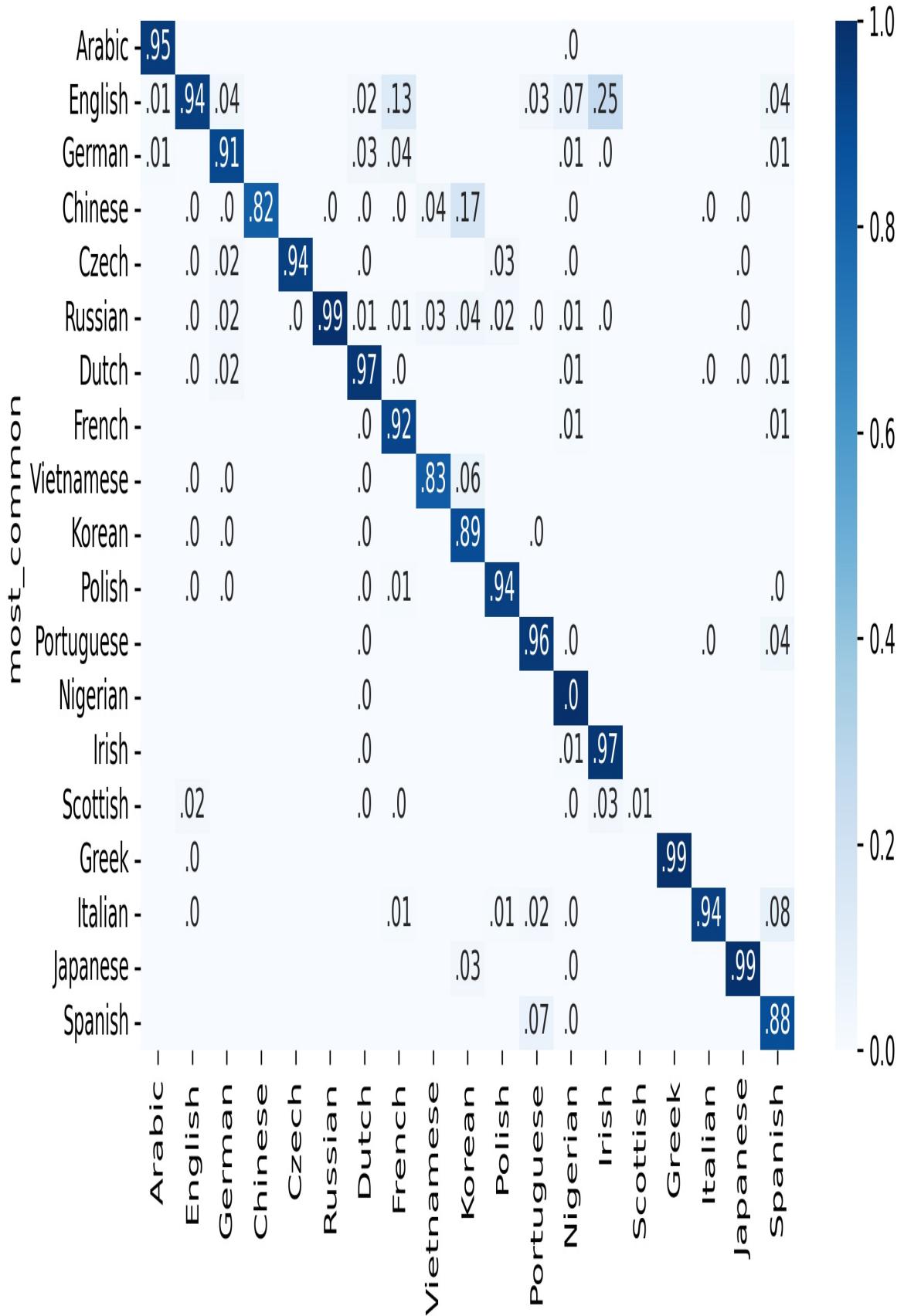


Important

A great way to find out if a machine learning pipeline has a chance of solving your problem, pretend you are the machine. Give yourself training on a few of the examples in your training set. Then try to answer a few of the "questions" in your test set without looking at the correct label. Your NLP pipeline should probably be able to solve your problem almost as well as you could. And in some cases you might find machines are much better than you, because they can balance many patterns in their head more accurately than you can.

By computing the most popular nationality for each name in the dataset, it is possible to create a confusion matrix, using the most common nationality as the "true" label for a particular name. This can reveal several quirks in the dataset that should influence what the model learns and how well it can perform this task. There is no confusion at all for Arabic names, because there are very few unique Arabic names and none of them are included in the other nationalities. And a significant overlap exists between Spanish, Portuguese, Italian and English names. Interestingly, for the 100 Scottish names in the dataset, None of them are most commonly labeled as Scottish. Scottish names are more often labeled as English and Irish names. This is because there are thousands of English and Irish names, but only 100 Scottish names in the original PyTorch tutorial dataset.

Figure 8.8. The dataset is confused even before training



We've added 26 more nationalities to the original PyTorch dataset. This creates much more ambiguity or overlap in the class labels. Many names are common in multiple different regions of the world. An RNN can deal with this ambiguity quite well, using the statistics of patterns in the character sequences to guide its classification decisions.

8.2.1 Build an RNN from scratch

Here's the heart of your RNN class in [8.7](#) Like all Python classes, a PyTorch Module class has an *init()* method where you can set some configuration values that control how the rest of the class works. For an RNN you can use the *init()* method to set the hyperparameters that control the number of neurons in the hidden vector as well as the size of the input and output vectors.

For an NLP application that relies on tokenizers it's a good idea to include the tokenizer parameters within the init method to make it easier to instantiate again from data saved to disk. Otherwise you'll find that you end up with several different models saved on your disk. And each model may use a different vocabulary or dictionary for tokenization and vectorization of your data. Keeping all those models and tokenizers connected is a challenge if they aren't stored together in one object.

The same goes for the vectorizers in your NLP pipeline. Your pipeline must be consistent about where it stores each word for your vocabulary. And you also have to be consistent about the ordering of your categories if your output is a class label. You can easily get confused if you aren't exactly consistent with the ordering of your category labels each time you reuse your model. The output will be garbled nonsense labels, if the numerical values used by your model aren't consistently mapped to human readable names for those categories. If you store your vectorizers in your model class (see listing [8.7](#)), it will know exactly which category labels it wants to apply to your data.

Listing 8.7. Heart of an RNN

```
>>> class RNN(nn.Module):
```

```

>>> def __init__(self, n_hidden=128, categories, char2i):      #1
...     super().__init__()
...     self.categories = categories
...     self.n_categories = len(self.categories)      #2
...     print(f'RNN.categories: {self.categories}')
...     print(f'RNN.n_categories: {self.n_categories}')

...     self.char2i = dict(char2i)
...     self.vocab_size = len(self.char2i)

...     self.n_hidden = n_hidden

...     self.W_c2h = nn.Linear(self.vocab_size + self.n_hidden, s
...     self.W_c2y = nn.Linear(self.vocab_size + self.n_hidden, s
...     self.softmax = nn.LogSoftmax(dim=1)

>>> def forward(self, x, hidden):      #3
...     combined = torch.cat((x, hidden), 1)
...     hidden = self.W_c2h(combined)
...     y = self.W_c2y(combined)
...     y = self.softmax(y)
...     return y, hidden      #4

```

Technically, your model doesn't need the full `char2i` vocabulary. It just needs the size of the one-hot token vectors you plan to input into it during training and inference. Likewise for the category labels. Your model only really needs to know the number of categories. The names of those categories are meaningless to the machine. But by including the category labels within your model you can print them to the console whenever you want to debug the internals of your model.

8.2.2 Training an RNN, one token at a time

The 30000+ surnames for 37+ countries in the `nlpia2` project is manageable, even on a modest laptop. So you should be able to train it using the in a reasonable amount of time. If your laptop has 4 or more CPU cores and 6 GB or more of RAM, the training will take about 30 minutes. And if you limit yourself to only 10 countries, 10000 surnames, and get lucky (or smart) with your choice of learning rate, you can train a good model in two minutes.

Rather than using the built in `torch.nn.RNN` layer you can build your first RNN from scratch using plain old `Linear` layers. This will generalize your

understanding so you can design your own RNNs for almost any application.

Listing 8.8. Training on a single sample must loop through the characters

```
>>> def train_sample(model, category_tensor, char_seq_tens,
...                   criterion=nn.NLLLoss(), lr=.005):
...     """ Train for one epoch (one example name nationality tensor
...     hidden = torch.zeros(1, model.n_hidden)      #1
...     model.zero_grad()      #2
...     for char_onehot_vector in char_seq_tens:
...         category_predictions, hidden = model(      #3
...             x=char_onehot_vector, hidden=hidden)      #4
...         loss = criterion(category_predictions, category_tensor)
...         loss.backward()
...
...         for p in model.parameters():
...             p.data.add_(p.grad.data, alpha=-lr)
...
...     return model, category_predictions, loss.item()
```

The `nlpia2` package contains a script to orchestrate the training process and allow you to experiment with different hyperparameters.

```
>>> %run classify_name_nationality.py      #1
      surname  nationality
0   Tesfaye    Ethiopian
...
[36241 rows x 7 columns]
```



Tip

You want to use the `%run` magic command within the ipython console rather than running your machine learning scripts in the terminal using the python interpreter. The ipython console is like a debugger. It allows you to inspect all the global variables and functions after your script finishes running. And if you cancel the run or if there is an error that halts the script, you will still be able to examine the global variables without having to start over from scratch.

Once you launch the `classify_name_nationality.py` script it will prompt you with sever questions about the model's hyperparameters. This is one of

the best ways to develop an instinct about deep learning models. And this is why we chose a relatively small dataset and small problem that can be successfully trained in a reasonable amount of time. This allows you to try many different hyperparameter combinations and fine tune your intuitions about NLP while fine tuning your model.

Listing 8.9 shows some hyperparameter choices that will give you pretty good results. But we've left you room to explore the "hyperspace" of options on your own. Can you find a set of hyperparameters that can identify a broader set of nationalities with better accuracy?

Listing 8.9. Interactive prompts so you can play with hyperparameters

```
How many nationalities would you like to train on? [10]? 25
```

```
model: RNN(
```

```
    n_hidden=128,  
    n_categories=25,  
    categories=[Algerian..Nigerian],  
    vocab_size=58,  
    char2i['A']=6
```

```
)
```

```
How many samples would you like to train on? [10000]? 1500
```

```
What learning rate would you like to train with? [0.005]? 0.010
```

```
2%|██████████| 30/1500 [00:06<05:16, 4.64it/s]000030 2% 00:06 3  
000030 2% 00:06 3.1712 Cai => Moroccan (21) X should be Nepalese
```

Even this simplified RNN model with only 128 neurons and 1500 epochs takes several minutes to converge to a decent accuracy. This example was trained on laptop with a 4-core (8-thread) i7 Intel processor and 64 GB of RAM. If your compute resources are more limited, you can train a simpler model on only 10 nationalities and it should converge much more quickly. Keep in mind that many names were assigned to multiple nationalities. And some of the nationality labels were more general language labels like "Arabic" that apply to many many countries. So you don't expect to get very high accuracy, especially when you give the model many nationalities (categories) to choose from.

Listing 8.10. Training output log

```

001470 98% 06:31 1.7358 Maouche => Algerian (0) ✓
001470 98% 06:31 1.8221 Quevedo => Mexican (20) ✓
...
001470 98% 06:31 0.7960 Tong => Chinese (4) ✓
001470 98% 06:31 1.2560 Nassiri => Moroccan (21) ✓
mean_train_loss: 2.1883266236980754
mean_train_acc: 0.5706666666666667
mean_val_acc: 0.2934249263984298
100%|██████████| 1500/1500 [06:39<00:00, 3.75it/s]

```

Looks like the RNN achieved 57% accuracy on the training set and 29% accuracy on the validation set. This is an unfair measure of the model's usefulness. Because the dataset was deduplicated before splitting into training and validation sets, so that there is only one row in the dataset for each name-nationality combination. This means that a name that is associated with one nationality in the training set will likely be associated with a *different* nationality in the validation set. This is why the PyTorch tutorial doesn't create test or validation datasets in the official docs. They don't want to confuse you.

Now that you understand the ambiguity in the dataset you can see how hard the problem is and that this RNN does a really good job of generalizing from the patterns it found in the character sequences. It generalizes to the validation set much better than random chance. Random guesses would have achieved 4% accuracy on 25 categories ($1/25 == .04$) even if there was no ambiguity in the nationality associated with each name.

Let's try it on some common surnames that are used in many countries. An engineer named Rochdi Khalid helped create one of the diagrams in this chapter. He lives and works in Casablanca, Morocco. Even though Morocco isn't the top prediction for "Khalid", Morocco is in second place!

```

>>> model.predict_category("Khalid")
'Algerian'
>>> predictions = topk_predictions(model, 'Khalid', topk=4)
>>> predictions
      text  log_loss nationality
rank
0      Khalid    -1.17    Algerian
1      Khalid    -1.35    Moroccan
2      Khalid    -1.80   Malaysian
3      Khalid    -2.40     Arabic

```

The top 3 predictions are all for Arabic speaking countries. I don't think there are expert linguists that could do this prediction as fast or as accurately as this RNN model did.

Now it's time to dig deeper and examine some more predictions to see if you can figure out how only 128 neurons are able to predict someone's nationality so well.

8.2.3 Understanding the results

In order to use a model like this in the real world you will need to be able to explain how it works to your boss. Germany, the Netherlands, and Finland, the Netherlands (and soon in all of the EU) are regulating how AI can be used, with the goal of forcing businesses to explain their AI algorithms so users can protect themselves.^[311] Businesses won't be able to hide their exploitative business practices within algorithms for long.^[312] You can imagine how government and business might use a nationality prediction algorithm for evil. Once you understand how this RNN works you'll be able to use that knowledge to trick algorithms into doing what's right, elevating rather than discriminating against historically disadvantaged groups and cultures.

Perhaps the most important piece of an AI algorithm is the metric you used to train it. You used `NLLLoss` for the PyTorch optimization training loop in listing . The `NLL` part stands for "Negative Log Likelihood". You should already know how to invert the `log()` part of that expression. Try to guess what the mathematical function and python code is to invert the `log()` function before checking out the code snippet below. As with most ML algorithms, `log` means natural log, sometimes written as *ln* or *log to the base e*.

```
>>> predictions = topk_predictions(model, 'Khalid', topk=4)
>>> predictions['likelihood'] = np.exp(predictions['log_loss'])
>>> predictions
   text  log_loss nationality  likelihood
rank
0    Khalid     -1.17    Algerian      0.31
1    Khalid     -1.35    Moroccan      0.26
2    Khalid     -1.80  Malaysian      0.17
```

This means that the model is only 31% confident that Rochdi is Algerian. These probabilities (likelihoods) can be used to explain how confident your model is to your boss or teammates or even your users.

If you're a fan of "debug by print" you can modify your model to print out anything you're interested in about the math the model uses to make predictions. PyTorch models can be instrumented with print statements whenever you want to record some of the internal goings on. If you do decide to use this approach, you only need to `.detach()` the tensors from the GPU or CPU where they are located in order to bring them back into your working RAM for recording in your model class.

A nice feature of RNNs is that the predictions are built up step by step as your `forward()` method is run on each successive token. This means you may not even need to add print statements or other instrumentation to your model class. Instead you can just make predictions of the hidden and output tensors for parts of the input text.

You may want to add some `predict_*` convenience functions for your model class to make it easier to explore and explain the model's predictions. If you remember the LogisticRegression model in scikit-learn it has a `predict_proba` method to predict probabilities in addition to the `predict` method used to predict the category. An RNN has an addition hidden state vector you may sometimes want to examine for clues as to how the network is making predictions. So you can create a `predict_hidden` method to output the 128-D hidden tensor and a `predict_proba` to show you the predicted probabilities for each of the target categories (nationalities).

```
>>> def predict_hidden(self, text="Khalid"):
...     text_tensor = self.encode_one_hot_seq(text)
...     with torch.no_grad():    #1
...         hidden = self.hidden_init
...         for i in range(text_tensor.shape[0]):      #2
...             y, hidden = self(text_tensor[i], hidden)   #3
...     return hidden
```

This `predict_hidden` convenience method converts the text (surname) into a tensor before iterating through the one-hot tensors to run the forward method

(or just the model's `self`).

```
>>> def predict_proba(self, text="Khalid"):
...     text_tensor = self.encode_one_hot_seq(text)
...     with torch.no_grad():
...         hidden = self.hidden_init
...         for i in range(text_tensor.shape[0]):
...             y, hidden = self(text_tensor[i], hidden)
...     return y #1
```

This `predict_hidden` method gives you access to the most interesting part of the model where the "logic" of the predictions is taking place. The hidden layer evolves as it learns more and more about the nationality of a name with each character.

Finally, you can use a `predict_category` convenience method to run the model's forward pass predictions to predict the nationality of a name.

```
>>> def predict_category(self, text):
...     tensor = self.encode_one_hot_seq(text)
...     y = self.predict_proba(tensor)      #1
...     pred_i = y.topk(1)[1][0].item()    #2
...     return self.categories[pred_i]
```

The key thing to recognize is that for all of these methods you don't necessarily have to input the entire string for the surname. It is perfectly fine to reevaluate the first part of the surname text over and over again, as long as you reset the hidden layer each time.

If you input an expanding window of text you can see how the predictions and hidden layer evolve in their understanding of the surname. During mob programming sessions with other readers of the book we noticed that nearly all names started out with predictions of "Chinese" as the nationality for a name until after the 3rd or 4th character. This is perhaps because so many Chinese surnames contain 4 (or fewer) characters.[\[313\]](#)

Now that you have helper functions you can use them to record the hidden and category predictions as the RNN is run on each letter in a name.

```
>>> text = 'Khalid'
>>> pred_categories = []
```

```

>>> pred_hiddens = []

>>> for i in range(1, len(text) + 1):
...     pred_hiddens.append(model.predict_hidden(text[:i]))      #1
...     pred_categories.append(model.predict_category(text[:i]))

>>> pd.Series(pred_categories, input_texts)
# K          English
# Kh         Chinese
# Kha        Chinese
# Khal       Chinese
# Khali      Algerian
# Khalid    Arabic

```

And you can create a 128×6 matrix of all the hidden layer values in a 6-letter name. The list of PyTorch tensors can be converted to a list of lists and then a DataFrame to make it easier to manipulate and explore.

```

>>> hiddens = [h[0].tolist() for h in hiddens]
>>> df_hidden = pd.DataFrame(hidden_lists, index=list(text))
>>> df_hidden = df_hidden.T.round(2)      #1

>>> df_hidden
   0   1   2   3   4   5   ...   122   123   124   125
K  0.10 -0.06 -0.06  0.21  0.07  0.04 ...  0.16  0.12  0.03  0.06
h -0.03  0.03  0.02  0.38  0.29  0.27 ... -0.08  0.04  0.12  0.30
a -0.06  0.14  0.15  0.60  0.02  0.16 ... -0.37  0.22  0.30  0.33
l -0.04  0.18  0.14  0.24 -0.18  0.02 ...  0.27 -0.04  0.08 -0.02
i -0.11  0.12 -0.00  0.23  0.03 -0.19 ... -0.04  0.29 -0.17  0.08
d  0.01  0.01 -0.28 -0.32  0.10 -0.18 ...  0.09  0.14 -0.47 -0.02
[6 rows x 128 columns]

```

This wall of numbers contains every thing your RNN "thinks" about the name as it is reading through it.



Tip

There are some Pandas display options that will help you get a feel for the numbers in a large DataFrame without TMI ("too much information"). Here are some of the settings that helped improve the printouts of tables in this book

To display only 2 decimal places of precision for floating point values try:

```
pd.options.display.float_format = '{:.2f}'.
```

To display a maximum of 12 columns and 7 rows of data from your DataFrame: `pd.options.display.max_columns = 12` and `pd.options.display.max_rows = 7`

These only affect the displayed representation of your data, not the internal values used when you do addition or multiplication.

As you've probably done with other large tables of numbers, it's often helpful to find patterns by correlating it with other numbers that are interesting to you. For example you may want to find out if any of the hidden weights are keeping track of the RNNs position within the text - how many characters it is from the beginning or end of the text.

```
>>> position = pd.Series(range(len(text)), index=df_hidden.index)
>>> pd.DataFrame(position).T
#      K   h   a   l   i   d
# 0    0   1   2   3   4   5

>>> df_hidden_raw.corrwith(position).sort_values()
# 11    -0.99
# 84    -0.98
# 21    -0.97
#
# ...
# 6     0.94
# 70    0.96
# 18    0.96
```

Interestingly our hidden layer has room in it's hidden memory to record the position in many different places. And the strongest correlation seems to be negative. These are likely helping the model to estimate the likelihood of the current character being the last character in the name. When we looked at a wide range of example names, the predictions only seemed to converge on the correct answer at the very last character or two. Andrej Karpathy experimented with several more ways to glean insight from the weights of your RNN model in his blog post "The unreasonable effectiveness of RNNs" in the early days of discovering RNNs. [\[314\]](#)

8.2.4 Multiclass classifiers vs multi-label taggers

How can you deal with the ambiguity of multiple different correct nationalities for surnames? The answer is multi-label classification or tagging rather than the familiar multiclass classification. Because the terms "multiclass classification" and "multi-label classification" sound so similar and are easily confused, you probably want to use the term "multi-label tagging" or just "tagging" instead of "multi-label classification." And if you're looking for the `sklearn` models suited to this kind of problem you want to search for "multi-output classification."

Multi-label taggers are made for ambiguity. In NLP intent classification and tagging is full of intent labels that have fuzzy overlapping boundaries. We aren't talking about a graffiti war between Banksy and Bario Logan street artists when we say "taggers". We're talking about a kind of machine learning model that is able to assign multiple discrete labels to an object in your dataset.

A multiclass classifier has multiple different categorical labels that are matched to objects, one label for each object. A categorical variable takes on only one of a number of mutually exclusive classes or categories. For example if you wanted to predict both the language and the gender associated with first names (given names), then that would require a multiclass classifier. But if you want to label a name with all the relevant nationalities and genders that are appropriate, then you would need a tagging model.

This may seem like splitting hairs to you, but it's much more than just semantics. It's the semantics (meaning) of the text that you are processing that is getting lost in the noise of bad advice on the Internet. David Fischer at ReadTheDocs.com (RTD) and the organizer for San Diego Python ran into these misinformed blog posts when he started learning about NLP to build a Python package classifier. Ultimately he ended up building a tagger, which gave RTD advertisers more effective placements for their ads and gave developers reading documentation more relevant advertisements.



Tip

To turn any multi-class classifier into a multi-label tagger you must change your activation function from softmax to an element-wise sigmoid function.

A softmax creates a probability distribution across all the mutually exclusive categorical labels. A sigmoid function allows each and every value to take on any value between zero and one, such that each dimension in your multi-label tagging output represents the independent binary probability of that particular label applying to that instance.

[304] Lex Fridman interview with ex-spy Andrew Bustamante (<https://lexfridman.com/andrew-bustamante>)

[305] There's more info and data scraping code in the nlpia2 package (<https://proai.org/nlpia-ch08-surnames>)

[306] PyTorch RNN Tutorial by Sean Robertson (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.htm)

[307] The original PyTorch RNN Tutorial surname dataset with duplicates (<https://download.pytorch.org/tutorial/data.zip>)

[308] iPython history log in the nlpia2 repository on GitLab with examples for scraping surname data (<https://proai.org/nlpia-ch08-surnames>)

[309] PyTorch character-based RNN tutorial (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.htm)

[310] Qary (<https://docs.qary.ai>) combines technology and data from all our multilingual chatbots (<https://tangibleai.com/our-work>)

[311] AI algorithm registry launched in Amsterdam in 2020 (<https://algoritmeregister.amsterdam.nl/en/ai-register/>)

[312] "EU Artificial Intelligence Act (<https://artificialintelligenceact.eu/>) and the accepted OECD AI Council recommendations (<https://legalinstruments.oecd.org/en/instruments/OECD-LEGAL-0449>)"

[313] Thank you Tiffany Kho for pointing this out.

[314] footnote: ["The unreasonable effectiveness of RNNs" by Andrej Karpathy (<https://karpathy.github.io/2015/05/21/rnn-effectiveness>)

8.3 Backpropagation through time

Backpropagation for RNNs is a lot more work than for CNNs. The reason training an RNN is so computationally expensive is that it must perform the forward and backward calculations many times for each text example - once for each token in the text. And then it has to do all that again for the next layer in the RNN. And this sequence of operations is really important because the computation for one token depends on the previous one. You are recycling the output and hidden state tensors back into the calculation for the next token. For CNNs and fully connected neural networks the forward and backward propagation calculations could run all at once on the entire layer. The calculations for each token in your text did not affect the calculation for the neighboring tokens in the same text. RNNs do forward and backward propagation in time, from one token in the sequence to the next.

But you can see in the unrolled RNN in figure 8.7 that your training must propagate the error back through all the weight matrix multiplications. This is so the weight matrices, even though the weight matrices are the same, or shared, or tied for all the tokens in your data, they must work on each and every token in each of your texts. So your training loop will need to loop through all the tokens backward to ensure that the error at each step of the way is used to adjust the weights.

The initial error value is the distance between the final output vector and the "true" vector for the label appropriate for that sample of text. Once you have that difference between the truth and the predicted vector you can work your way back through time (tokens) to propagate that error to the previous time step (previous token). The PyTorch package will use something very similar to the chain rule that you used in algebra or calculus class to make this happen. PyTorch calculates the gradients it needs during forward propagation and then multiplies those gradients by the error for each token to decide how much to adjust the weights and improve the predictions.

And once you've adjusted the weights for all the tokens in one layer you do the same thing again for all the tokens on the next layer. Working your way from the output of the network all the way back to the inputs (tokens) you will eventually have to "touch" or adjust all of the weights many times for

each text example. Unlike backpropagation through a linear layer or CNN layer, the backpropagation on an RNN must happen serially, one token at a time.

An RNN is just a normal feedforward neural network "rolled up" so that the Linear weights are multiplied again and again for each token in your text. If you unroll it you can see all the weight matrices that need to be adjusted. And like the CNN, many of the weight matrices are shared across all of the tokens in the unrolled view of the neural network computational graph. An RNN is one long kernel that reuses "all" of the weights for each text document. The weights of an RNN are one long, giant kernel. At each time step, it is the *same* neural network, just processing a different input and output at that location in the text.

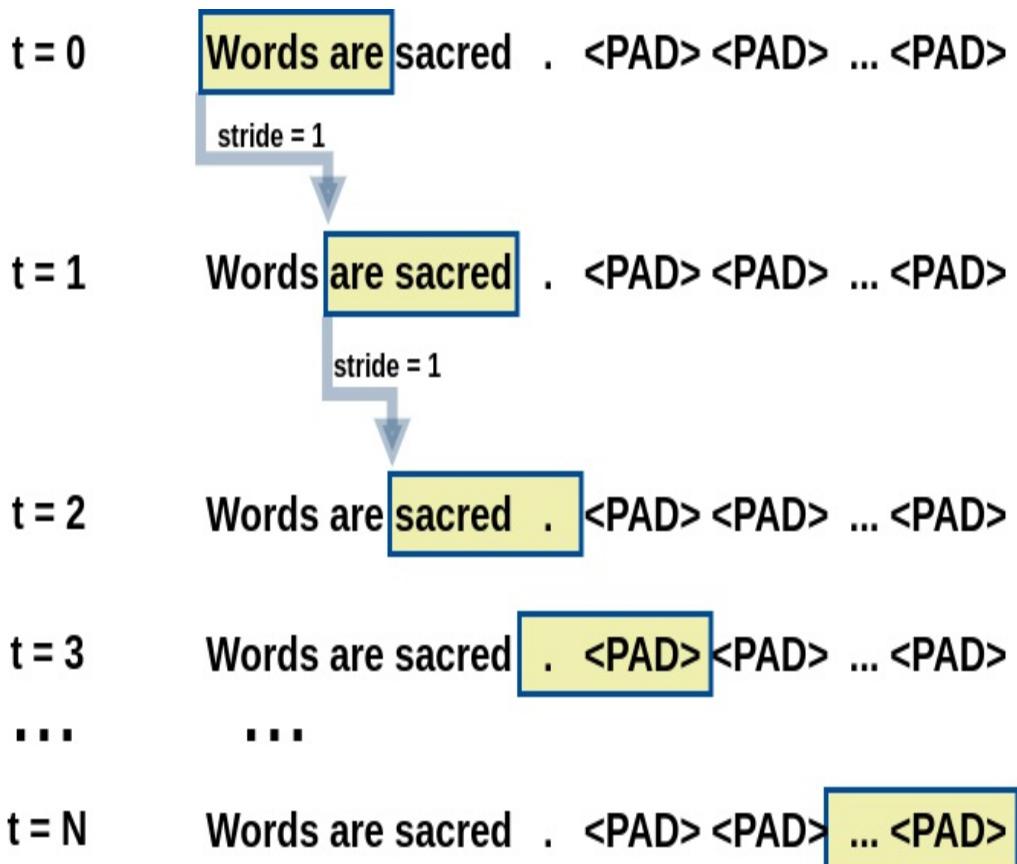


Tip

In all of these examples, you have been passing in a single training example, the *forward pass*, and then backpropagating the error. As with any neural network, this forward pass through your network can happen after each training sample, or you can do it in batches. And it turns out that batching has benefits other than speed. But for now, think of these processes in terms of just single data samples, single sentences, or documents.

In chapter 7 you learned how to process a string all at once with a CNN. CNNs can recognize patterns of meaning in text using kernels (matrices of weights) that represent those patterns. CNNs and the techniques of previous chapters are great for most NLU tasks such as text classification, intent recognition, and creating embedding vectors to represent the meaning of text in a vector. CNNs accomplish this with overlapping windows of weights that can detect almost any pattern of meaning in text.

Figure 8.9. 1D convolution with embeddings



In chapter 7 you imagined striding the kernel window over your text, one step at a time. But in reality, the machine is doing all the multiplications in parallel. The order of operations doesn't matter. For example, the convolution algorithm can do the multiplication on the pair of words and then hop around to all the other possible locations for the window. It just needs to compute a bunch of dot products and then sum them all up or pool them together at the end. Addition is commutative (order doesn't matter). And none of the convolution dot products depend on any of the others. In fact, on a GPU these matrix multiplications (dot products) are all happening *in parallel* at approximately the *same* time.

But an RNN is different. With an RNN you're recycling the output of one token back into the dot product you're doing on the next token. So even though we talked about RNNs working on any length text, to speed things up, most RNN pipelines truncate and pad the text to a fixed length. This unrolls the RNN matrix multiplications so that And you need two matrix multiplications for an RNN compared to one multiplication for a CNN. You need one matrix of weights for the hidden vector and another for the output

vector.

If you've done any signal processing or financial modeling you may have used an RNN without knowing it. The recurrence part of a CNN is called 'auto-regression' in the world of signal processing and quantitative financial analysis. An *auto-regressive moving average* (ARMA) model is an RNN in disguise.[\[315\]](#)

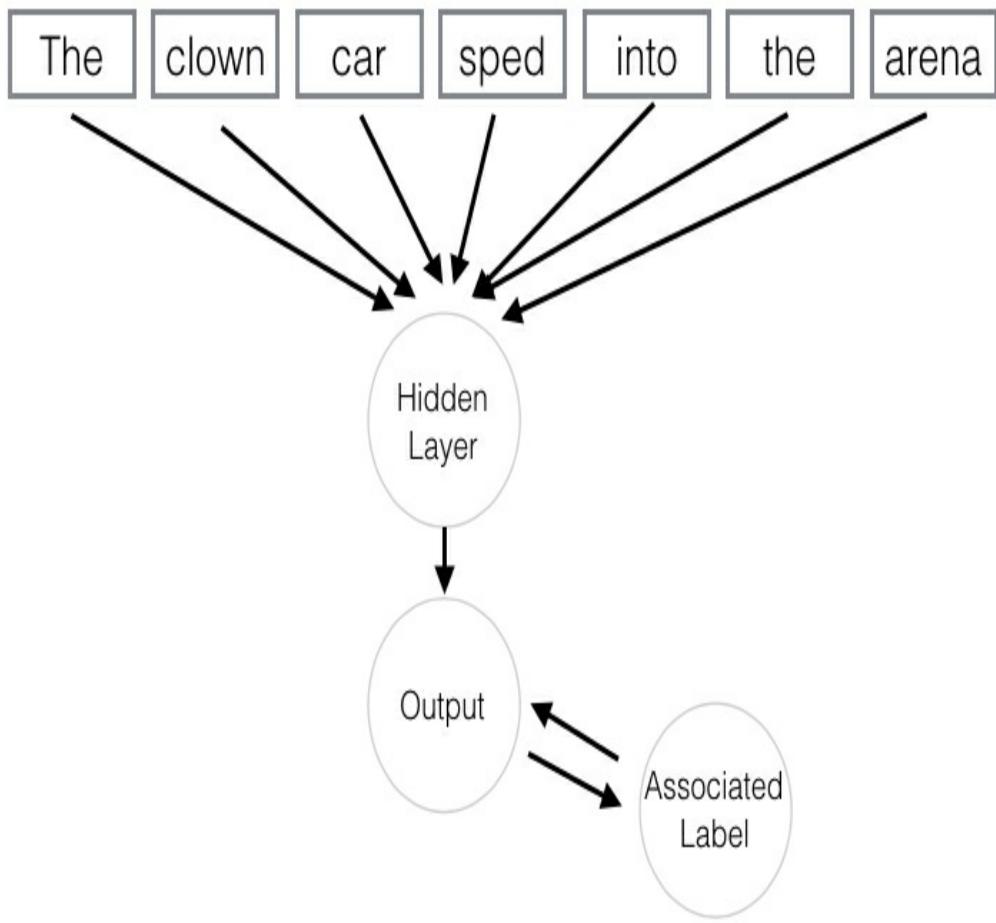
In this chapter you are learning about a new way to structure the input data. Just as in a CNN, each token is associated with a time (t) or position within the text. The variable t is just another name for the index variable in your sequence of tokens.

You will even see places where you use the integer value of t to retrieve a particular token in the sequence of tokens with an expression such as `token = tokens[t]`. So when you see $t-1$ or `tokens[t-1]` you know that is referring to the preceding time step or token. And $t+1$ and `tokens[t+1]` refers to the next time step or token. In past chapters you may have seen that we sometimes used i for this index value.

Now you will use multiple different indexes to keep track of what has been passed into the network and is being output by the network:

- t or `token_num`: time step or token position for the current tensor being input to the network
- k or `sample_num`: sample number within a batch for the text example being trained on
- b or `batch_num`: batch number of the set of samples being trained
- `epoch_num`: number of epochs that have passed since the start of training

Figure 8.10. Data fed into a recurrent network



This 2-D tensor representation of a document is similar to the "player piano" representation of text in chapter 2. Only this time you are creating a dense representation of each token using word embeddings.

For an RNN you no longer need to process each text sample all at once. Instead, you process text one token at a time.

In your recurrent neural net, you pass in the word vector for the first token and get the network's output. You then pass in the second token, but you also pass in the output from the first token! And then pass in the third token along with the output from the second token. And so on. The network has a concept of before and after, cause and effect, some vague notion of time (see figure 8.8).

8.3.1 Initializing the hidden layer in an RNN

There's a chicken and egg problem with the hidden layer when you restart the training of an RNN on each new document. For each text string you want to process, there is no "previous" token or previous hidden state vector to recycle back into the network. You don't have anything to prime the pump with and start the recycling (recurrence) loop. Your model's `forward()` method needs a vector to concatenate with the input vector so that it will be the right size for multiplying by w_{c2h} and w_{c2o} .

The most obvious approach is to set the initial hidden state to all zeroes and allow the biases and weights to quickly ramp up to the best values during the training on each sample. This can be great for any of the neurons that are keeping track of time, the position in the token sequence that is currently (recurrently) being processed. But there are also neurons that are trying to predict how far from the end of the sequence you are. And your network has a defined polarity with 0 for off and 1 for on. So you may want your network to start with a mix of zeros and ones for your hidden state vector. Better yet you can use some gradient or pattern of values between zero and 1 that is your particular "secret sauce", based on your experience with similar problems.

Getting creative and being consistent with your initialization of deep learning networks has the added benefit of creating more "explainable" AI. You will often create predictable structure in your weights. And by doing it the same way each time you will know where to look within all the layers. For example, you will know which positions in the hidden state vector are keeping track of position (time) within the text.

To get the full benefit of this consistency in your initialization values you will also need to be consistent with the ordering of your samples used during training. You can sort your texts by their lengths, as you did with CNNs in chapter 7. But many texts will have the same length, so you will also need a sort algorithm that consistently orders the samples with the same length. Alphabetizing is an obvious option, but this will tend to trap your model in local minima as it's trying to find the best possible predictions for your data. It would get really good at the "A" names, but do poorly on "Z" names. So

don't pursue this advanced seeding approach until you've fully mastered the random sampling and shuffling that has proven so effective.

As long as you are consistent throughout the training process, your network will learn the biases and weights that your network needs to layer on top of these initial values. And that can create recognizable structure in your neural network weights.



Tip

In some cases it can help to seed your neural networks with an initial hidden state other than all zeros. Johnathon Frankle and Michael Carbin found that being intentional about reuse of good initialization values can be key to helping a network find the *global minimum* loss achievable for a particular dataset "Lottery Ticket Hypothesis" paper, [\[316\]](#) Their approach is to initialize all weights and biases using a random seed that can be reused in subsequent trainings.

Now your network is remembering something! Well, sort of. A few things remain for you to figure out. For one, how does backpropagation even work in a structure like this?

Another approach that is popular in the Keras community is to retain the hidden layer from a previous batch of documents. This "pretrained" hidden layer embedding gives your language model information about the context of the new document - the text that came before it. However, this only makes sense if you've maintained the order of your documents within the batches and across batches that you are training. In most cases you shuffle and reshuffle your training examples with each epoch. You do this when you want your model to work equally well at making predictions "cold" without any priming by reading similar documents or nearby passages of text.

So unless you are trying to squeeze out every last bit of accuracy you can for a really difficult problem you should probably just reset it to zeros every time to start feeding a new document into your model. And if you do use this *stateful* approach to training an RNN, make sure you will be able to warm up your model on context documents for each prediction it needs to make in the

real world (or on your test set). And make sure you prepare your documents in a consistent order and can reproduce this document ordering for a new set of documents that you need to do prediction on with your model.

[315] ARMA model explanation
(https://en.wikipedia.org/wiki/Autoregressive_model)

[316] <https://arxiv.org/pdf/1803.03635.pdf>

8.4 Remembering with recurrent networks

An RNN remembers previous words in the text they are processing and can keep adding more and more patterns to its memory as it processes a theoretically limitless amount of text. This can help it understand patterns that span the entire text and recognize the difference between two texts that have dramatically different meaning depending on where words occur.

I apologize for the lengthy letter. I didn't have time to write a shorter one.

I apologize for the short letter. I didn't have time to write a lengthy one.

Swapping the words "short" and "lengthy", flips the meaning of this Mark Twain quote. Knowing Mark Twain's dry sense of humor and passion for writing, can you tell which quote is his? It's the one where he apologizes for the lengthy letter. He's making light of the fact that editing and writing conciesly is hard work. It's something that smart humans can still do better than even the smartest AI.

The CNNs you learned about in chapter 7 would have a hard time making the connection between these two sentences about lengthy and short letters, whereas RNNs make this connection easily. This is because CNNs have a limited window of text that they can recognize patterns within. To make sense of an entire paragraph, you would have to build up layers of CNNs with overlapping kernels or windows of text that they understand. RNNs do this naturally. RNNs remember something about each and every token in the document they've read. They remember everything you've input into them, until you tell them you are done with that document. This makes them better

at summarizing lengthly Mark Twain letters and makes them better at understanding his long sophisticated jokes.

Mark Twain was right. Communicating things concisely requires skill and intelligence and attention to detail. In the paper "Attention is All You Need" Ashish Vaswani revealed how transformers can add an attention matrix that allows RNNs to accurately understand much longer documents.[\[317\]](#) In chapter 9 you'll see this attention mechanism at work, as well as the other tricks that make the transformer approach to RNNs the most successful and versatile deep learning architecture so far.

Summarization of lengthy text is still an unsolved problem in NLP. Even the most advanced RNNs and transformers make elementary mistakes. In fact, The Hutter Prize for Artificial Intelligence will give you 5000 Euros for each one percent improvement in the compression (lossless summarization) of Wikipedia.[\[318\]](#) The Hutter Prize focuses on the compression of the symbols within Wikipedia. You're going to learn how to compress the meaning of text. That's even harder to do well. And it's hard to measure how well you've done it.

You will have to develop generally intelligent machines that understand common sense logic and can organize and manipulate memories and symbolic representations of those memories. That may seem hopeless, but it's not. The RNNs you've built so far can remember everything in one big hidden representation of their understanding. Can you think of a way to give some structure to that memory, so that your machine can organize its thoughts about text a bit better? What if you gave your machine a separate ways to maintain both short term memories and long term memories? This would give it a working memory that it could then store in long term memory whenever it ran across a concept that was important to remember.

8.4.1 Word-level Language Models

And all the most impressive language models that you've read about use words as their tokens, rather than individual characters. So, before you jump into GRUs and LSTMs you will need to rearrange your training data to contain sequences of word IDs rather than character (letter) IDs. And you're

going to have to deal with much longer documents than just surnames, so you will want to *batchify* your dataset to speed it up.

Take a look at the Wikitext-2 dataset and think about how you will preprocess it to create a sequence of token IDs (integers).

```
>>> lines = open('data/wikitext-2/train.txt').readlines()
>>> for line in lines[:4]:
...     print(line.rstrip()[:70])

= Valkyria Chronicles III =
=====

Senjō no Valkyria 3 : <unk> Chronicles ( Japanese : 戦場のヴァルキリ
```

Oh wow, this is going to be an interesting dataset. Even the English language version of Wikipedia contains a lot of other natural languages in it, such as Japanese in this first article. If you use your tokenization and vocabulary building skills of previous chapters you should be able to create a Corpus class like the one used in the RNN examples coming up. [\[319\]](#)

```
>>> from nlpia2.ch08.data import Corpus

>>> corpus = Corpus('data/wikitext-2')
>>> corpus.train
tensor([ 4,  0,  1,  ..., 15,  4,  4])
```

And you always want to make sure that your vocabulary has all the info you need to generate the correct words from the sequence of word IDs:

```
>>> vocab = corpus.dictionary
>>> [vocab.idx2word[i] for i in corpus.train[:7]]
['<eos>', '=', 'Valkyria', 'Chronicles', 'III', '=', '<eos>']
```

Now, during training your RNN will have to read each token one at a time. That can be pretty slow. What if you could train it on multiple passages of text simultaneously? You can do this by splitting your text into batches or *batchifying* your data. These batches can each become columns or rows in a matrix that PyTorch can more efficiently perform math on within a *GPU* (Graphics Processing Unit).

In the `nlpia2.ch08.data` module you'll find some functions for batchifying

long texts.

```
>>> def batchify_slow(x, batch_size=8, num_batches=5):
...     batches = []
...     for i in range(int(len(x)/batch_size)):
...         if i > num_batches:
...             break
...         batches.append(x[i*batch_size:i*batch_size + batch_size])
...     return batches
>>> batches = batchify_slow(corpus.train)

>>> batches
[tensor([4, 0, 1, 2, 3, 0, 4, 4]),
 tensor([ 5, 6, 1, 7, 8, 9, 2, 10]),
 tensor([11, 8, 12, 13, 14, 15, 1, 16]),
 tensor([17, 18, 7, 19, 13, 20, 21, 22]),
 tensor([23, 1, 2, 3, 24, 25, 13, 26]),
 tensor([27, 28, 29, 30, 31, 32, 33, 34])]
```

One last step, and your data is ready for training. You need to stack the tensors within this list so that you have one large tensor to iterate through during your training.

```
>>> torch.stack(batches)
tensor([[4, 0, 1, 2, 3, 0, 4, 4],
        [ 5, 6, 1, 7, 8, 9, 2, 10],
        [11, 8, 12, 13, 14, 15, 1, 16],
        ...]
```

8.4.2 Gated Recurrent Units (GRUs)

For short text, ordinary RNNs with a single activation function for each neuron works well. All your neurons need to do is recycle and reuse the hidden vector representation of what they have read so far in the text. But ordinary RNNs have a short attention span that limits their ability to understand longer texts. The influence of the first token in a string fades over time as your machine reads more and more of the text. That's the problem that GRU (Gated Recurrent Unit) and LSTM (Long and Short Term Memory) neural networks aim to fix.

How do you think you could counteract fading memory of early tokens in a text string? How could you stop the fading, but just for a few important

tokens at the beginning of a long text string? What about adding an `if` statement to record or emphasize particular words in the text. That's what GRUs do. GRUs add `if` statements, called *logic gates* (or just "gates"), to RNN neurons.

The magic of machine learning and back propagation will take care of the if statement conditions for you, so you don't have to adjust logic gate thresholds manually. Gates in an RNN learn the best thresholds by adjusting biases and weights that affect the level of a signal that triggers a zero or 1 output (or something in between). And the magic of back-propagation in time will train the LSTM gates to let important signals (aspects of token meaning) to pass through and get recorded in the hidden vector and cell state vector.

But wait, you probably thought we already had if statements in our network. After all, each neuron has a nonlinear activation function that acts to squash some outputs to zero and push others up close to 1. So the key isn't that LSTMs add gates (activation functions) to your network. The key is that the new gates are *inside* the neuron and connected together in a way that creates structure to your neural network that wouldn't naturally just emerge from a normal linear, fully-connected layer of neurons. And that structure was intentionally designed with a purpose, reflecting what researchers thought would help RNN neurons deal with this long term memory problem.

In addition to the original RNN output gate, GRUs add two new logic gates or activation functions within your recurrent unit.

1. Reset gate: What parts of the hidden layer should be blocked because they are no longer relevant to the current output.
2. Update gate: What parts of the hidden layer should matter to the current output (now, at time t).

You already had an activation function on the output of your RNN layer. This output logic gate is called the "new" logic gate in a GRU.

```
>>> r = sigmoid(w_i2r.mm(x) + b_i2r + w_h2r.mm(h) + b_h2r)
>>> z = sigmoid(w_i2z.mm(x) + b_i2z + w_h2z.mm(h) + b_h2z)

>>> n = tanh(w_i2n.mm(x) + b_i2n + r*(w_h2n.mm(h) + b_h2n))
```

So when you are thinking about how many units to add to your neural network to solve a particular problem, each LSTM or GRU unit gives your network a capacity similar to 2 "normal" RNN neurons or hidden vector dimensions. A unit is just a more complicated, higher capacity neuron, and you can see this if you count up the number of "learned parameters" in your LSTM model and compare it to those of an equivalent RNN.

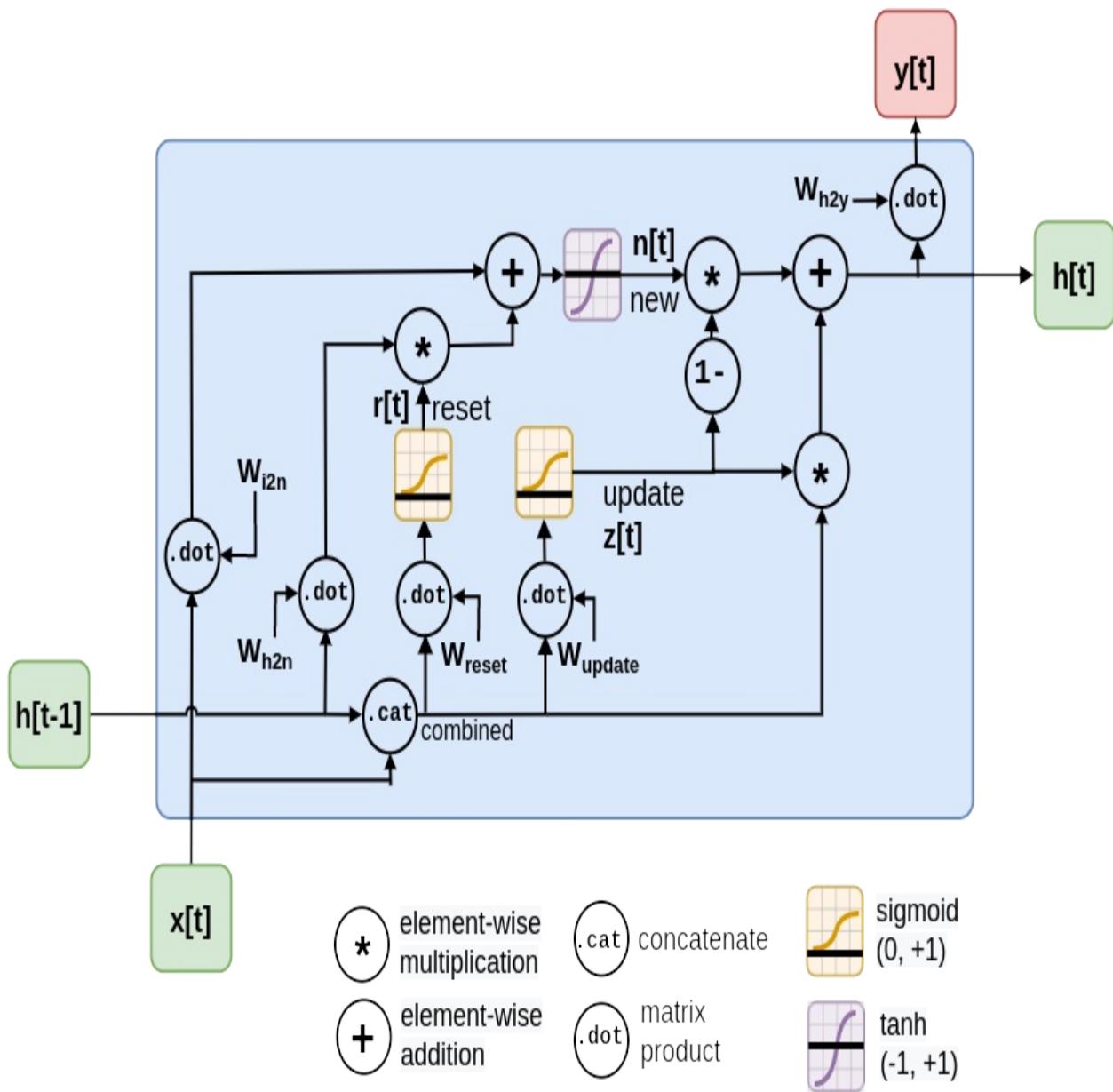


Note

You're probably wondering why we started using the word "unit" rather than "neuron" for the elements of this neural net. Researchers use the terms "unit" or "cell" to describe the basic building blocks of an LSTM or GRU neural network because they are a bit more complicated than a neuron. Each unit or cell in an LSTM or GRU contains internal gates and logic. This gives your GRU or LSTM units more capacity for learning and understanding text, so you will probably need fewer of them to achieve the same performance as an ordinary RNN.

The *reset*, *update*, and *new* logic gates are implemented with the fully-connected linear matrix multiplications and nonlinear activation functions you are familiar with from Chapter 5. What's new is that they are implemented on each token recurrently and they are implemented on the hidden and input vectors in parallel. Figure 8.12 shows how the input vector and hidden vector for a single token flow through the logic gates and outputs the prediction and hidden state tensors.

Figure 8.11. GRUs add capacity with logic gates



If you have gotten good at reading data flow diagrams like Figure 8.12 you may be able to see that the GRU *update* and *relevance* logic gates are implementing the following two functions: [\[320\]](#)

```
r = sigmoid(W_i2r.dot(x) + b_i2r + W_h2r.dot(h) + b_h2r)      #1
z = sigmoid(W_i2z.dot(x) + b_i2z + W_h2z.dot(h) + b_h2z)      #2
```

Looking at these two lines of code you can see that inputs to the formula are exactly the same. Both the hidden and input tensors are multiplied by weight matrices in both formulas. And if you remember your linear algebra and

matrix multiplication operations, you might be able to simplify the And you may notice in the block diagram (figure 8.12) that the input and hidden tensors are concatenated together before the matrix multiplication by W_{reset} , the reset weight matrix.

Once you add GRUs to your mix of RNN model architectures, you'll find that they are much more efficient. A GRU will achieve better accuracy with fewer learned parameters and less training time and less data. The gates in a GRU give structure to the neural network that creates more efficient mechanisms for remembering important bits of meaning in the text. To measure efficiency you'll need some code to count up the learned (trainable) parameters in your models. This is the number of weight values that your model must adjust to optimize the predictions. The `requires_grad` attribute is an easy way to check whether a particular layer contains learnable parameters or not.[\[321\]](#)

```
>>> def count_parameters(model, learned=True):
...     return sum(
...         p.numel() for p in model.parameters()      #1
...         if not learned or p.requires_grad        #2
...     )
```

The more weights or learned parameters there are, the greater the capacity of your model to learn more things about the data. But the whole point of all the clever ideas, like convolution and recurrence, is to create neural networks that are efficient. Choosing the right combination of algorithms, sizes and types of layers, you can reduce the number of weights or parameters your model must learn while simultaneously creating smarter models with greater capacity to make good predictions.

If you experiment with a variety of GRU hyperparameters using the `nlpia2/ch08/rnn_word/hypertune.py` script you can aggregate all the results with your RNN results to compare them all together.

```
>>> import jsonlines      #1
>>> with jsonlines.open('experiments.jsonl') as fin:
...     lines = list(fin)
>>> df = pd.DataFrame(lines)
>>> df.to_csv('experiments.csv')
```

```

>>> cols = 'learned_parameters rnn_type epochs lr num_layers'
>>> cols += ' dropout epoch_time test_loss'
>>> cols = cols.split()
>>> df[cols].round(2).sort_values('test_loss', ascending=False)

>>> df
   parameters  rnn_type  epochs  lr  layers  drop  time (s)  l
3      13746478  RNN_TANH     1  0.5      5  0.0    55.46  6
155     14550478        GRU     1  0.5      5  0.2    72.42  6
147     14550478        GRU     1  0.5      5  0.0    58.94  6
146     14068078        GRU     1  0.5      3  0.0    39.83  6
1      13505278  RNN_TANH     1  0.5      2  0.0    32.11  6
..      ...
133     13505278  RNN_RELU    32  2.0      2  0.2   1138.91  5
134     13585678  RNN_RELU    32  2.0      3  0.2   1475.43  4
198     14068078        GRU    32  2.0      3  0.0   1223.56  4
196     13585678        GRU    32  2.0      1  0.0    754.08  4
197     13826878        GRU    32  2.0      2  0.0   875.17  4

```

You can see from these experiments that GRUs are your best bet for creating language models that understand text well enough to predict the next word. Surprisingly GRUs do not need as many layers as other RNN architectures to achieve the same accuracy. And they take less time to train than RNNs to achieve comparable accuracy.

8.4.3 Long and Short-Term Memory (LSTM)

An LSTM neuron adds two more internal gates in an attempt to improve both long and short term memory capacity of an RNN. An LSTM retains the update and relevance gates but adds new gates for forgetting and the output gate. four internal gates, each with a different purpose. The first one is just the normal activation function that you are familiar with.

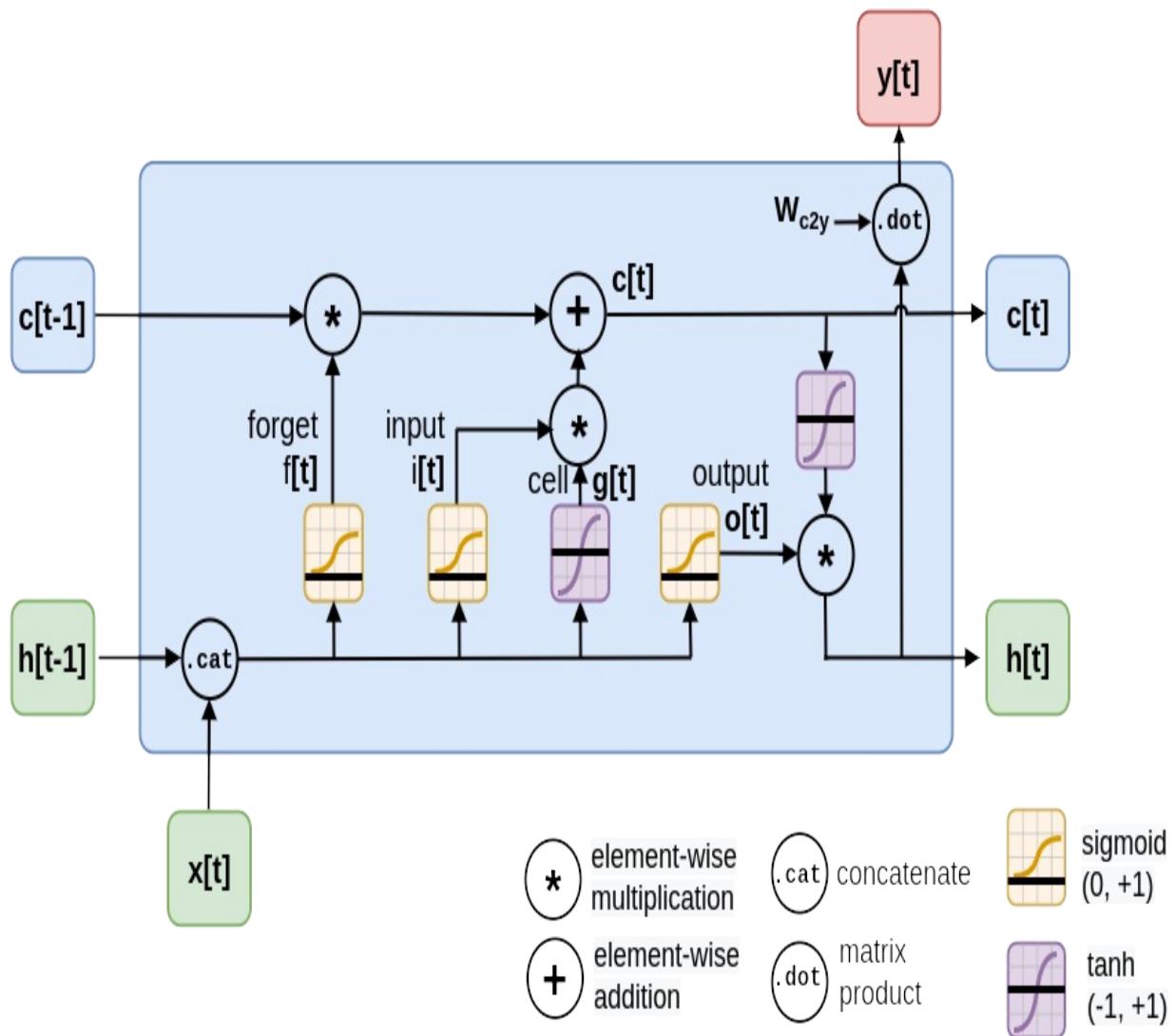
1. Forgetting gate (f): Whether to completely ignore some element of the hidden layer to make room in memory for future more important tokens.
2. Input or update gate (i): What parts of the hidden layer should matter to the current output (now, at time t).
3. Relevance or cell gate (c): What parts of the hidden layer should be blocked because they are not longer relevant to the current output.
4. Output gate (o): What parts of the hidden layer should be output, both to the neurons output as well as to the hidden layer for the next token in the

text.

But what about that unlabeled tanh activation function at upper right of figure 8.12? That's just the original output activation used to create the hidden state vector from the cell state. The hidden state vector holds information about the most recently processed tokens; it's the short term memory of the LSTM. The cell state vector holds a representation of the meaning of the text over the long term, since the beginning of a document.

In Figure 8.13 you can see how these four logic gates fit together. The various weights and biases required for each of logic gates are hidden to declutter the diagram. You can imagine the weight matrix multiplications happen within each of the activation functions that you see in the diagram. Another thing to notice is that the hidden state is not the only recurrent input and output. You've now got another encoding or state tensor called the *cell state*. As before, you only need the hidden state to compute the output at each time step. But the new cell state tensor is where the long and short term memories of past patterns are encoded and stored to be reused on the next token.

Figure 8.12. LSTMs add a forgetting gate and a cell output



One thing in this diagram that you'll probably only see in the smartest blog posts is the explicit linear weight matrix needed to compute the output tensor. [322] Even the PyTorch documentation glosses over this tidbit. You'll need to add this fully connected linear layer yourself at whichever layer you are planning to compute predictions based on your hidden state tensor. The PyTorch

You're probably saying to yourself "wait, I thought all hidden states (encodings) were the same, why do we have this new *cell state* thing?" Well that's the long term memory part of an LSTM. The cell state is maintained separately so the logic gates can remember things and store them there, without having to mix them in with the shorter term memory of the hidden state tensor. And the cell state logic is a bit different from the hidden state

logic. It's designed to be selective in the things it retrans to keep room for things it learns about the text long before it reaches the end of the string.

The formulas for computing the LSTM logic gates and outputs are very similar to those for the GRU. The main difference is the addition of 3 more functions to compute all the signals you need. And some of the signals have been rerouted to create a more complicated network for storing more complex patterns of connections between long and short term memory of the text. It's this more complicated interaction between hidden and cell states that creates more "capacity" or memory and computation in one cell. Because an LSTM cell contains more nonlinear activation functions and weights it has more information processing capacity.

```
r = sigmoid(w_i2r.mm(x) + b_i2r + w_h2r.mm(h) + b_h2r)      #1
z = sigmoid(w_i2z.mm(x) + b_i2z + w_h2z.mm(h) + b_h2z)      #2
n = tanh(w_i2n.mm(x) + b_i2n + r*(w_h2n.mm(h) + b_h2n))    #3

f = sigmoid(w_i2f.mm(x) + b_i2f + w_h2f.mm(h) + b_h2f)      #1
i = sigmoid(w_i2i.mm(x) + b_i2i + w_h2i.mm(h) + b_h2i)      #2
g = tanh(w_i2g.mm(x) + b_i2g + w_h2g.mm(h) + b_h2g)        #3
o = sigmoid(w_i2o.mm(x) + b_i2o + w_h2o.mm(h) + b_h2o)      #4
c = f*c + i*g      #5
h = o*tanh(c)
```

8.4.4 Give your RNN a tuneup

As you learned in chapter 7, hyperparameter tuning becomes more and more important as your neural networks get more and more complicated. Your intuitions about layers and network capacity and training time all get a fuzzier and fuzzier as the models get complicated. RNNs are particularly intuitive. To jump start your intuition we've trained dozens of different basic RNNs with different combinations of hyperparameters such as the number of layers and number of hidden units in each layer. You can explore all the hyperparameters that you are curious about using the code in `nlpia2/ch08`. [\[323\]](#)

```
import pandas as pd
import jsonlines

with jsonlines.open('experiments.jsonl') as fin:
    lines = list(fin)
```

```

df = pd.DataFrame(lines)
df.to_csv('experiments.csv')
cols = 'rnn_type epochs lr num_layers dropout epoch_time test_loss'
cols = cols.split()
df[cols].round(2).sort_values('test_loss').head(10)

   epochs    lr  num_layers  dropout  epoch_time  test_loss
37      12  2.0          2      0.2       35.43     5.23
28      12  2.0          1      0.0       22.66     5.23
49      32  0.5          2      0.0       32.35     5.22
57      32  0.5          2      0.2       35.50     5.22
38      12  2.0          3      0.2       46.14     5.21
50      32  0.5          3      0.0       37.36     5.20
52      32  2.0          1      0.0       22.90     5.10
55      32  2.0          5      0.0       56.23     5.09
53      32  2.0          2      0.0       32.49     5.06
54      32  2.0          3      0.0       38.78     5.04

```

It's really exciting thing to explore the hyperspace of options like this and discover surprising tricks for building accurate models. Surprisingly, for this RNN language model trained on small subset of Wikipedia, you can get great results without maximizing the size and capacity of the model. You can achieve better accuracy with a 3-layer RNN than a 5-layer RNN. You just need to start with an aggressive learning rate and keep the dropout to a minimum. And the fewer layers you have the faster the model will train.



Tip

Experiment often, and always document what things your tried and how well the model worked. This kind of hands-on work provides the quickest path toward an intuition that speeds your model building and learning. Your lifelong goal is to train your mental model to predict which hyperparameter values will produce the best results in any given situation.

If you feel the model is overfitting the training data but you can't find a way to make your model simpler, you can always try increasing the Dropout (percentage). This is a sledgehammer that reduce overfitting while allowing your model to have as much complexity as it needs to match the data. If you set the dropout percentage much above 50%, the model starts to have a difficult time learning. Your learning will slow and validation error

may bounce around a lot. But 20% to 50% is a pretty safe range for a lot of RNNs and most NLP problems.

If you're like Cole and I when we were getting started in NLP, you're probably wondering what a "unit" is. All the previous deep learning models have used "neurons" as the fundamental units of computation within a neural network. Researchers use the more general term "unit" to describe the elements of an LSTM or GRU that contain internal gates and logic. So when you are thinking about how many units to add to your neural network to solve a particular problem, each LSTM or GRU unit gives your network a capacity similar to two "normal" RNN neurons or hidden vector dimensions. A unit is just a more complicated, higher capacity neuron, and you can see this if you count up the number of "learned parameters" in your LSTM model and compare it to those of an equivalent RNN.

[317] "Attention Is All You Need" by Ashish Vaswani et al
(<https://arxiv.org/abs/1706.03762>)

[318] https://en.wikipedia.org/wiki/Hutter_Prize

[319] The full source code is in the nlpia2 package
(https://gitlab.com/tangibleai/nlpia2/-/blob/main/src/nlpia2/ch08/rnn_word/data.py)

[320] PyTorch docs for GRU layers
(<https://pytorch.org/docs/stable/generated/torch.nn.GRU.html#torch.nn.GRU>)

[321] PyTorch docs discussion about counting up learned parameters
(<https://discuss.pytorch.org/t/how-do-i-check-the-number-of-parameters-of-a-model/4325/9>)

[322] Thank you Rian Dolphin for your rigorous explanation
(<https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9>)

[323] The hypertune.py script in the ch08/rnn_word module within the nlpia2 Python package https://gitlab.com/tangibleai/nlpia2/-/blob/main/src/nlpia2/ch08/rnn_word/hypertune.py

8.5 Predicting

The word based RNN language model you trained for this chapter used the WikiText - 2 corpus.[\[324\]](#) The nice thing about working with this corpus is that it is often used by researchers to benchmark their language model accuracy against. And the Wikipedia article text has already been tokenized for you. And the uninteresting sections such as the References at the end of the articles have been removed.

Unfortunately the PyTorch version of the WikiText-2 includes "<unk>" tokens that randomly replace, or mask, 2.7% of the tokens. That means that your model will never get very high accuracy unless there is some predictable pattern that determined which tokens were masked with "<unk>". But if you download the original raw text without the masking tokens you can train your language model on it and get a quick boost in accuracy.[\[325\]](#) And you can compare the accuracy of your LSTM and GRU models to those of the experts that use this benchmark data.[\[326\]](#)

Here is an example paragraph at the end of the masked training dataset `train.txt`.

```
>>> from nlpia2.ch08.rnn_word.data import Corpus
>>> corpus = Corpus('data/wikitext-2')
>>> passage = corpus.train.numpy()[-89:-35]

>>> ' '.join([vocab.idx2word[i] for i in passage])
Their ability at mimicry is so great that strangers have looked i
for the human they think they have just heard speak . <eos>
Common starlings are trapped for food in some Mediterranean count
The meat is tough and of low quality , so it is <unk> or made int
```

It seems that the last Wikipedia article in the WikiText-2 benchmark corpus is about the common starling (a small bird in Europe). And from the article, it seems that the starling appears to be good at mimicking human speech, just as your RNN can.

What about those "<unk>" tokens? These are designed to test machine learning models. Language models are trained with the goal of predicting the

words that were replaced with the "<unk>" (unknown) tokens. Because you have a pretty good English language model in your brain you can probably predict the tokens that have been masked out with all those "<unk>" tokens.

But if your machine learning model you are training thinks these are normal English words, you may confuse it. The RNN you are training in this chapter is trying to discern the *meaning* of the meaningless "<unk>" token, and this will reduce its understanding of all other words in the corpus.



Tip

If you want to avoid this additional source of error and confusion, you can try training your RNN on the unofficial raw text for the wikitext-2 benchmark. There is a one-to-one correspondence between the tokens of the official wikitext-2 corpus and the unofficial raw version in the nlpia2 repository. [\[327\]](#)

So how many "<eos>" and "<unk>" tokens are there in this training set?

```
>>> num_eos = sum([vocab.idx2word[i] == '<eos>' for i in corpus.t
>>> num_eos
36718
>>> num_unk = sum([vocab.idx2word[i] == '<unk>' for i in corpus.t
>>> num_unk
54625
>>> num_normal = sum([
...     vocab.idx2word[i] not in ('<unk>', '<eos>')
...     for i in corpus.train.numpy()])
>>> num_normal
1997285
>>> num_unk / (num_normal + num_eos + num_unk)
0.0261...
```

So 2.6% of the tokens have been replaced with the meaningless "<unk>" token. And the "<eos>" token marks the newlines in the original text, which is typically the end of a paragraph in a Wikipedia article.

So lets see how well it does at writing new sentences similar to those in the WikiText-2 dataset, including the "<unk>" tokens. We'll prompt the model to start writing with the word "The" to find out what's on the top of its "mind".

```

>>> import torch
>>> from preprocessing import Corpus
>>> from generate import generate_words
>>> from model import RNNModel

>>> corpus = Corpus('data/wikitext-2')
>>> vocab = corpus.dictionary
>>> with open('model.pt', 'rb') as f:
...     orig_model = torch.load(f, map_location='cpu')      #1

>>> model = RNNModel('GRU', vocab=corpus.dictionary, num_layers=1
>>> model.load_state_dict(orig_model.state_dict())
>>> words = generate_words(
...     model=model, vocab=vocab, prompt='The', temperature=.1)

>>> print(' '.join(w for w in words))
...
= = Valkyria Valkyria Valkyria Valkyria = = The kakapo is a comm
...

```

The first line in the training set is "= Valkyria Chronicles III =" and the last article in the training corpus is titled "= Common starling =". So this GRU remembers how to generate text similar to text at the beginning and end of the text passages it has read. So it surely seems to have both long and short term memory capability. This is exciting, considering we only trained a very simple model on a very small dataset. But this GRI doesn't yet seem to have the capacity to store all of the English language patterns that it found in the two million token long sequence. And it certainly isn't going to do any sense-making any time soon.



Note

Sense-making is the way people give meaning to the experiences that they share. When you try to explain to yourself why others are doing what they are doing, you are doing sense-making. And you don't have to do it alone. A community can do it as a group through public conversation mediated by social media apps and even conversational virtual assistants. That's why it's often called "collective sense-making." Startups like DAOStack are experimenting with chatbots that bubble up the best ideas of a community and use them for building knowledge bases and making decisions. [\[328\]](#)

You now know how to train a versatile NLP language model that you can use on word-level or character-level tokens. And you can use these models to classify text or even generate modestly interesting new text. And you didn't have to go crazy on expensive GPUs and servers.

[324] PyTorch torchtext dataset

(<https://pytorch.org/text/0.8.1/datasets.html#wikitext-2>)

[325] Raw, unmasked text with "answers" for all the "unk" tokens

(<https://s3.amazonaws.com/research.metamind.io/wikitext/wikitext-2-raw-v1.zip>)

[326] AI researchers(<https://www.salesforce.com/products/einstein/ai-research/the-wikitext-dependency-language-modeling-dataset/>)

[327] nlpia2 package with code and data for the rnn_word model code and datasets used in this chapter (https://gitlab.com/tangibleai/nlpia2/-/tree/main/src/nlpia2/ch08/rnn_word/data/wikitext-2)

[328] DAOStack platform for decentralized governance

(<https://daostack.io/deck/DAOstack-Deck-ru.pdf>)

8.6 Review

- What are some tricks to improve "retention" for reading long documents with an RNN?
- What are some "unreasonably effective" applications for RNNs in the real world?
- How could you use a name classifier for good? What are some unethical uses of a name classifier?
- What are some ethical and prosocial AI uses for a dataset with millions of username-password pairs such as Mark Burnett's password dataset?
[329]
- Train an rnn_word model on the raw text, unmasked text for the Wikitext-2 dataset the proportion of tokens that are "<unk>". Did this improve the accuracy of your word-level RNN language model?
- Modify the dataset to label each name with a multihot tensor indicating

all the nationalities for each name.^[330] ^[331] How should you measure accuracy? Does your accuracy improve?

^[329] Alexander Fishkov's analysis (<https://www.city-data.com/blog/1424-passwords-on-the-internet-publicly-available-dataset/>) of Mark Burnett's (<https://xato.net/author/mb/>) ten million passwords (<https://xato.net/passwords/ten-million-passwords>)

^[330] PyTorch community multi-label (tagging) data format example (<https://discuss.pytorch.org/t/multi-label-classification-in-pytorch/905/45>)

^[331] Example torchtext Dataset class multi-label text classification (<https://discuss.pytorch.org/t/how-to-do-multi-label-classification-with-torchtext/11571/3>)

8.7 Summary

- In natural language token sequences, an RNN can remember everything it has read up to that point, not just a limited window.
- Splitting a natural language statement along the dimension of time (tokens) can help your machine deepen its understanding of natural language.
- You can backpropagate errors back in time (token) as well as in the layers of a deep learning network.
- Because RNNs are particularly deep neural nets, RNN gradients are particularly temperamental, and they may disappear or explode.
- Efficiently modeling natural language character sequences wasn't possible until recurrent neural nets were applied to the task.
- Weights in an RNN are adjusted in aggregate across time for a given sample.
- You can use different methods to examine the output of recurrent neural nets.
- You can model the natural language sequence in a document by passing the sequence of tokens through an RNN backward and forward in time simultaneously.

9 Stackable deep learning (Transformers)

This chapter covers

- Understanding what makes transformers so powerful
- Seeing how transformers enable limitless "stacking" options for NLP
- Encoding text to create meaningful vector representations
- Decoding semantic vectors to generate text
- Finetuning transformers (BERT, GPT) for your application
- Applying transformers to extractive and abstraction summarization of long documents
- Generating grammatically correct and interesting text with transformers
- Estimating the information capacity of a transformer network required for a particular problem

Transformers are changing the world. The increased intelligence that transformers bring to AI is transforming culture, society, and the economy. For the first time, transformers are making us question the long-term economic value of human intelligence and creativity. And the ripple effects of transformers go deeper than just the economy. Transformers are changing not only how we work and play, but even how we think, communicate, and create. Within less than a year, transformer-enabled AI known as Large Language Models (LLMs) created whole new job categories such as prompt engineering and real-time content curation and fact-checking (grounding). Tech companies are racing to recruit engineers that can design effective LLM prompts and incorporate LLMs into their workflows. Transformers are automating and accelerating productivity for information economy jobs that previously required a level of creativity and abstraction out of reach for machines.

As transformers automate more and more information economy tasks, workers are reconsidering whether their jobs are as essential to their employers as they thought. For example, influential CyberSecurity experts

are bragging about augmenting their thinking, planning, and creativity with the help of dozens of ChatGPT suggestions every day.^[332] Microsoft News and the MSN.com website laid off its journalists in 2020, replacing them with transformer models capable of curating and summarizing news articles automatically. This race to the bottom (of the content quality ladder) probably won't end well for media companies or their advertisers and employees.

In this chapter, you will learn how to use transformers to *improve* the accuracy and thoughtfulness of natural language text. Even if your employer tries to program away your job, you will know how to program transformers to create new opportunities for yourself. Program or be programmed. Automate or be automated.

And transformers are your best choice not only for natural language generation, but also for natural language understanding. Any system that relies on a vector representation of meaning can benefit from transformers.

- At one point Replika used GPT-3 to generate more than 20% of its replies
- Qary uses BERT to generate open domain question answers
- Google uses models based on BERT to improve search results and query a knowledge graph
- nboost uses transformers to create a semantic search proxy for ElasticSearch
- aidungeon.io uses GPT-3 to generate an endless variety of rooms
- Most vector databases for semantic search rely on transformers.^[333]

Even if you only want to get good at prompt engineering, your understanding of transformers will help you design prompts for LLMs that avoid the holes in LLM capabilities. And LLMs are so full of holes that engineers and statisticians often use the swiss cheese model when thinking about how LLMs fail.^[334] The conversational interface of LLMs makes it easy to learn how to cajole the snarky conversational AI systems into doing valuable work. People that understand how LLMs work and can fine tune them for their own applications, those people will have their hands at the helm of a powerful machine. Imagine how sought-after you'd be if you could build a "TutorGPT" that can help students solve arithmetic and math word problems.

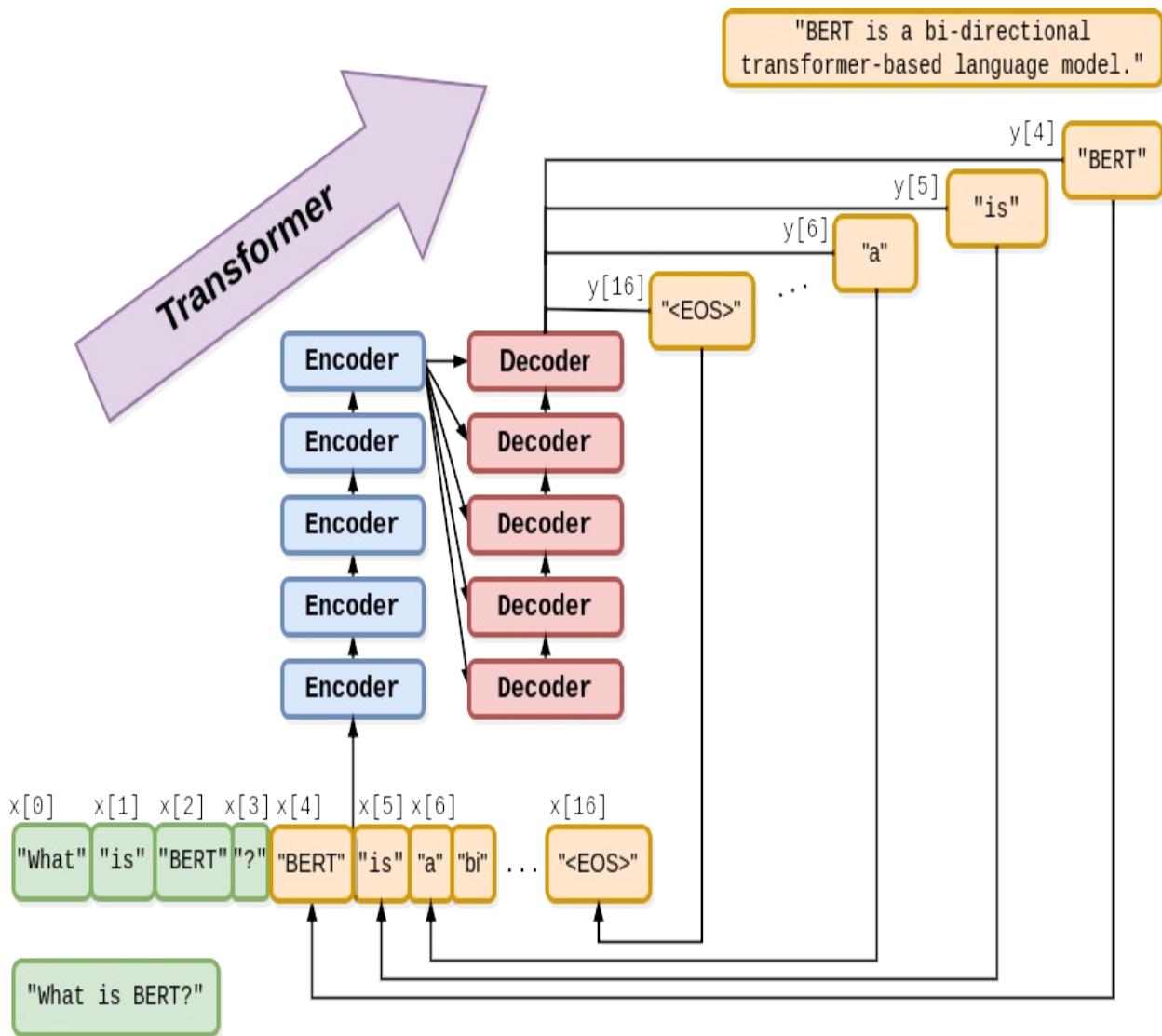
Shabnam Aggarwal at Rising Academies in Kigali is doing just that with her Rori.AI WhatsApp math tutor bot for middle school students.[\[335\]](#) [\[336\]](#) And Vishvesh Bhat did this for college math students as a passion project.[\[337\]](#) [\[338\]](#)

9.1 Recursion vs recurrence

Transformers are the latest big leap forward in auto-regressive NLP models. Auto-regressive models predict one discrete output value at a time, usually a token or word in natural language text. An autoregressor recycles the output to reuse it as an input for predicting the next output so auto-regressive neural networks are *recursive*. The word "recursive" is a general term for any recycling of outputs back into the input, a process that can continue indefinitely until an algorithm or computation "terminates." A recursive function in computer science will keep calling itself until it achieves the desired result.

But transformers are recursive in a bigger and more general way than *Recurrent Neural Networks*. Transformers are called *recursive* NNs rather than *recurrent* NNs because *recursive* is a more general term for any system that recycles the input.[\[339\]](#) The term *recurrent* is used exclusively to describe RNNs such as LSTMs and GRUs where the individual neurons recycle their outputs into the same neuron's input for each step through the sequence tokens.

Transformers are a *recursive* algorithm but do not contain *recurrent* neurons. As you learned in Chapter 8, recurrent neural networks recycle their output within each individual neuron or RNN *unit*. But transformers wait until the very last layer to output a token embedding that can be recycled back into the input. The entire transformer network, both the encoder and the decoder, must be run to predict each token so that token can be used to help it predict the next one. In the computer science world, you can see that a transformer is one big recursive function calling a series of nonrecursive functions inside. The whole transformer is run *recursively* to generate one token at a time.



Because there is no recurrence within the inner guts of the transformer it doesn't need to be "unrolled." This gives transformers a huge advantage over RNNs. The individual neurons and layers in a transformer can be run in parallel all at once. For an RNN, you had to run the functions for the neurons and layers one at a time in sequence. *Unrolling* all these recurrent function calls takes a lot of computing power and it must be performed in order. You can't skip around or run them in parallel. They must be run sequentially all the way through the entire text. A transformer breaks the problem into a much smaller problem, predicting a single token at a time. This way all the neurons of a transformer can be run in parallel on a GPU or multi-core CPU to dramatically speed up the time it takes to make a prediction.

They use the last predicted output as the input to predict the next output. But transformers are *recursive* not *recurrent*. Recurrent neural networks (RNNs) include variational autoencoders, RNNs, LSTMs, and GRUs. When researchers combine five NLP ideas to create the transformer architecture, they discovered a total capability that was much greater than the sum of its parts. Let's looks at these ideas in detail.

[332] For months following ChatGPT's public release, Dan Miessler spent almost half of his "Unsupervised Learning" podcasts discussing transformer-based tools such as InstructGPT, ChatGPT, Bard and Bing
(<https://danielmiessler.com/>)

[333] PineCone.io, Milvus.io, Vespa.ai, Vald.vdaas.org use transformers

[334] "Swiss cheese model" on Wikipedia
(https://en.wikipedia.org/wiki/Swiss_cheese_model)

[335] Sebastian Larson, an actual middle schooler, won our competition to develop Rori's mathtext NLP algorithm
(<https://gitlab.com/tangibleai/community/team/-/tree/main/exercises/2-mathtext>)

[336] All of Rori.AI's NLP code is open source and available on Huggingface (<https://huggingface.co/spaces/TangibleAI/mathtext-fastapi>).

[337] Vish built an transformer-based teaching assistant called Clevrly (clevrly.io)

[338] Some of Vish's fine tuned transformers are available on Huggingface (<https://huggingface.co/clevrly>)

[339] Stats Stack Exchange answer
(<https://stats.stackexchange.com/a/422898/15974>)

9.2 Attention is NOT all you need

- Byte pair encoding (BPE):: Tokenizing words based on character

- sequence statistics rather than spaces and punctuation
- *Attention*:: Connecting important word patterns together across long stretches of text using a connection matrix (attention)
- *Positional encoding*:: Keeping track of where each token or pattern is located within the token sequence

Byte pair encoding (BPE) is an often overlooked enhancement of transformers. BPE was originally invented to encode text in a compressed binary (byte sequence) format. But BPE really came into its own when it was used as a tokenizer in NLP pipelines such as search engines. Internet search engines often contain millions of unique words in their vocabulary. Imagine all the important names a search engine is expected to understand and index. BPE can efficiently reduce your vocabulary by several orders of magnitude. The typical transformer BPE vocabulary size is only 5000 tokens. And when you're storing a long embedding vector for each of your tokens, this is a big deal. A BPE vocabulary trained on the entire Internet can easily fit in the RAM of a typical laptop or GPU.

Attention gets most of the credit for the success of transformers because it made the other parts possible. The attention mechanism is a much simpler approach than the complicated math (and computational complexity) of CNNs and RNNs. The attention mechanism removes the recurrence of the encoder and decoder networks. So a transformer has neither the vanishing gradients nor the exploding gradients problem of an RNN. Transformers are limited in the length of text they can process because the attention mechanism relies on a fixed-length sequence of embeddings for both the inputs and outputs of each layer. The attention mechanism is essentially a single CNN kernel that spans the entire sequence of tokens. Instead of rolling across the text with convolution or recurrence, the attention matrix is simply multiplied once by the entire sequence of token embeddings.

The loss of recurrence in a transformer creates a new challenge because the transformer operates on the entire sequence all at once. A transformer is *reading* the entire token sequence all at once. And it outputs the tokens all at once as well, making bi-directional transformers an obvious approach. Transformers do not care about the normal causal order of tokens while it is reading or writing text. To give transformers information about the causal

sequence of tokens, positional encoding was added. And it doesn't even require additional dimensions within the vector embedding, positional encoding is spread out over the entire embedding sequence by multiplying them by the sine and cosine functions. Positional encoding enables nuanced adjustment to a transformer's understanding of tokens depending on their location in a text. With positional encoding, the word "sincerely" at the beginning of an email has a different meaning than it does at the end of an email.

Limiting the token sequence length had a cascading effect of efficiency improvements that give transformers an unexpectedly powerful advantage over other architectures: *scalability*. BPE plus *attention* plus positional encoding combine together to create unprecedented scalability. These three innovations and simplifications of neural networks combined to create a network that is both much more stackable and much more parallelizable.

- *Stackability*:: The inputs and outputs of a transformer layer have the exact same structure so they can be stacked to increase capacity
- *Parallelizability*:: The cookie cutter transformer layers all rely heavily on large matrix multiplications rather than complex recurrence and logical switching gates

This stackability of transformer layers combined with the parallelizability of the matrix multiplication required for the attention mechanism creates unprecedented scalability. And when researchers tried out their large-capacity transformers on the largest datasets they could find (essentially the entire Internet), they were taken aback. The extremely large transformers trained on extremely large datasets were able to solve NLP problems previously thought to be out of reach. Smart people are beginning to think that world-transforming conversational machine intelligence (AGI) may only be years away, if it isn't already upon us.

9.3 Much attention about everything

You might think that all this talk about the power of attention is much ado about nothing. Surely transformers are more than just a simple matrix multiplication across every token in the input text. Transformers combine

many other less well-known innovations such as BPE, self-supervised training, and positional encoding. But the attention matrix was the connector between all these ideas that helped them work together effectively. And the attention matrix enables a transformer to accurately model the connections between *all* the words in a long body of text, all at once.

As with CNNs and RNNs (LSTMs & GRUs), each layer of a transformer gives you a deeper and deeper representation of the *meaning* or *thought* of the input text. But unlike CNNs and RNNs, the transformer layer outputs an encoding that is the exact same size and shape as the previous layers. Likewise for the decoder, a transformer layer outputs a fixed-size sequence of embeddings representing the semantics (meaning) of the output token sequence. The outputs of one transformer layer can be directly input into the next transformer layer making the layers even more *stackable* than CNN's. And the attention matrix within each layer spans the entire length of the input text, so each transformer layer has the same internal structure and math. You can stack as many transformer encoder and decoder layers as you like creating as deep a neural network as you need for the information content of your data.

Every transformer layer outputs a consistent *encoding* with the same size and shape. Encodings are just embeddings but for token sequences instead of individual tokens. In fact, many NLP beginners use the terms "encoding" and "embedding" interchangeably, but after this chapter, you will understand the difference. The word "embedding", used as a noun, is 3 times more popular than "encoding", but as more people catch up with you in learning about transformers that will change. [\[340\]](#) If you don't need to make it clear which ones you are talking about you can use "semantic vector", a term you learned in Chapter 6.

Like all vectors, encodings maintain a consistent structure so that they represent the meaning of your token sequence (text) in the same way. And transformers are designed to accept these encoding vectors as part of their input to maintain a "memory" of the previous layers' understanding of the text. This allows you to stack transformer layers with as many layers as you like if you have enough training data to utilize all that capacity. This "scalability" allows transformers to break through the diminishing returns

ceiling of RNNs.

And because the attention mechanism is just a connection matrix, it can be implemented as a matrix multiplication with a PyTorch `Linear` layer. Matrix multiplications are parallelized when you run your PyTorch network on a GPU or multicore CPU. This means that much larger transformers can be parallelized and these much larger models can be trained much faster.
Stackability plus Parallelizability equals Scalability.

Transformer layers are designed to have inputs and outputs with the same size and shape so that the transformer layers can be stacked like Lego bricks that all have the same shape. The transformer innovation that catches most researchers' attention is the attention mechanism. Start there if you want to understand what makes transformers so exciting to NLP and AI researchers. Unlike other deep learning NLP architectures that use recurrence or convolution, the transformer architecture uses stacked blocks of attention layers which are essentially fully-connected feedforward layers with the same.

In chapter 8, you used RNNs to build encoders and decoders to transform text sequences. In encoder-decoder (*transcoder* or *transduction*) networks, [341] the encoder processes each element in the input sequence to distill the sentence into a fixed-length thought vector (or context vector). That thought vector can then be passed on to the decoder where it is used to generate a new sequence of tokens.

The encoder-decoder architecture has a big limitation—it can't handle longer texts. If a concept or thought is expressed in multiple sentences or a long complex sentence, then the encoded thought vector fails to accurately encapsulate *all* of that thought. The attention mechanism presented by Bahdanau et al [342] to solve this issue is shown to improve sequence-to-sequence performance, particularly on long sentences, however it does not alleviate the time sequencing complexity of recurrent models.

The introduction of the *transformer* architecture in "Attention Is All You Need" [343] propelled language models forward and into the public eye. The transformer architecture introduced several synergistic features that worked

together to achieve as yet impossible performance:

The most widely recognized innovation in the transformer architecture is *self-attention*. Similar to the memory and forgetting gates in a GRU or LSTM, the attention mechanism creates connections between concepts and word patterns within a lengthy input string.

In the next few sections, you'll walk through the fundamental concepts behind the transformer and take a look at the architecture of the model. Then you will use the base PyTorch implementation of the Transformer module to implement a language translation model, as this was the reference task in "Attention Is All You Need", to see how it is both powerful and elegant in design.

9.3.1 Self-attention

When we were writing the first edition of this book, Hannes and Cole (the first edition coauthors) were already focused on the attention mechanism. It's now been 6 years and attention is still the most researched topic in deep learning. The attention mechanism enabled a leap forward in capability for problems where LSTMs struggled:

- *Conversation*—Generate plausible responses to conversational prompts, queries, or utterances.
- Abstractive summarization or paraphrasing:: Generate a new shorter wording of a long text summarization of sentences, paragraphs, and even several pages of text.
- Open domain question answering:: Answering a general question about anything the transformer has ever read.
- Reading comprehension question answering:: Answering questions about a short body of text (usually less than a page).
- *Encoding*:: A single vector or sequence of embedding vectors that represent the meaning of body of text in a vector space—sometimes called task-independent sentence embedding.
- Translation and code generation—Generating plausible software expressions and programs based on plain English descriptions of the program's purpose.

Self-attention is the most straight-forward and common way to implement attention. It takes the input sequence of embedding vectors and puts them through linear projections. A linear projection is merely a dot product or matrix multiplication. This dot product creates key, value and query vectors. The query vector is used along with the key vector to create a context vector for the words' embedding vectors and their relation to the query. This context vector is then used to get a weighted sum of values. In practice, all these operations are done on sets of queries, keys, and values packed together in matrices, Q , K , and V , respectively.

There are two ways to implement the linear algebra of an attention algorithm: additive attention or dot-product attention. The one that was most effective in transformers is a scaled version of dot-production attention. For dot-product attention, the scalar products between the query vectors Q and the key vectors K , are scaled down based on how many dimensions there are in the model. This makes the dot product more numerically stable for large dimensional embeddings and longer text sequences. Here's how you compute the self-attention outputs for the query, key, and value matrices Q , K , and V .

Equation 9.1 Self-attention outputs

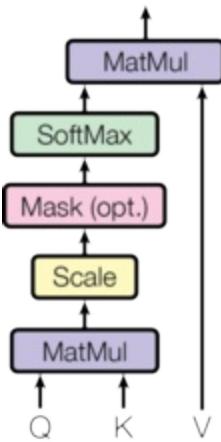
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The high dimensional dot products create small gradients in the softmax due to the law of large numbers. To counteract this effect, the product of the query and key matrices is scaled by

$$\frac{1}{\sqrt{d_k}}$$

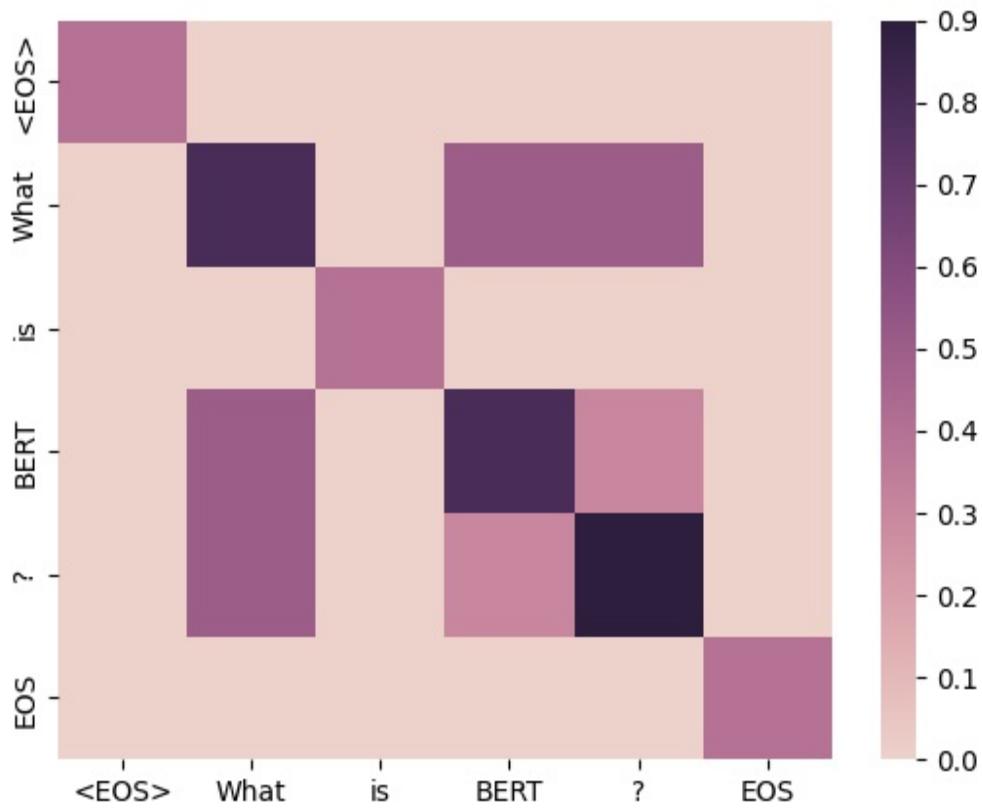
The softmax normalizes the resulting vectors so that they are all positive and sum to 1. This "scoring" matrix is then multiplied with the values matrix to get the weighted values matrix in figure [Figure 9.1](#). [\[344\]](#) [\[345\]](#)

Figure 9.1. Scaled dot product attention



Unlike, RNNs where there is recurrence and shared weights, in self-attention all of the vectors used in the query, key, and value matrices come from the input sequences' embedding vectors. The entire mechanism can be implemented with highly optimized matrix multiplication operations. And the $Q K$ product forms a square matrix that can be understood as the connection between words in the input sequence. A toy example is shown in figure [Figure 9.2](#).

Figure 9.2. Encoder attention matrix as connections between words



9.3.2 Multi-Head Self-Attention

Multi-head self-attention is an expansion of the self-attention approach to creating multiple attention heads that each attend to different aspects of the words in a text. So if a token has multiple meanings that are all relevant to the interpretation of the input text, they can each be accounted for in the separate attention heads. You can think of each attention head as another dimension of the encoding vector for a body of text, similar to the additional dimensions of an embedding vector for an individual token (see Chapter 6). The query, key, and value matrices are multiplied n (n_heads , the number of attention heads) times by each different d_q , d_k , and d_v dimension, to compute the total attention function output. The n_heads value is a hyperparameter of the transformer architecture that is typically small, comparable to the number of transformer layers in a transformer model. The d_v -dimensional outputs are concatenated and again projected with a W^o matrix as shown in the next

equation.

Equation 9.2 Multi-Head self-attention

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_n)W^o$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$

The multiple heads allow the model to focus on different positions, not just ones centered on a single word. This effectively creates several different vector subspaces where the transformer can encode a particular generalization for a subset of the word patterns in your text. In the original transformers paper, the model uses $n=8$ attention heads such that

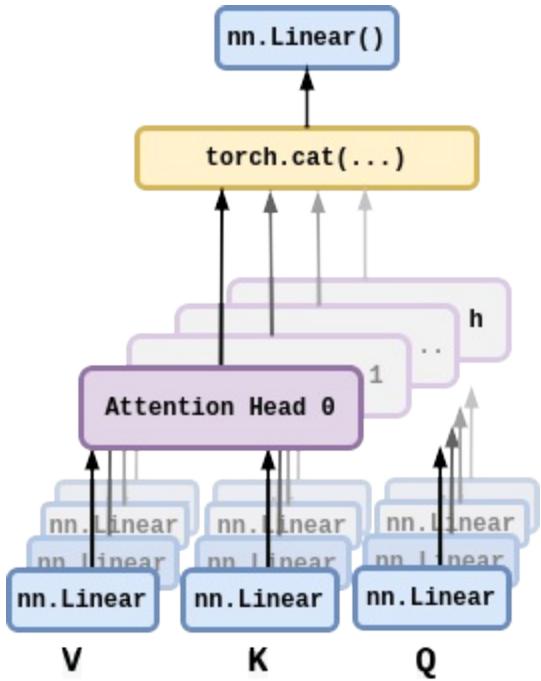
$$d_k = d_v = \frac{d_{\text{model}}}{n} = 64$$

The reduced dimensionality in the multi-head setup is to ensure the computation and concatenation cost is nearly equivalent to the size of a full-dimensional single-attention head.

If you look closely you'll see that the attention matrices (attention heads) created by the product of Q and K all have the same shape, and they are all square (same number of rows as columns). This means that the attention matrix merely rotates the input sequence of embeddings into a new sequence of embeddings, without affecting the shape or magnitude of the embeddings. And this makes it possible to explain a bit about what the attention matrix is doing for a particular example input text.

This allows them to each This is because it needs It turns out, the multi-head attention layer acts a lot like a fully connected linear layer.

Figure 9.3. Multi-Head Self-Attention



It turns out, the multi-head attention mechanism is just a fully connected linear layer under the hood. After all is said and done, the deepest of the deep learning models turned to be nothing more than a clever stacking of what is essentially linear and logistic regressions. This is why it was so surprising that transformers were so successful. And this is why it was so important for you to understand the basics of linear and logistic regression described in earlier chapters.

[340] N-Gram Viewer query "embedding_NOUN" / "encoding_NOUN"
[https://books.google.com/ngrams/graph?
content=embedding_NOUN+%2F+encoding_NOUN&year_start=2010&year_end=2019&smoothing=3](https://books.google.com/ngrams/graph?content=embedding_NOUN+%2F+encoding_NOUN&year_start=2010&year_end=2019&smoothing=3)

[341] "Gentle Introduction to Transduction in Machine Learning" blog post on *Machine Learning Mastery* by Jason Brownlee 2017
<https://machinelearningmastery.com/transduction-in-machine-learning/>

[342] Neural Machine Translation by Jointly Learning to Align and Translate:
<https://arxiv.org/abs/1409.0473>

[343] "Attention Is All You Need" by Vaswani, Ashish et al. 2017 at Google Brain and Google Research (<https://arxiv.org/abs/1706.03762>)

[344] "Scaled dot product attention from scratch" by Jason Brownlee
(<https://machinelearningmastery.com/how-to-implement-scaled-dot-product-attention-from-scratch-in-tensorflow-and-keras/>)

[345] "Attention is all you Need" by Ashish Vaswani et al 2017
(<https://arxiv.org/abs/1706.03762>)

9.4 Filling the attention gaps

The attention mechanism compensates for some problems with RNNs and CNNs of previous chapters but creates some additional challenges. Encoder-decoders based on RNNs don't work very well for longer passages of text where related word patterns are far apart. Even long sentences are a challenge for RNNs doing translation.^[346] And the attention mechanism compensates for this by allowing a language model to pick up important concepts at the beginning of a text and emphasize connect them to text that is towards the end. The attention mechanism gives the transformer a way to reach back to any word it has ever seen. Unfortunately, adding the attention mechanism forces you to remove all recurrence from the transformer.

CNNs are another way to connect concepts that are far apart in the input text. A CNN can do this by creating a hierarchy of convolution layers that progressively "necks down" the encoding of the information within the text it is processing. And this hierarchical structure means that a CNN has information about the large-scale position of patterns within a long text document. Unfortunately, the outputs and the inputs of a convolution layer usually have different shapes. So CNNs are not stackable, making them tricky to scale up for greater capacity and larger training datasets. So to give a transformer the uniform data structure it needs for stackability, transformers use byte pair encoding and positional encoding to spread the semantic and position information uniformly across the encoding tensor.

9.4.1 Positional encoding

Word order in the input text matters, so you need a way to bake in some positional information into the sequence of embeddings that's passed along

between layers in a transformer. A positional encoding is simply a function that adds information about the relative or absolute position of a word in a sequence to the input embeddings. The encodings have the same dimension, d_{model} , as the input embeddings so they can be summed with the embedding vectors. The paper discusses learned and fixed encodings and proposes a sinusoidal function of sin and cosine with different frequencies, defined as:

Equation 9.3 Positional encoding function

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

This mapping function was chosen because for any offset k , $PE_{(pos+k)}$ can be represented as a linear function of PE_{pos} . In short, the model should be able to learn to attend to relative positions easily.

Let's look at how this can be coded in Pytorch. The official Pytorch Sequence-to-Sequence Modeling with `nn.Transformer` tutorial [\[347\]](#) provides an implementation of a `PositionEncoding` `nn.Module` based on the previous function:

Listing 9.1. Pytorch PositionalEncoding

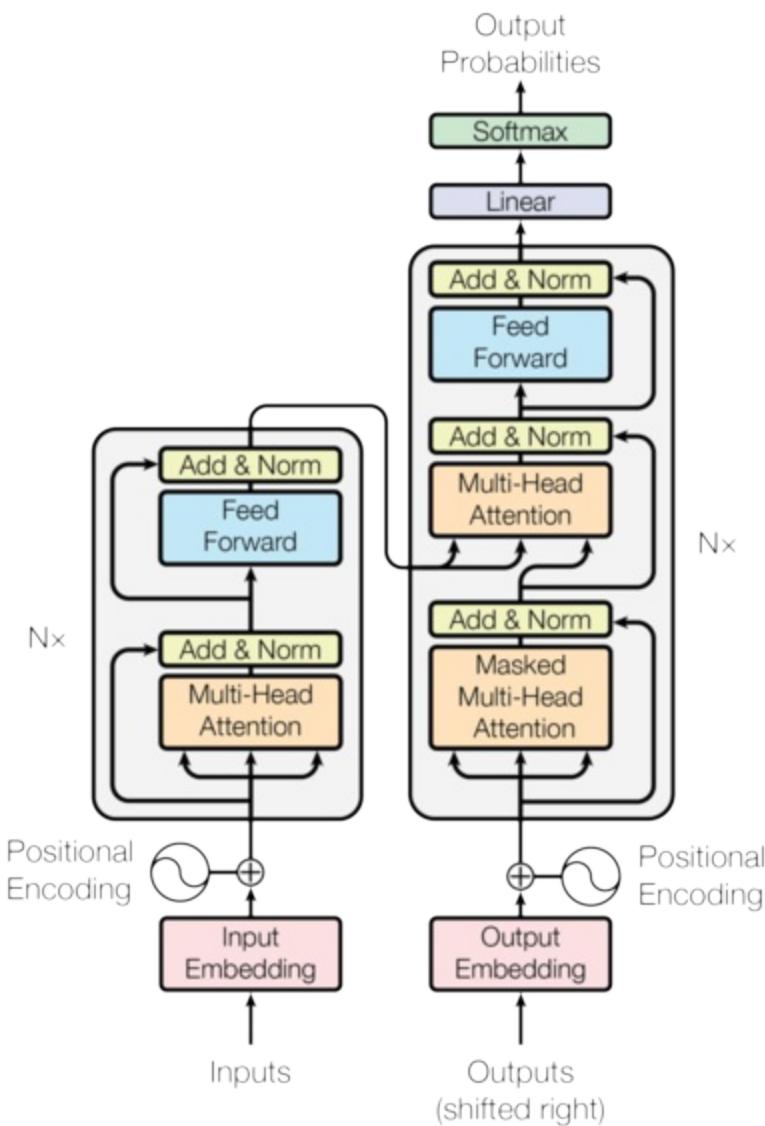
```
...     pe[:, 0::2] = torch.sin(position * div_term) #5
...     pe[:, 1::2] = torch.cos(position * div_term)
...     pe = pe.unsqueeze(0).transpose(0, 1)
...     self.register_buffer('pe', pe)
...
...     def forward(self, x):
...         x = x + self.pe[:x.size(0), :]
...         return self.dropout(x)
```

You will use this module in the translation transformer you build. However, first, we need to fill in the remaining details of the model to complete your understanding of the architecture.

9.4.2 Connecting all the pieces

Now that you've seen the hows and whys of BPE, embeddings, positional encoding, and multi-head self-attention, you understand all the elements of a transformer layer. You just need a lower dimensional linear layer at the output to collect all those attention weights together to create the output sequence of embeddings. And the linear layer output needs to be scaled (normalized) so that the layers all have the same scale. These linear and normalization layers are stacked on top of the attention layers to create reusable stackable transformer blocks as shown in figure [Figure 9.4](#).

Figure 9.4. Transformer architecture



In the original transformer, both the encoder and decoder are comprised of $N = 6$ stacked identical encoder and decoder layers, respectively.

Encoder

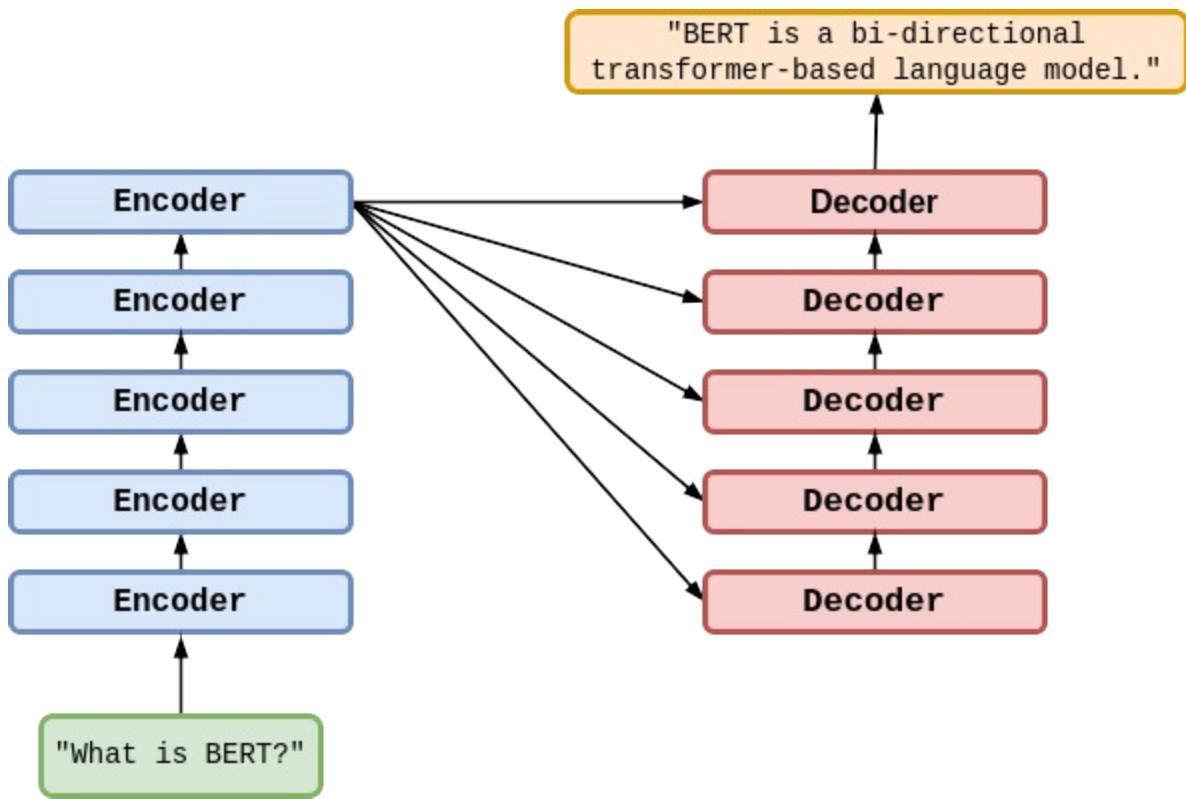
The encoder is composed of multiple encoder layers. Each encoder layer has two sub-layers: a multi-head attention layer and a position-wise fully connected feedforward network. A residual connection is made around each sub-layer. And each encoder layer has its output normalized so that all the values of the encodings passed between layers range between zero and one. The outputs of all sub-layers in a transformer layer (PyTorch module) that are

passed between layers all have dimension d_{model} . And the input embedding sequences to the encoder are summed with the positional encodings before being input into the encoder.

Decoder

The decoder is nearly identical to the encoder in the model but has three sublayers instead of one. The new sublayer is a fully connected layer similar to the multi-head self-attention matrix but contains only zeros and ones. This creates a *masking* of the output sequences that are to the right of the current target token (in a left-to-right language like English). This ensures that predictions for position i can depend only on previous outputs, for positions less than i . In other words, during training, the attention matrix is not allowed to "peek ahead" at the subsequent tokens that it is supposed to be generating in order to minimize the loss function. This prevents *leakage* or "cheating" during training, forcing the transformer to attend only to the tokens it has already seen or generated. Masks are not required within the decoders for an RNN, because each token is only revealed to the network one at a time. But transformer attention matrices have access to the entire sequence all at once during training.

Figure 9.5. Connections between encoder and decoder layers



9.4.3 Transformer Language Translation Example

Transformers are suited for many tasks. The "Attention Is All You Need" paper showed off a transformer that achieved better translation accuracy than any preceding approach. Using `torchtext`, you will prepare the Multi30k dataset for training a Transformer for German-English translation using the `torch.nn.Transformer` module. In this section, you will customize the decoder half of the `Transformer` class to output the self-attention weights for each sublayer. You use the matrix of self-attention weights to explain how the words in the input German text were combined together to create the embeddings used to produce the English text in the output. After training the model you will use it for inference on a test set to see for yourself how well it translates German text into English.

Preparing the Data

You can use the Hugging Face datasets package to simplify bookkeeping and ensure your text is fed into the Transformer in a predictable format

compatible with PyTorch. This is one of the trickiest parts of any deep learning project, ensuring that the structure and API for your dataset matches what your PyTorch training loop expects. Translation datasets are particularly tricky unless you use Hugging Face:

Listing 9.2. Load a translation dataset in Hugging Face format

```
>>> from datasets import load_dataset #1
>>> opus = load_dataset('opus_books', 'de-en')
>>> opus
DatasetDict({
    train: Dataset({
        features: ['id', 'translation'],
        num_rows: 51467
    })
})
```

Not all Hugging Face datasets have predefined test and validation splits of the data. But you can always create your own splits using the `train_test_split` method as in listing [Listing 9.3](#).

Listing 9.3. Load a translation dataset in Hugging Face format

```
>>> sents = opus['train'].train_test_split(test_size=.1)
>>> sents
DatasetDict({
    train: Dataset({
        features: ['id', 'translation'],
        num_rows: 48893
    })
    test: Dataset({
        features: ['id', 'translation'],
        num_rows: 2574
    })
})
```

It's always a good idea to examine some examples in your dataset before you start a long training run. This can help you make sure the data is what you expect. The `opus_books` doesn't contain many books. So it's not a very diverse (representative) sample of German. It has been segmented into only 50,000 aligned sentence pairs. Imagine having to learn German by having only a few translated books to read.

```
>>> next(iter(sents['test'])) #1
{'id': '9206',
 'translation': {'de': 'Es war wenigstens zu viel in der Luft.',
 'en': 'There was certainly too much of it in the air.'}}
```

If you would like to use a custom dataset of your own creation, it's always a good idea to comply with an open standard like the Hugging Face datasets package shown in listing [Listing 9.2](#) gives you a "best practice" approach to structuring your datasets. Notice that a translation dataset in Hugging Face contains an array of paired sentences with the language code in a dictionary. The dict keys of a translation example are the two-letter language code (from ISO 639-2)[\[348\]](#). The dict values of an example text are the sentences in each of the two languages in the dataset.



Tip

You'll avoid insidious, sometimes undetectable bugs if you resist the urge to invent your own data structure and instead use widely recognized open standards.

If you have access to a GPU, you probably want to use it for training transformers. Transformers are made for GPUs with their matrix multiplication operations for all the most computationally intensive parts of the algorithm. CPUs are adequate for most pre-trained Transformer models (except LLMs), but GPUs can save you a lot of time for training or fine-tuning a transformer. For example, GPT2 required 3 days to train with a relatively small (40 MB) training dataset on a 16-core CPU. It trained in 2 hours for the same dataset on a 2560-core GPU (40x speedup, 160x more cores). Listing [Listing 9.4](#) will enable your GPU if one is available.

Listing 9.4. Enable any available GPU

```
>>> DEVICE = torch.device(
...     'cuda' if torch.cuda.is_available()
...     else 'cpu')
```

To keep things simple you can tokenize your source and target language texts separately with specialized tokenizers for each. If you use the Hugging Face

tokenizers they will keep track of all of the special tokens that you'll need for a transformer to work on almost any machine learning task:

start-of-sequence token::typically "<SOS>" or "<s>" **end-of-sequence token**::typically "<EOS>" or "</s>" **out-of-vocabulary (unknown) token**::typically "<OOV>", "<unk>" **mask token**::typically "<mask>" **padding token**::typically "<pad>"

The start-of-sequence token is used to trigger the decoder to generate a token that is suitable for the first token in a sequence. And many generative problems will require you to have an end-of-sequence token, so that the decoder knows when it can stop recursively generating more tokens. Some datasets use the same token for both the *start-of-sequence* and the *end-of-sequence* marker. They do not need to be unique because your decoder will always "know" when it is starting a new generation loop. The padding token is used to fill in the sequence at the end for examples shorter than the maximum sequence length. The mask token is used to intentionally hide a known token for training task-independent encoders such as BERT. This is similar to what you did in Chapter 6 for training word embeddings using skip grams.

You can choose any tokens for these marker (special) tokens, but you want to make sure that they are not words used within the vocabulary of your dataset. So if you are writing a book about natural language processing and you don't want your tokenizer to trip up on the example SOS and EOS tokens, you may need to get a little more creative to generate tokens not found in your text.

Create a separate Hugging Face tokenizer for each language to speed up your tokenization and training and avoid having tokens leak from your source language text examples into your generated target language texts. You can use any language pair you like, but the original AIAYN paper demo examples usually translate from English (source) to German (target).

```
>>> SRC = 'en' #1
>>> TGT = 'de' #2
>>> SOS, EOS = '<s>', '</s>'
>>> PAD, UNK, MASK = '<pad>', '<unk>', '<mask>'
>>> SPECIAL_TOKS = [SOS, PAD, EOS, UNK, MASK]
>>> VOCAB_SIZE = 10_000
```

```
...
>>> from tokenizers import ByteLevelBPETokenizer #3
>>> tokenize_src = ByteLevelBPETokenizer()
>>> tokenize_src.train_from_iterator(
...     [x[SRC] for x in sents['train']['translation']],
...     vocab_size=10000, min_frequency=2,
...     special_tokens=SPECIAL_TOKS)
>>> PAD_IDX = tokenize_src.token_to_id(PAD)
...
>>> tokenize_tgt = ByteLevelBPETokenizer()
>>> tokenize_tgt.train_from_iterator(
...     [x[TGT] for x in sents['train']['translation']],
...     vocab_size=10000, min_frequency=2,
...     special_tokens=SPECIAL_TOKS)
>>> assert PAD_IDX == tokenize_tgt.token_to_id(PAD)
```

The ByteLevel part of your BPE tokenizer ensures that your tokenizer will never miss a beat (or byte) as it is tokenizing your text. A byte-level BPE tokenizer can always construct any character by combining one of the 256 possible single-byte tokens available in its vocabulary. This means it can process any language that uses the Unicode character set. A byte-level tokenizer will just fall back to representing the individual bytes of a Unicode character if it hasn't seen it before or hasn't included it in its token vocabulary. A byte-level tokenizer will need an average of 70% more tokens (almost double the vocabulary size) to represent a new text containing characters or tokens that it hasn't been trained on.

Character-level BPE tokenizers have their disadvantages too. A character-level tokenizer must hold each one of the multibyte Unicode characters in its vocabulary to avoid having any meaningless OOV (out-of-vocabulary) tokens. This can create a huge vocabulary for a multilingual transformer expected to handle most of the 161 languages covered by Unicode characters. There are 149,186 characters with Unicode code points for both historical (Egyptian hieroglyphs for example) and modern written languages. That's about 10 times the memory to store all the embeddings and tokens in your transformer's tokenizer. In the real world, it is usually practical to ignore historical languages and some rare modern languages when optimizing your transformer BPE tokenizer for memory and balancing that with your transformer's accuracy for your problem.



Important

The BPE tokenizer is one of the five key "superpowers" of transformers that makes them so effective. And a ByteLevel BPE tokenizer isn't quite as effective at representing the meaning of words even though it will never have OOV tokens. So in a production application, you may want to train your pipeline on both a character-level BPE tokenizer as well as a byte-level tokenizer. That way you can compare the results and choose the approach that gives you the best performance (accuracy and speed) for your application.

You can use your English tokenizer to build a preprocessing function that *flattens* the Dataset structure and returns a list of lists of token IDs (without padding).

```
def preprocess(examples):
    src = [x[source_lang] for x in examples["translation"]]
    src_toks = [tokenize_src(x) for x in src]
    # tgt = [x[target_lang] for x in examples["translation"]]
    # tgt_toks = [tokenize_tgt(x) for x in tgt]
    return src_toks
```

TranslationTransformer Model

At this point, you have tokenized the sentences in the Multi30k data and converted them to tensors consisting of indexes into the vocabularies for the source and target languages, German and English, respectively. The dataset has been split into separate training, validation and test sets, which you have wrapped with iterators for batch training. Now that the data is prepared you turn your focus to setting up the model. Pytorch provides an implementation of the model presented in "Attention Is All You Need", `torch.nn.Transformer`. You will notice the constructor takes several parameters, familiar amongst them are `d_model=512`, `nhead=8`, `num_encoder_layers=6`, and `num_decoder_layers=6`. The default values are set to the parameters employed in the paper. Along with several other parameters for the feedforward dimension, dropout, and activation, the model also provides support for a `custom_encoder` and `custom_decoder`. To make things interesting, create a custom decoder that additionally outputs a list of

attention weights from the multi-head self-attention layer in each sublayer of the decoder. It might sound complicated, but it's actually fairly straightforward if you simply subclass `torch.nn.TransformerDecoderLayer` and `torch.nn.TransformerDecoder` and augment the `forward()` methods to return the auxiliary outputs - the attention weights.

Listing 9.5. Extend `torch.nn.TransformerDecoderLayer` to additionally return multi-head self-attention weights

```
>>> from torch import Tensor
>>> from typing import Optional, Any

>>> class CustomDecoderLayer(nn.TransformerDecoderLayer):
...     def forward(self, tgt: Tensor, memory: Tensor,
...                tgt_mask: Optional[Tensor] = None,
...                memory_mask: Optional[Tensor] = None,
...                tgt_key_padding_mask: Optional[Tensor] = None
...                ) -> Tensor:
...         """Like decode but returns multi-head attention weigh
...         tgt2 = self.self_attn(
...             tgt, tgt, tgt, attn_mask=tgt_mask,
...             key_padding_mask=tgt_key_padding_mask)[0]
...         tgt = tgt + self.dropout1(tgt2)
...         tgt = self.norm1(tgt)
...         tgt2, attention_weights = self.multihead_attn(
...             tgt, memory, memory, #1
...             attn_mask=memory_mask,
...             key_padding_mask=mem_key_padding_mask,
...             need_weights=True)
...         tgt = tgt + self.dropout2(tgt2)
...         tgt = self.norm2(tgt)
...         tgt2 = self.linear2(
...             self.dropout(self.activation(self.linear1(tgt))))
...         tgt = tgt + self.dropout3(tgt2)
...         tgt = self.norm3(tgt)
...         return tgt, attention_weights #2
```

Listing 9.6. Extend `torch.nn.TransformerDecoder` to additionally return list of multi-head self-attention weights

```
>>> class CustomDecoder(nn.TransformerDecoder):
...     def __init__(self, decoder_layer, num_layers, norm=None):
...         super().__init__(
...             decoder_layer, num_layers, norm)
```

```

...     def forward(self,
...                 tgt: Tensor, memory: Tensor,
...                 tgt_mask: Optional[Tensor] = None,
...                 memory_mask: Optional[Tensor] = None,
...                 tgt_key_padding_mask: Optional[Tensor] = None
...                 ) -> Tensor:
...             """Like TransformerDecoder but cache multi-head attention
...             self.attention_weights = [] #1
...             output = tgt
...             for mod in self.layers:
...                 output, attention = mod(
...                     output, memory, tgt_mask=tgt_mask,
...                     memory_mask=memory_mask,
...                     tgt_key_padding_mask=tgt_key_padding_mask)
...                 self.attention_weights.append(attention) #2
...
...             if self.norm is not None:
...                 output = self.norm(output)
...
...             return output

```

The only change to `.forward()` from the parent's version is to cache weights in the list member variable, `attention_weights`.

To recap, you have extended the `torch.nn.TransformerDecoder` and its sublayer component, `torch.nn.TransformerDecoderLayer`, mainly for exploratory purposes. That is, you save the multi-head self-attention weights from the different decoder layers in the Transformer model you are about to configure and train. The `forward()` methods in each of these classes copy the one in the parent nearly verbatim, with the exception of the changes called out to save the attention weights.

The `torch.nn.Transformer` is a somewhat bare-bones version of the sequence-to-sequence model containing the main secret sauce, the multi-head self-attention in both the encoder and decoder. If one looks at the source code for the module [\[349\]](#), the model does not assume the use of embedding layers or positional encodings. Now you will create your *TranslationTransformer* model that uses the custom decoder components, by extending `torch.nn.Transformer` module. Begin with defining the constructor, which takes parameters `src_vocab_size` for a source embedding size, and `tgt_vocab_size` for the target, and uses them to initialize a basic `torch.nn.Embedding` for each. Notice a `PositionalEncoding` member,

`pos_enc`, is created in the constructor for adding the word location information.

Listing 9.7. Extend nn.Transformer for translation with a CustomDecoder

```
>>> from einops import rearrange #1
...
>>> class TranslationTransformer(nn.Transformer): #2
...     def __init__(self,
...                  device=DEVICE,
...                  src_vocab_size: int = VOCAB_SIZE,
...                  src_pad_idx: int = PAD_IDX,
...                  tgt_vocab_size: int = VOCAB_SIZE,
...                  tgt_pad_idx: int = PAD_IDX,
...                  max_sequence_length: int = 100,
...                  d_model: int = 512,
...                  nhead: int = 8,
...                  num_encoder_layers: int = 6,
...                  num_decoder_layers: int = 6,
...                  dim_feedforward: int = 2048,
...                  dropout: float = 0.1,
...                  activation: str = "relu"
...                  ):
...
...         decoder_layer = CustomDecoderLayer(
...             d_model, nhead, dim_feedforward, #3
...             dropout, activation)
...         decoder_norm = nn.LayerNorm(d_model)
...         decoder = CustomDecoder(
...             decoder_layer, num_decoder_layers,
...             decoder_norm) #4
...
...         super().__init__(
...             d_model=d_model, nhead=nhead,
...             num_encoder_layers=num_encoder_layers,
...             num_decoder_layers=num_decoder_layers,
...             dim_feedforward=dim_feedforward,
...             dropout=dropout, custom_decoder=decoder)
...
...         self.src_pad_idx = src_pad_idx
...         self.tgt_pad_idx = tgt_pad_idx
...         self.device = device
...
...         self.src_emb = nn.Embedding(
...             src_vocab_size, d_model) #5
...         self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
```

```
...         self.pos_enc = PositionalEncoding(
...             d_model, dropout, max_sequence_length) #6
...         self.linear = nn.Linear(
...             d_model, tgt_vocab_size) #7
```

Note the import of `rearrange` from the `einops` [\[350\]](#) package. Mathematicians like it for tensor reshaping and shuffling because it uses a syntax common in graduate level applied math courses. To see why you need to `rearrange()` your tensors refer to the `torch.nn.Transformer` documentation [\[351\]](#). If you get any one of the dimensions of any of the tensors wrong it will mess up the entire pipeline, sometimes invisibly.

Listing 9.8. `torch.nn.Transformer` "shape" and dimension descriptions

S: source sequence length
T: target sequence length
N: batch size
E: embedding dimension number (the feature number)

src: (S, N, E)

tgt: (T, N, E)
src_mask: (S, S)
tgt_mask: (T, T)
memory_mask: (T, S)
src_key_padding_mask: (N, S)
tgt_key_padding_mask: (N, T)
memory_key_padding_mask: (N, S)

output: (T, N, E)

The datasets you created using `torchtext` are batch-first. So, borrowing the nomenclature in the Transformer documentation, your source and target tensors have shape (N, S) and (N, T), respectively. To feed them to the `torch.nn.Transformer` (i.e. call its `forward()` method), the source and target must be reshaped. Also, you want to apply the embeddings plus the positional encoding to the source and target sequences. Additionally, a padding key mask is needed for each and a memory key mask is required for the target. Note, you can manage the embeddings and positional encodings outside the class, in the training and inference sections of the pipeline. However, since the model is specifically set up for translation, you make a

stylistic/design choice to encapsulate the source and target sequence preparation within the class. To this end, you define `prepare_src()` and `prepare_tgt()` methods for preparing the sequences and generating the required masks.

Listing 9.9. TranslationTransformer `prepare_src()`

```
>>>     def _make_key_padding_mask(self, t, pad_idx):
...         mask = (t == pad_idx).to(self.device)
...         return mask
...
...     def prepare_src(self, src, src_pad_idx):
...         src_key_padding_mask = self._make_key_padding_mask(
...             src, src_pad_idx)
...         src = rearrange(src, 'N S -> S N')
...         src = self.pos_enc(self.src_emb(src)
...                            * math.sqrt(self.d_model))
...         return src, src_key_padding_mask
```

The `make_key_padding_mask()` method returns a tensor set to 1's in the position of the padding token in the given tensor, and zero otherwise. The `prepare_src()` method generates the padding mask and then rearranges the `src` to the shape that the model expects. It then applies the positional encoding to the source embedding multiplied by the square root of the model's dimension. This is taken directly from "Attention Is All You Need". The method returns the `src` with positional encoding applied, and the key padding mask for it.

The `prepare_tgt()` method used for the target sequence is nearly identical to `prepare_src()`. It returns the `tgt` adjusted for positional encodings, and a target key padding mask. However, it also returns a "subsequent" mask, `tgt_mask`, which is a triangular matrix for which columns (ones) in a row that are permitted to be observed. To generate the subsequent mask you use `Transformer.generate_square_subsequent_mask()` method defined in the base class as shown in the following listing.

Listing 9.10. TranslationTransformer `prepare_tgt()`

```
>>>     def prepare_tgt(self, tgt, tgt_pad_idx):
...         tgt_key_padding_mask = self._make_key_padding_mask(
...             tgt, tgt_pad_idx)
```

```

...
    tgt = rearrange(tgt, 'N T -> T N')
    tgt_mask = self.generate_square_subsequent_mask(
        tgt.shape[0]).to(self.device)
    tgt = self.pos_enc(self.tgt_emb(tgt)
        * math.sqrt(self.d_model))
...
    return tgt, tgt_key_padding_mask, tgt_mask

```

You put `prepare_src()` and `prepare_tgt()` to use in the model's `forward()` method. After preparing the inputs, it simply invokes the parent's `forward()` and feeds the outputs through a Linear reduction layer after transforming from (T, N, E) back to batch first (N, T, E) . We do this for consistency in our training and inference.

Listing 9.11. TranslationTransformer forward()

```

>>>     def forward(self, src, tgt):
...         src, src_key_padding_mask = self.prepare_src(
...             src, self.src_pad_idx)
...         tgt, tgt_key_padding_mask, tgt_mask = self.prepare_tg(
...             tgt, self.tgt_pad_idx)
...         memory_key_padding_mask = src_key_padding_mask.clone()
...         output = super().forward(
...             src, tgt, tgt_mask=tgt_mask,
...             src_key_padding_mask=src_key_padding_mask,
...             tgt_key_padding_mask=tgt_key_padding_mask,
...             memory_key_padding_mask=memory_key_padding_mask)
...         output = rearrange(output, 'T N E -> N T E')
...         return self.linear(output)

```

Also, define an `init_weights()` method that can be called to initialize the weights of all submodules of the Transformer. Xavier initialization is commonly used for Transformers, so use it here. The Pytorch `nn.Module` documentation [\[352\]](#) describes the `apply(fn)` method that recursively applies `fn` to every submodule of the caller.

Listing 9.12. TranslationTransformer init_weights()

```

>>>     def init_weights(self):
...         def _init_weights(m):
...             if hasattr(m, 'weight') and m.weight.dim() > 1:
...                 nn.init.xavier_uniform_(m.weight.data)
...         self.apply(_init_weights); #1

```

The individual components of the model have been defined and the complete model is shown in the next listing.

Listing 9.13. TranslationTransformer complete model definition

```
>>> class TranslationTransformer(nn.Transformer):
...     def __init__(self,
...                  device=DEVICE,
...                  src_vocab_size: int = 10000,
...                  src_pad_idx: int = PAD_IDX,
...                  tgt_vocab_size: int = 10000,
...                  tgt_pad_idx: int = PAD_IDX,
...                  max_sequence_length: int = 100,
...                  d_model: int = 512,
...                  nhead: int = 8,
...                  num_encoder_layers: int = 6,
...                  num_decoder_layers: int = 6,
...                  dim_feedforward: int = 2048,
...                  dropout: float = 0.1,
...                  activation: str = "relu"
...                 ):
...         decoder_layer = CustomDecoderLayer(
...             d_model, nhead, dim_feedforward,
...             dropout, activation)
...         decoder_norm = nn.LayerNorm(d_model)
...         decoder = CustomDecoder(
...             decoder_layer, num_decoder_layers, decoder_norm)
...
...         super().__init__(
...             d_model=d_model, nhead=nhead,
...             num_encoder_layers=num_encoder_layers,
...             num_decoder_layers=num_decoder_layers,
...             dim_feedforward=dim_feedforward,
...             dropout=dropout, custom_decoder=decoder)
...
...         self.src_pad_idx = src_pad_idx
...         self.tgt_pad_idx = tgt_pad_idx
...         self.device = device
...         self.src_emb = nn.Embedding(src_vocab_size, d_model)
...         self.tgt_emb = nn.Embedding(tgt_vocab_size, d_model)
...         self.pos_enc = PositionalEncoding(
...             d_model, dropout, max_sequence_length)
...         self.linear = nn.Linear(d_model, tgt_vocab_size)
...
...     def init_weights(self):
...         def _init_weights(m):
```

```

...
    if hasattr(m, 'weight') and m.weight.dim() > 1:
        nn.init.xavier_uniform_(m.weight.data)
self.apply(_init_weights);

...
def _make_key_padding_mask(self, t, pad_idx=PAD_IDX):
    mask = (t == pad_idx).to(self.device)
    return mask

...
def prepare_src(self, src, src_pad_idx):
    src_key_padding_mask = self._make_key_padding_mask(
        src, src_pad_idx)
    src = rearrange(src, 'N S -> S N')
    src = self.pos_enc(self.src_emb(src)
        * math.sqrt(self.d_model))
    return src, src_key_padding_mask

...
def prepare_tgt(self, tgt, tgt_pad_idx):
    tgt_key_padding_mask = self._make_key_padding_mask(
        tgt, tgt_pad_idx)
    tgt = rearrange(tgt, 'N T -> T N')
    tgt_mask = self.generate_square_subsequent_mask(
        tgt.shape[0]).to(self.device)      #1
    tgt = self.pos_enc(self.tgt_emb(tgt)
        * math.sqrt(self.d_model))
    return tgt, tgt_key_padding_mask, tgt_mask

...
def forward(self, src, tgt):
    src, src_key_padding_mask = self.prepare_src(
        src, self.src_pad_idx)
    tgt, tgt_key_padding_mask, tgt_mask = self.prepare_tg(
        tgt, self.tgt_pad_idx)
    memory_key_padding_mask = src_key_padding_mask.clone()
    output = super().forward(
        src, tgt, tgt_mask=tgt_mask,
        src_key_padding_mask=src_key_padding_mask,
        tgt_key_padding_mask=tgt_key_padding_mask,
        memory_key_padding_mask = memory_key_padding_mask
    )
    output = rearrange(output, 'T N E -> N T E')
    return self.linear(output)

```

Finally, you have a complete transformer all your own! And you should be able to use it for translating between virtually any pair of languages, even character-rich languages such as traditional Chinese and Japanese. And you have explicit access to all the hyperparameters that you might need to tune your model for your problem. For example, you can increase the vocabulary

size for the target or source languages to efficiently handle *character-rich* languages such as traditional Chinese and Japanese.



Note

Traditional Chinese and Japanese (kanji) are called *character-rich* because they have a much larger number of unique characters than European languages. Chinese and Japanese languages use logograph characters. Logograph characters look a bit like small pictographs or abstract hieroglyphic drawings. For example, the kanji character "日" can mean day and it looks a little like the day block you might see on a calendar. Japanese logographic characters are roughly equivalent to word pieces somewhere between morphemes and words in the English language. This means that you will have many more unique characters in logographic languages than in European languages. For instance, traditional Japanese uses about 3500 unique kanji characters.^[353] English has roughly 7000 unique syllables within the most common 20,000 words.

You can even change the number of layers in the encoder and decoder sides of the transformer, depending on the source (encoder) or target (decoder) language. You can even create a translation transformer that simplifies text for explaining complex concepts to five-year-olds, or adults on Mastodon server focused on ELI5 ("explain it like I'm 5") conversations. If you reduce the number of layers in the decoder this will create a "capacity" bottleneck that can force your decoder to simplify or compress the concepts coming out of the encoder. Similarly, the number of attention heads in the encoder or decoder layers can be adjusted to increase or decrease the capacity (complexity) of your transformer.

Training the TranslationTransformer

Now let's create an instance of the model for our translation task and initialize the weights in preparation for training. For the model's dimensions you use the defaults, which correlate to the sizes of the original "Attention Is All You Need" transformer. Know that since the encoder and decoder building blocks comprise duplicate, stackable layers, you can configure the

model with any number of these layers.

Listing 9.14. Instantiate a TranslationTransformer

```
>>> model = TranslationTransformer(  
...     device=DEVICE,  
...     src_vocab_size=tokenize_src.get_vocab_size(),  
...     src_pad_idx=tokenize_src.token_to_id('<pad>'),  
...     tgt_vocab_size=tokenize_tgt.get_vocab_size(),  
...     tgt_pad_idx=tokenize_tgt.token_to_id('<pad>')  
...     ).to(DEVICE)  
>>> model.init_weights()  
>>> model #1
```

PyTorch creates a nice `__str__` representation of your model. It displays all the layers and their inner structure including the shapes of the inputs and outputs. You may even be able to see the parallels between the layers of your models and the diagrams of transformers that you see in this chapter or online. From the first half of the text representation for your transformer, you can see that all of the encoder layers have exactly the same structure. The inputs and outputs of each `TransformerEncoderLayer` have the same shape, so this ensures that you can stack them without reshaping linear layers between them. Transformer layers are like the floors of a skyscraper or a child's stack of wooden blocks. Each level has exactly the same 3D shape.

```
TranslationTransformer(  
    encoder): TransformerEncoder(  
        (layers): ModuleList(  
            (0-5): 6 x TransformerEncoderLayer(  
                (self_attn): MultiheadAttention(  
                    (out_proj): NonDynamicallyQuantizableLinear(  
                        in_features=512, out_features=512, bias=True)  
                )  
                (linear1): Linear(  
                    in_features=512, out_features=2048, bias=True)  
                (dropout): Dropout(p=0.1, inplace=False)  
                (linear2): Linear(  
                    in_features=2048, out_features=512, bias=True)  
                (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=  
                (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=  
                (dropout1): Dropout(p=0.1, inplace=False)  
                (dropout2): Dropout(p=0.1, inplace=False)  
            )
```

```
)  
    (norm): LayerNorm((512,), eps=1e-05, elementwise_affine=True)  
)  
...  
)
```

Notice that you set the sizes of your source and target vocabularies in the constructor. Also, you pass the indices for the source and target padding tokens for the model to use in preparing the source, targets, and associated masking sequences. Now that you have the model defined, take a moment to do a quick sanity check to make sure there are no obvious coding errors before you set up the training and inference pipeline. You can create "batches" of random integer tensors for the sources and targets and pass them to the model, as demonstrated in the following listing.

Listing 9.15. Quick model sanity check with random tensors

```
>>> src = torch.randint(1, 100, (10, 5)).to(DEVICE) #1  
>>> tgt = torch.randint(1, 100, (10, 7)).to(DEVICE)  
...  
>>> with torch.no_grad():  
...     output = model(src, tgt) #2  
...  
>>> print(output.shape)  
torch.Size([10, 7, 5893])
```

We created two tensors, `src` and `tgt`, each with random integers between 1 and 100 distributed uniformly. Your model accepts tensors having batch-first shape, so we made sure that the batch sizes (10 in this case) were identical - otherwise we would have received a runtime error on the forward pass, that looks like this:

```
RuntimeError: the batch number of src and tgt must be equal
```

It may be obvious, the source and target sequence lengths do not have to match, which is confirmed by the successful call to `model(src, tgt)`.



Tip

When setting up a new sequence-to-sequence model for training, you may want to initially use smaller tunables in your setup. This includes limiting

max sequence lengths, reducing batch sizes, and specifying a smaller number of training loops or epochs. This will make it easier to debug issues in your model and/or pipeline to get your program executing end-to-end more quickly. Be careful not to draw any conclusions on the capabilities/accuracy of your model at this "bootstrapping" stage; the goal is simply to get the pipeline to run.

Now that you feel confident the model is ready for action, the next step is to define the optimizer and criterion for training. "Attention Is All You Need" used Adam optimizer with a warmup period in which the learning rate is increased followed by a decreasing rate for the duration of training. You will use a static rate, 1e-4, which is smaller than the default rate 1e-2 for Adam. This should provide for stable training as long as you are patient to run enough epochs. You can play with learning rate scheduling as an exercise if you are interested. Other Transformer based models you will look at later in this chapter use a static learning rate. As is common for this type of task, you use `torch.nn.CrossEntropyLoss` for the criterion.

Listing 9.16. Optimizer and Criterion

```
>>> LEARNING_RATE = 0.0001
>>> optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_
>>> criterion = nn.CrossEntropyLoss(ignore_index=TRG_PAD_IDX) #1
```

Ben Trevett contributed much of the code for the Pytorch Transformer Beginner tutorial. He, along with colleagues, has written an outstanding and informative Jupyter notebook series for their Pytorch Seq2Seq tutorial [\[354\]](#) covering sequence-to-sequence models. Their Attention Is All You Need [\[355\]](#) notebook provides a from-scratch implementation of a basic transformer model. To avoid re-inventing the wheel, the training and evaluation driver code in the next sections is borrowed from Ben's notebook, with minor changes.

The `train()` function implements a training loop similar to others you have seen. Remember to put the model into `train` mode before the batch iteration. Also, note that the last token in the target, which is the EOS token, is stripped from `trg` before passing it as input to the model. We want the model to predict the end of a string. The function returns the average loss per iteration.

Listing 9.17. Model training function

```
>>> def train(model, iterator, optimizer, criterion, clip):
...     model.train() #1
...     epoch_loss = 0
...
...     for i, batch in enumerate(iterator):
...         src = batch.src
...         trg = batch.trg
...         optimizer.zero_grad()
...         output = model(src, trg[:, :-1]) #2
...         output_dim = output.shape[-1]
...         output = output.contiguous().view(-1, output_dim)
...         trg = trg[:, 1:].contiguous().view(-1)
...         loss = criterion(output, trg)
...         loss.backward()
...         torch.nn.utils.clip_grad_norm_(model.parameters(), cl
...         optimizer.step()
...         epoch_loss += loss.item()
...
...     return epoch_loss / len(iterator)
```

The `evaluate()` function is similar to `train()`. You set the model to eval mode and use the `with torch.no_grad()` paradigm as usual for straight inference.

Listing 9.18. Model evaluation function

```
>>> def evaluate(model, iterator, criterion):
...     model.eval() #1
...     epoch_loss = 0
...
...     with torch.no_grad(): #2
...         for i, batch in enumerate(iterator):
...             src = batch.src
...             trg = batch.trg
...             output = model(src, trg[:, :-1])
...             output_dim = output.shape[-1]
...             output = output.contiguous().view(-1, output_dim)
...             trg = trg[:, 1:].contiguous().view(-1)
...             loss = criterion(output, trg)
...             epoch_loss += loss.item()
...
...     return epoch_loss / len(iterator)
```

Next a straightforward utility function `epoch_time()`, used for calculating the time elapsed during training, is defined as follows.

Listing 9.19. Utility function for elapsed time

```
>>> def epoch_time(start_time, end_time):
...     elapsed_time = end_time - start_time
...     elapsed_mins = int(elapsed_time / 60)
...     elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
...     return elapsed_mins, elapsed_secs
```

Now, let's proceed to setup the training. You set the number of epochs to 15, to give the model enough opportunities to train with the previously selected learning rate of `1e-4`. You can experiment with different combinations of learning rates and epoch numbers. In a future example, you will use an early stopping mechanism to avoid over-fitting and unnecessary training time. Here you declare a filename for `BEST_MODEL_FILE` and after each epoch, if the validation loss is an improvement over the previous best loss, the model is saved and the best loss is updated as shown.

Listing 9.20. Run the TranslationTransformer model training and save the best model to file

```
>>> N_EPOCHS = 15
>>> CLIP = 1
>>> BEST_MODEL_FILE = 'best_model.pytorch'
>>> best_valid_loss = float('inf')
>>> for epoch in range(N_EPOCHS):
...     start_time = time.time()
...     train_loss = train(
...         model, train_iterator, optimizer, criterion, CLIP)
...     valid_loss = evaluate(model, valid_iterator, criterion)
...     end_time = time.time()
...     epoch_mins, epoch_secs = epoch_time(start_time, end_time)
...
...     if valid_loss < best_valid_loss:
...         best_valid_loss = valid_loss
...         torch.save(model.state_dict(), BEST_MODEL_FILE)
...         print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_'
...             train_ppl = f'{math.exp(train_loss):.3f}'
...             print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {train_'
...                 valid_ppl = f'{math.exp(valid_loss):.3f}'
...                 print(f'\t Val. Loss: {valid_loss:.3f} | Val. PPL: {valid_
```

```

Epoch: 01 | Time: 0m 55s
    Train Loss: 4.835 | Train PPL: 125.848
    Val. Loss: 3.769 | Val. PPL: 43.332
Epoch: 02 | Time: 0m 56s
    Train Loss: 3.617 | Train PPL: 37.242
    Val. Loss: 3.214 | Val. PPL: 24.874
Epoch: 03 | Time: 0m 56s
    Train Loss: 3.197 | Train PPL: 24.448
    Val. Loss: 2.872 | Val. PPL: 17.679

...
Epoch: 13 | Time: 0m 57s
    Train Loss: 1.242 | Train PPL: 3.463
    Val. Loss: 1.570 | Val. PPL: 4.805
Epoch: 14 | Time: 0m 57s
    Train Loss: 1.164 | Train PPL: 3.204
    Val. Loss: 1.560 | Val. PPL: 4.759
Epoch: 15 | Time: 0m 57s
    Train Loss: 1.094 | Train PPL: 2.985
    Val. Loss: 1.545 | Val. PPL: 4.689

```

Notice that we could have probably run a few more epochs given that validation loss was still decreasing prior to exiting the loop. Let's see how the model performs on a test set by loading the *best* model and running the `evaluate()` function on the test set.

Listing 9.21. Load *best* model from file and perform evaluation on test data set

```

>>> model.load_state_dict(torch.load(BEST_MODEL_FILE))
>>> test_loss = evaluate(model, test_iterator, criterion)
>>> print(f'| Test Loss: {test_loss:.3f} | Test PPL: {math.exp(te
| Test Loss: 1.590 | Test PPL: 4.902 |

```

Your translation transformer achieves a log loss of about 1.6 on the test set. For a translation model trained on such a small dataset, this is not too bad. Log loss of 1.59 corresponds to a 20% probability ($\exp(-1.59)$) of generating the correct token and the exact position it was provided in the test set. Because there are many different correct English translations for a given German text, this is a reasonable accuracy for a model that can be trained on a commodity laptop.

TranslationTransformer Inference

You are now convinced your model is ready to become your personal German-to-English interpreter. Performing translation requires only slightly more work to set up, which you do in the `translate_sentence()` function in the next listing. In brief, start by tokenizing the source *sentence* if it has not been tokenized already and end-capping it with the `<sos>` and `<eos>` tokens. Next, you call the `prepare_src()` method of the model to transform the `src` sequence and generate the source key padding mask as was done in training and evaluation. Then run the prepared `src` and `src_key_padding_mask` through the model's encoder and save its output (in `enc_src`). Now, here is the fun part, where the target sentence (the translation) is generated. Start by initializing a list, `trg_indexes`, to the `SOS` token. In a loop - while the generated sequence has not reached a maximum length - convert the current prediction, `trg_indexes`, to a tensor. Use the model's `prepare_tgt()` method to prepare the target sequence, creating the target key padding mask, and the target sentence mask. Run the current decoder output, the encoder output, and the two masks through the decoder. Get the latest predicted token from the decoder output and append it to `trg_indexes`. Break out of the loop if the prediction was an `<eos>` token (or if maximum sentence length is reached). The function returns the target indexes converted to tokens (words) and the attention weights from the decoder in the model.

Listing 9.22. Define `translate_sentence()` for performing inference

```
>>> def translate_sentence(sentence, src_field, trg_field,
...     model, device=DEVICE, max_len=50):
...     model.eval()
...     if isinstance(sentence, str):
...         nlp = spacy.load('de')
...         tokens = [token.text.lower() for token in nlp(sentence)]
...     else:
...         tokens = [token.lower() for token in sentence]
...     tokens = ([src_field.init_token] + tokens
...             + [src_field.eos_token]) #1
...     src_indexes = [src_field.vocab.stoi[token] for token in tokens]
...     src = torch.LongTensor(src_indexes).unsqueeze(0).to(device)
...     src, src_key_padding_mask = model.prepare_src(src, SRC_PA
...     with torch.no_grad():
...         enc_src = model.encoder(src,
...                             src_key_padding_mask=src_key_padding_mask)
...     trg_indexes = [
...         trg_field.vocab.stoi[trg_field.init_token]] #2
```

```

...
    for i in range(max_len):
        tgt = torch.LongTensor(trg_indexes).unsqueeze(0).to(d
        tgt, tgt_key_padding_mask, tgt_mask = model.prepare_t
        tgt, TRG_PAD_IDX)
        with torch.no_grad():
            output = model.decoder(
                tgt, enc_src, tgt_mask=tgt_mask,
                tgt_key_padding_mask=tgt_key_padding_mask)
            output = rearrange(output, 'T N E -> N T E')
            output = model.linear(output)

        pred_token = output.argmax(2)[:, -1].item() #3
        trg_indexes.append(pred_token)

        if pred_token == trg_field.vocab.stoi[
            trg_field.eos_token]: #4
            break

    trg_tokens = [trg_field.vocab.itos[i] for i in trg_indexe
    translation = trg_tokens[1:]

    return translation, model.decoder.attention_weights

```

Your `translate_sentence()` wraps up your big transformer into a handy package you can use to translate whatever German sentence you run across.

TranslationTransformer Inference Example 1

Now you can use your `translate_sentence()` function on an example text. Since you probably do not know German, you can use a random example from the test data. Try it for the sentence "Eine Mutter und ihr kleiner Sohn genießen einen schönen Tag im Freien." In the OPUS dataset the character case was folded so that the text you feed into your transformer should be "eine mutter und ihr kleiner sohn genießen einen schönen tag im freien." And the correct translation that you're looking for is: "A mother and her little [or young] son are enjoying a beautiful day outdoors."

Listing 9.23. Load sample at `test_data` index 10

```

>>> example_idx = 10
>>> src = vars(test_data.examples[example_idx])['src']
>>> trg = vars(test_data.examples[example_idx])['trg']

```

```

>>> src
['eine', 'mutter', 'und', 'ihr', 'kleiner', 'sohn', 'genießen',
 'einen', 'schönen', 'tag', 'im', 'freien', '.']
>>> trg
['a', 'mother', 'and', 'her', 'young', 'song', 'enjoying',
 'a', 'beautiful', 'day', 'outside', '.']

```

It looks like the OPUS dataset is not perfect - the target (translated) token sequence is missing the verb "are" between "song" and "enjoying". And, the German word "kleiner" can be translated as little or young, but OPUS dataset example only provides one possible "correct" translation. And what about that "young song," that seems odd. Perhaps that's a typo in the OPUS test dataset.

Now you can run the src token sequence through your translator to see how it deals with that ambiguity.

Listing 9.24. Translate the test data sample

```

>>> translation, attention = translate_sentence(src, SRC, TRG, mo
>>> print(f'translation = {translation}')
translation = ['a', 'mother', 'and', 'her', 'little', 'son', 'enj

```

Interestingly, it appears there is a typo in the translation of the German word for "son" ("sohn") in the OPUS dataset. The dataset incorrectly translates "sohn" in German to "song" in English. Based on context, it appears the model did well to infer that a mother is (probably) with her young (little) "son". The model gives us the adjective "little" instead of "young", which is acceptable, given that the direct translation of the German word "kleiner" is "smaller".

Let's focus our attention on, um, *attention*. In your model, you defined a *CustomDecoder* that saves the average attention weights for each decoder layer on each forward pass. You have the *attention* weights from the translation. Now write a function to visualize self-attention for each decoder layer using `matplotlib`.

Listing 9.25. Function to visualize self-attention weights for decoder layers of the TranslationTransformer

```

>>> import matplotlib.pyplot as plt
>>> import matplotlib.ticker as ticker
...
>>> def display_attention(sentence, translation, attention_weights)
...     n_attention = len(attention_weights)
...
...     n_cols = 2
...     n_rows = n_attention // n_cols + n_attention % n_cols
...
...     fig = plt.figure(figsize=(15,25))
...
...     for i in range(n_attention):
...
...         attention = attention_weights[i].squeeze(0)
...         attention = attention.cpu().detach().numpy()
...         cax = ax.matshow(attention, cmap='gist_yarg')
...
...         ax = fig.add_subplot(n_rows, n_cols, i+1)
...         ax.tick_params(labelsize=12)
...         ax.set_xticklabels([''] + ['<sos>'] +
...                           [t.lower() for t in sentence]+['<eos>'],
...                           rotation=45)
...         ax.set_yticklabels(['']+translation)
...         ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
...         ax.yaxis.set_major_locator(ticker.MultipleLocator(1))
...
...     plt.show()
...     plt.close()

```

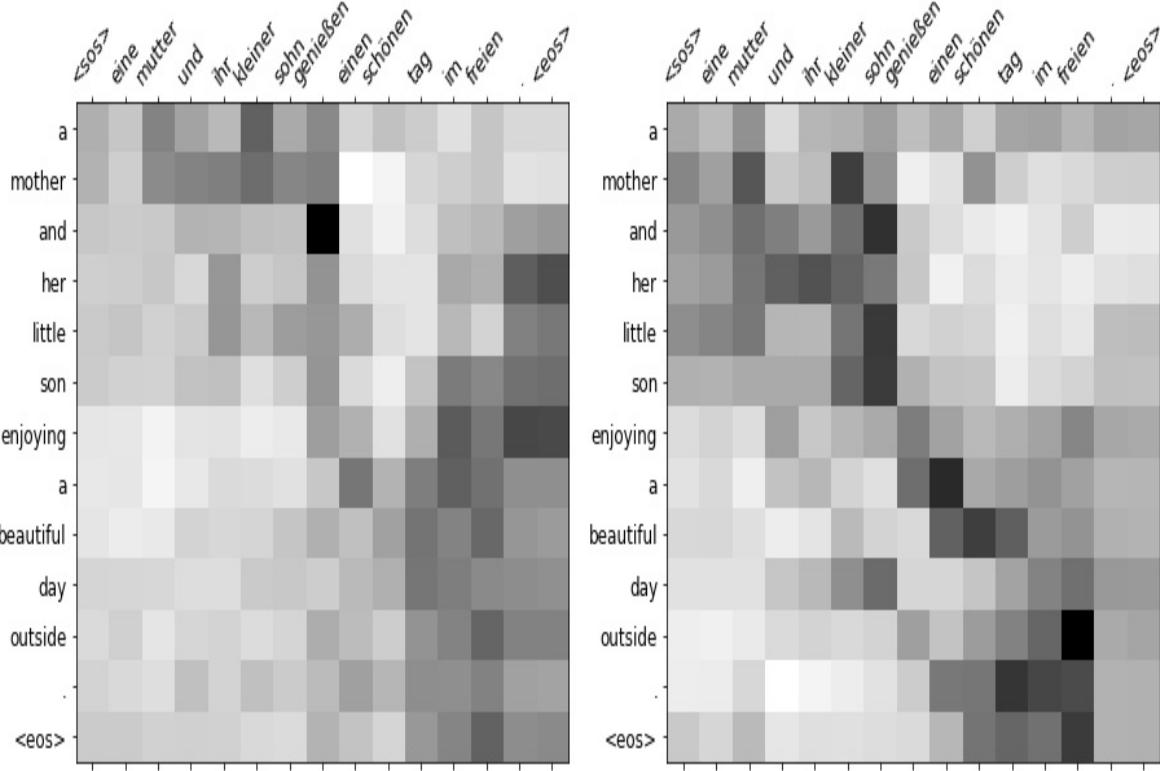
The function plots the attention values at each index in the sequence with the original sentence on the x-axis and the translation along the y-axis. We use the *gist_yarg* color map since it's a gray-scale scheme that is printer-friendly. Now you display the attention for the "mother and son enjoying the beautiful day" sentence.

Listing 9.26. Visualize the self-attention weights for the test example translation

```
>>> display_attention(src, translation, attention_weights)
```

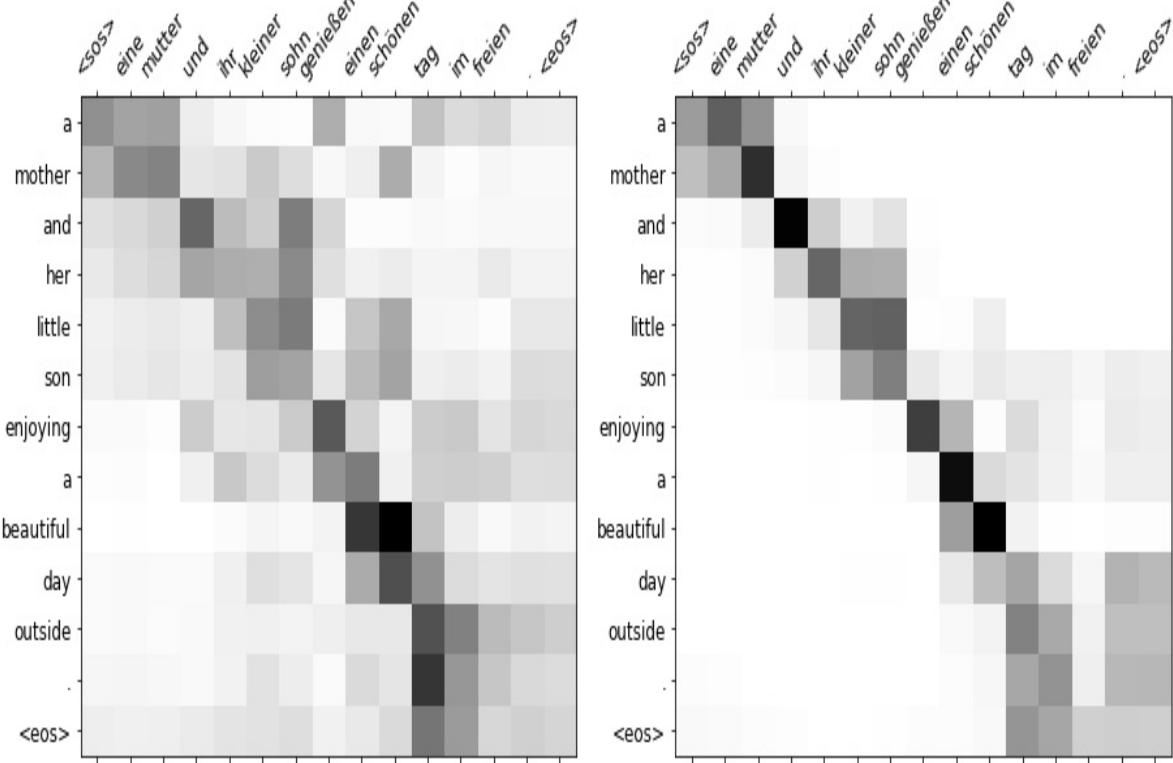
Looking at the plots for the initial two decoder layers we can see that an area of concentration is starting to develop along the diagonal.

Figure 9.6. Test Translation Example: Decoder Self-Attention Layers 1 and 2



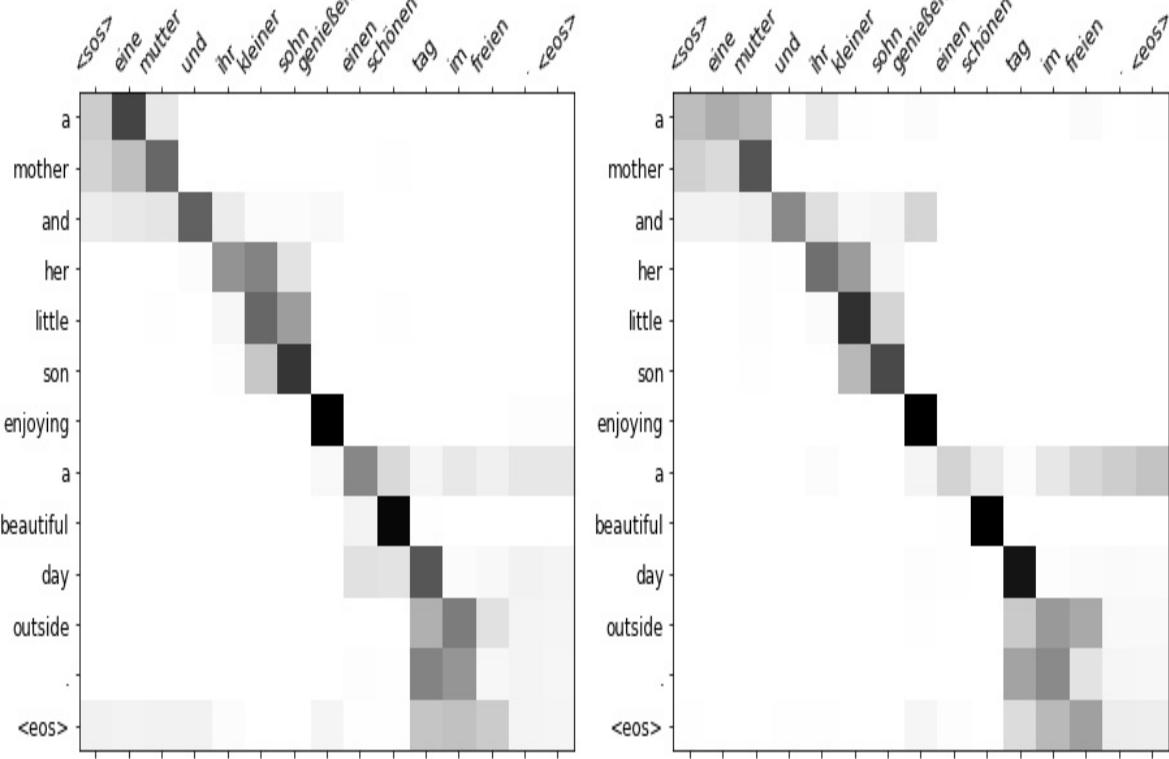
In the subsequent layers, three and four, the focus is appearing to become more refined.

Figure 9.7. Test Translation Example: Decoder Self-Attention Layers 3 and 4



In the final two layers, we see the attention is strongly weighted where direct word-to-word translation is done, along the diagonal, which is what you likely would expect. Notice the shaded clusters of article-noun and adjective-noun pairings. For example, "son" is clearly weighted on the word "sohn", yet there is also attention given to "kleiner".

Figure 9.8. Test Translation Example: Decoder Self-Attention Layers 5 and 6



You selected this example arbitrarily from the test set to get a sense of the translation capability of the model. The attention plots appear to show that the model is picking up on relations in the sentence, but the word importance is still strongly positional in nature. By that, we mean the German word at the current position in the original sentence is generally translated to the English version of the word at the same or similar position in the target output.

TranslationTransformer Inference Example 2

Have a look at another example, this time from the validation set, where the ordering of clauses in the input sequence and the output sequence are different, and see how the attention plays out. Load and print the data for the validation sample at index 25 in the next listing.

Listing 9.27. Load sample at *valid_data* index 25

```
>>> example_idx = 25
...
>>> src = vars(valid_data.examples[example_idx])['src']
```

```

>>> trg = vars(valid_data.examples[example_idx])['trg']
...
>>> print(f'src = {src}')
>>> print(f'trg = {trg}')
src = ['zwei', 'hunde', 'spielen', 'im', 'hohen', 'gras', 'mit',
trg = ['two', 'dogs', 'play', 'with', 'an', 'orange', 'toy', 'in'

```

Even if your German comprehension is not great, it seems fairly obvious that the orange toy ("orangen spielzeug") is at the end of the source sentence, and the in the tall grass is in the middle. In the English sentence, however, "in tall grass" completes the sentence, while "with an orange toy" is the direct recipient of the "play" action, in the middle part of the sentence. Translate the sentence with your model.

Listing 9.28. Translate the validation data sample

```

>>> translation, attention = translate_sentence(src, SRC, TRG, mo
>>> print(f'translation = {translation}')
translation = ['two', 'dogs', 'are', 'playing', 'with', 'an', 'or

```

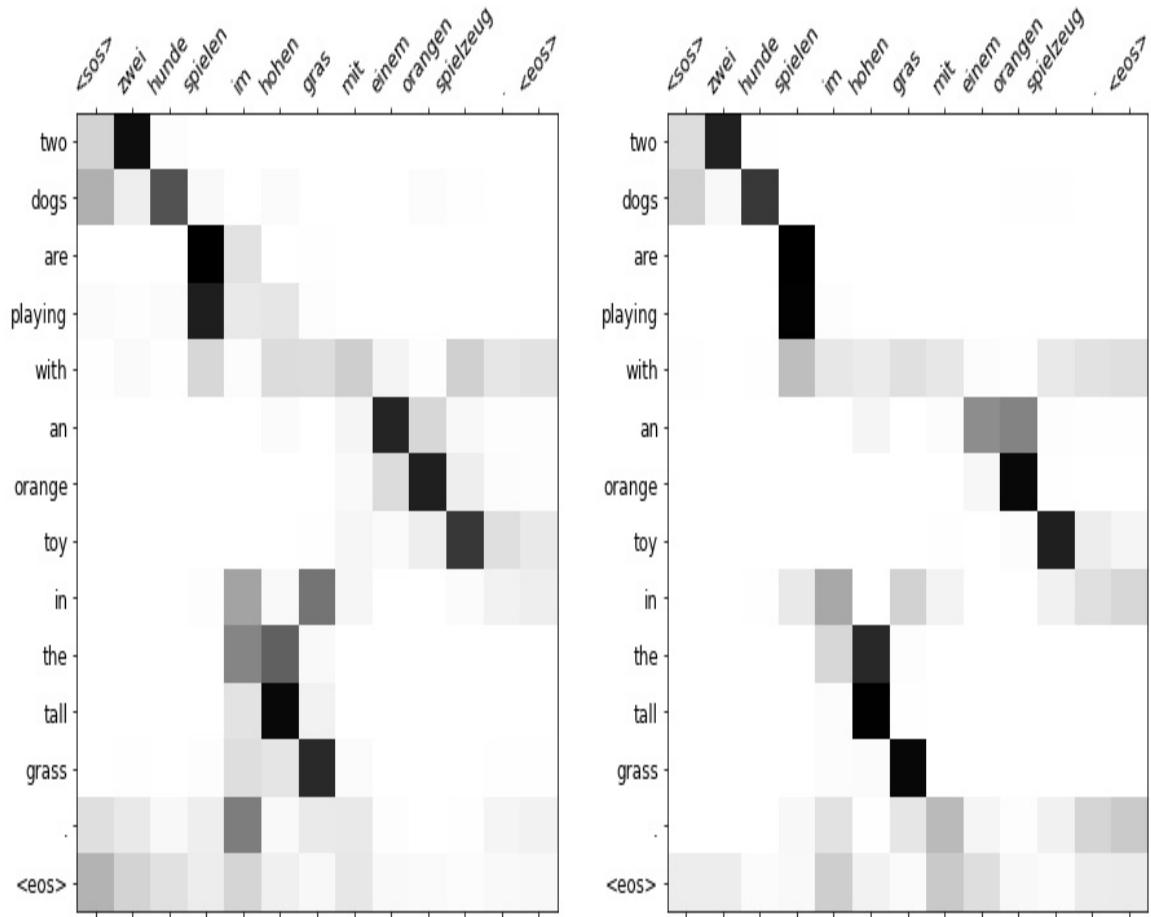
This is a pretty exciting result for a model that took about 15 minutes to train (depending on your computing power). Again, plot the attention weights by calling the `display_attention()` function with the `src`, `translation` and `attention`.

Listing 9.29. Visualize the self-attention weights for the validation example translation

```
>>> display_attention(src, translation, attention)
```

Here we show the plots for the last two layers (5 and 6).

Figure 9.9. Validation Translation Example: Decoder Self-Attention Layers 5 and 6



This sample excellently depicts how the attention weights can break from the position-in-sequence mold and actually attend to words later or earlier in the sentence. It truly shows the uniqueness and power of the multi-head self-attention mechanism.

To wrap up the section, you will calculate the BLEU (bilingual evaluation under study) score for the model. The `torchtext` package supplies a function, `bleu_score`, for doing the calculation. You use the following function, again from Mr. Trevett's notebook, to do inference on a dataset and return the score.

```
>>> from torchtext.data.metrics import bleu_score
...
>>> def calculate_bleu(data, src_field, trg_field, model, device,
...     trgs = []
...     pred_trgs = []
...     for datum in data:
```

```

...
    src = vars(datum)['src']
...
    trg = vars(datum)['trg']
...
    pred_trg, _ = translate_sentence(
        src, src_field, trg_field, model, device, max_len
    # strip <eos> token
    pred_trg = pred_trg[:-1]
    pred_trgs.append(pred_trg)
    trgs.append([trg])
...
...
    return bleu_score(pred_trgs, trgs)

```

Calculate the score for your test data.

```

>>> bleu_score = calculate_bleu(test_data, SRC, TRG, model, devic
>>> print(f'BLEU score = {bleu_score*100:.2f}')
BLEU score = 37.68

```

To compare to Ben Trevett's tutorial code, a convolutional sequence-to-sequence model [356] achieves a 33.3 BLEU and the smaller-scale Transformer scores about 35. Your model uses the same dimensions of the original "Attention Is All You Need" Transformer, hence it is no surprise that it performs well.

[346] <http://www.adeveloperdiary.com/data-science/deep-learning/nlp/machine-translation-using-attention-with-pytorch/>

[347] Pytorch Sequence-to-Sequence Modeling With nn.Transformer Tutorial: <https://simpletransformers.ai/docs/multi-label-classification/>

[348] List of ISO 639 language codes on Wikipedia
(https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes)

[349] Pytorch nn.Transformer source:
<https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/transformer>

[350] einops: <https://github.com/arogozhnikov/einops>

[351] Pytorch torch.nn.Transformer documentation:
<https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html>

[352] Pytorch nn.Module documentation:

<https://pytorch.org/docs/stable/generated/torch.nn.Module.html>

[353] Japanese StackExchange answer with counts of Japanese characters (<https://japanese.stackexchange.com/a/65653/56506>)

[354] Trevett,Ben - PyTorch Seq2Seq: <https://github.com/bentrevett/pytorch-seq2seq>

[355] Trevett,Ben - Attention Is All You Need Jupyter notebook: <https://github.com/bentrevett/pytorch-seq2seq/blob/master/6%20-%20Attention%20is%20All%20You%20Need.ipynb>

[356] Trevett,Ben - Convolutional Sequence to Sequence Learning: <https://github.com/bentrevett/pytorch-seq2seq/blob/master/5%20-%20Convolutional%20Sequence%20to%20Sequence%20Learning.ipynb>

9.5 Bidirectional backpropagation and "BERT"

Sometimes you want to predict something in the middle of a sequence—perhaps a masked-out word. Transformers can handle that as well. And the model doesn't need to be limited to reading your text from left to right in a "causal" way. It can read the text from right to left on the other side of the mask as well. When generating text, the unknown word your model is trained to predict is at the end of the text. But transformers can also predict an interior word, for example, if you are trying to unredacted the secret blacked-out parts of the Meuller Report.

When you want to predict an unknown word *within* your example text you can take advantage of the words before and *after* the masked word. A human reader or an NLP pipeline can start wherever they like. And for NLP you always have a particular piece of text, with finite length, that you want to process. So you could start at the end of the text or the beginning... or *both*! This was the insight that BERT used to create task-independent embeddings of any body of text. It was trained on the general task of predicting masked-out words, similar to how you learned to train word embeddings using skip-grams in Chapter 6. And, just as in word embedding training, BERT created a lot of useful training data from unlabeled text simply by masking out

individual words and training a bidirectional transformer model to restore the masked word.

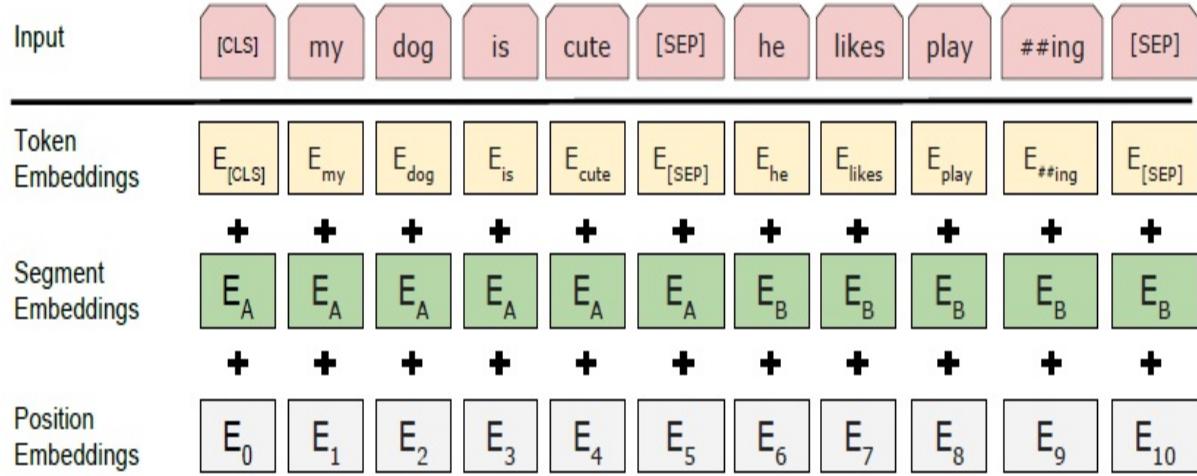
In 2018, researchers at Google AI unveiled a new language model they call BERT, for "Bi-directional Encoder Representations from Transformers" [\[357\]](#). The "B" in "BERT" is for "bidirectional." It isn't named for a Sesame Street character it means "Bidirectional Encoder Representations from Transformers" - basically just a bidirectional transformer. Bidirectional transformers were a huge leap forward for machine-kind. In the next chapter, chapter 9, you'll learn about the three tricks that helped Transformers (souped-up RNNs) reach the top of the leaderboard for many of the hardest NLP problems. Giving RNNs the ability to read in both directions simultaneously was one of these innovative tricks that helped machines surpass humans at reading comprehension tasks.

The BERT model, which comes in two flavors (configurations) - BERT_{BASE} and BERT_{LARGE} - is comprised of a stack of encoder transformers with feedforward and attention layers. Different from transformer models that preceded it, like OpenAI GPT, BERT uses masked language modeling (MLM) objective to train a deep bi-directional transformer. MLM involves randomly masking tokens in the input sequence and then attempting to predict the actual tokens from context. More powerful than typical left-to-right language model training, the MLM objective allows BERT to better generalize language representations by joining the left and right context of a token in all layers. The BERT models were pre-trained in a semi-supervised fashion on the English Wikipedia sans tables and charts (2500M words), and the BooksCorpus (800M words and upon which GPT was also trained). With simply some tweaks to inputs and the output layer, the models can be fine tuned to achieve state-of-the-art results on specific sentence-level and token-level tasks.

9.5.1 Tokenization and Pre-training

You The input sequences to BERT can ambiguously represent a single sentence or a pair of sentences. BERT uses WordPiece embeddings with the first token of each sequence always set as a special [CLS] token. Sentences are distinguished by a trailing separator token, [SEP]. Tokens in a sequence

are further distinguished by a separate segment embedding with either sentence A or B assigned to each token. Additionally, a positional embedding is added to the sequence, such that each position the input representation of a token is formed by summation of the corresponding token, segment, and positional embeddings as shown in the figure below (from the published paper):



During pre-training a percentage of input tokens are masked randomly (with a [MASK] token) and the model predicts the actual token IDs for those masked tokens. In practice, 15% of the WordPiece tokens were selected to be masked for training, however, a downside of this is that during fine-tuning there is no [MASK] token. To work around this, the authors came up with a formula to replace the selected tokens for masking (the 15%) with the [MASK] token 80% of the time. For the other 20%, they replace the token with a random token 10% of the time and keep the original token 10% of the time. In addition to this MLM objective pre-training, secondary training is done for Next Sentence Prediction (NSP). Many downstream tasks, such as Question Answering (QA), depend upon understanding the relationship between two sentences, and cannot be solved with language modeling alone. For the NSP wave of training, the authors generated a simple binarized NSP task by selecting pairs of sentences A and B for each sample and labeling them as *IsNext* and *NotNext*. Fifty percent of the samples for the pre-training had selections where sentence B followed sentence A in the corpus, and for the other half sentence B was chosen at random. This plain solution shows

that sometimes one need not overthink a problem.

9.5.2 Fine-tuning

For most BERT tasks, you will want to load the $\text{BERT}_{\text{BASE}}$ or $\text{BERT}_{\text{LARGE}}$ model with all its parameters initialized from the pre-training and fine tune the model for your specific task. The fine-tuning should typically be straightforward; one simply plugs in the task-specific inputs and outputs and then commence training all parameters end-to-end. Compared to the initial pre-training, the fine-tuning of the model is much less expensive. BERT is shown to be more than capable on a multitude of tasks. For example, at the time of its publication, BERT outperformed the current state-of-the-art OpenAI GPT model on the General Language Understanding Evaluation (GLUE) benchmark. And BERT bested the top-performing systems (ensembles) on the Stanford Question Answering Dataset (SQuAD v1.1), where the task is to select the text span from a given Wikipedia passage that provides the answer to a given question. Unsurprisingly, BERT was also best at a variation of this task, SQuAD v2.0, where it is allowed that a short answer for the problem question in the text might not exist.

9.5.3 Implementation

Borrowing from the discussion on the original transformer earlier in the chapter, for the BERT configurations, L denotes the number of transformer layers. The hidden size is H and the number of self-attention heads is A . $\text{BERT}_{\text{BASE}}$ has dimensions $L=12$, $H=768$, and $A=12$, for a total of 110M parameters. $\text{BERT}_{\text{LARGE}}$ has $L=24$, $H=1024$, and $A=16$ for 340M total parameters! The large model outperforms the base model on all tasks, however depending on hardware resources available to you, you may find working with the base model more than adequate. There are are *cased* and *uncased* versions of the pre-trained models for both, the base and large configurations. The *uncased* version had the text converted to all lowercase before pre-training WordPiece tokenization, while there were no changes made to the input text for the *cased* model.

The original BERT implementation was open-sourced as part of the

TensorFlow *tensor2tensor* library [358]. A Google Colab notebook [359] demonstrating how to fine tune BERT for sentence-pair classification tasks was published by the TensorFlow Hub authors circa the time the BERT academic paper was released. Running the notebook requires registering for access to Google Cloud Platform Compute Engine and acquiring a Google Cloud Storage bucket. At the time of this writing, it appears Google continues to offer monetary credits for first-time users, but generally, you will have to pay for access to computing power once you have exhausted the initial trial offer credits.



Note

As you go deeper into NLP models, especially with the use of models having deep stacks of transformers, you may find that your current computer hardware is insufficient for computationally expensive tasks of training and/or fine-tuning large models. You will want to evaluate the costs of building out a personal computer to meet your workloads and weigh that against pay-per-use cloud and virtual computing offerings for AI. We reference basic hardware requirements and compute options in this text, however, discussion of the "right" PC setup or providing an exhaustive list of competitive computing options are outside the scope of this book. In addition to the Google Compute Engine, just mentioned, the appendix has instructions for setting up Amazon Web Services (AWS) GPU.

Accepted op-for-op Pytorch versions of BERT models were implemented as *pytorch-pre-trained-bert* [360] and then later incorporated in the indispensable HuggingFace *transformers* library [361]. You would do well to spend some time reading the "Getting Started" documentation and the summaries of the transformer models and associated tasks on the site. To install the *transformers* library, simply use `pip install transformers`. Once installed, import the `BertModel` from *transformers* using the `BertModel.from_pre-trained()` API to load one by name. You can print a summary for the loaded "bert-base-uncased" model in the listing that follows, to get an idea of the architecture.

Listing 9.30. Pytorch summary of BERT architecture

```
>>> from transformers import BertModel
>>> model = BertModel.from_pre-trained('bert-base-uncased')
>>> print(model)

BertModel(
    (embeddings): BertEmbeddings(
        (word_embeddings): Embedding(30522, 768, padding_idx=0)
        (position_embeddings): Embedding(512, 768)
        (token_type_embeddings): Embedding(2, 768)
        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
        (layer): ModuleList(
            (0): BertLayer(
                (attention): BertAttention(
                    (self): BertSelfAttention(
                        (query): Linear(in_features=768, out_features=768, bias=True)
                        (key): Linear(in_features=768, out_features=768, bias=True)
                        (value): Linear(in_features=768, out_features=768, bias=True)
                        (dropout): Dropout(p=0.1, inplace=False)
                    )
                    (output): BertSelfOutput(
                        (dense): Linear(in_features=768, out_features=768, bias=True)
                        (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
                        (dropout): Dropout(p=0.1, inplace=False)
                    )
                )
                (intermediate): BertIntermediate(
                    (dense): Linear(in_features=768, out_features=3072, bias=True)
                )
                (output): BertOutput(
                    (dense): Linear(in_features=3072, out_features=768, bias=True)
                    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
                    (dropout): Dropout(p=0.1, inplace=False)
                )
            )
        )
    )
    ... #1

    (11): BertLayer(
        (attention): BertAttention(...))
        (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
        )
        (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
        )
    )
)
```

```
) ) ) )
(pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh() ) )
```

After import a BERT model you can display its string representation to get a summary of its structure. This is a good place to start if you are considering designing your own custom bidirectional transformer. But in most cases you can use BERT directly to create encodings of English text that accurately represent the meaning of most text. A pretrained BERT model is all you may need for applications such as chatbot intent labeling (classification or tagging), sentiment analysis, social media moderation, semantic search, and FAQ question answering. And if you're considering storing embeddings in a vector database for semantic search, vanilla BERT encodings are your best bet.

In the next section you'll see an example for how to use a pretrained BERT model to identify toxic social media messages. And then you will see how to fine tune a BERT model for your application by training it for additional epochs on your dataset. You will see that fine tuning BERT can significantly improve your toxic comment classification accuracy without overfitting.

9.5.4 Fine-tuning a pre-trained BERT model for text classification

In 2018, the Conversation AI [\[362\]](#) team (a joint venture between Jigsaw and Google) hosted a Kaggle competition to develop a model to detect various types of toxicity in online social media posts. At the time, LSTM's and Convolutional Neural Networks were the state of the art. Bi-directional LSTMs with attention achieved the best scores in this competition. The promise of BERT is that it can simultaneously learn word context from words both left and right of the current word being processed by the transformer. This makes it especially useful for creating multipurpose encoding or embedding vectors for use in classification problems like detecting toxic social media comments. And because BERT is pre-trained on a large corpus, you don't need a huge dataset or supercomputer to be able to fine tune a model that achieves good performance using the power of transfer learning.

In this section you will use the library to quickly fine tune a pre-trained BERT model for classifying toxic social media posts. After that, you will make some adjustments to improve the model in your quest to combat bad behavior and rid the world of online trolls.

A toxic dataset

You can download the "Toxic Comment Classification Challenge" dataset (`archive.zip`) from kaggle.com. [\[363\]](#) You can put the data in your `$HOME/.nlpia2-data/` directory with all the other large datasets from the book, if you like. When you unzip the `archive.zip` file you'll see it contains the training set (`train.csv`) and test set (`test.csv`) as separate CSV files. In the real world you would probably combine the training and test sets to create your own sample of validation and test examples. But to make your results comparable to what you see on the competition website you will first only work with the training set.

Begin by loading the training data using pandas and take a look at the first few entries as shown in the next listing. Normally you would want to take a look at examples from the dataset to get a feel for the data and see how it is formatted. It's usually helpful to try to do the same task that you are asking the model to do, to see if it's a reasonable problem for NLP. Here are the first five examples in the training set. Fortunately the dataset is sorted to contain the nontoxic posts first, so you won't have to read any toxic comments until the very end of this section. If you have a grandmother named "Terri" you can close your eyes at the last line of code in the last code block of in this section ; -).

Listing 9.31. Load the toxic comments dataset

```
>>> import pandas as pd
>>> df = pd.read_csv('data/train.csv') #1
>>> df.head()
      comment_text  toxic  severe  obscene  threat  insul
Explanation\nWhy the edits made      0      0      0      0
D'aww! He matches this backgrou    0      0      0      0
Hey man, I'm really not trying    0      0      0      0
"\nMore\nI can't make any real     0      0      0      0
You, sir, are my hero. Any chan    0      0      0      0
```

```
>>> df.shape  
(159571, 8)
```

Whew, luckily none of the first five comments are obscene, so they're fit to print in this book.



Spend time with the data

Typically at this point you would explore and analyze the data, focusing on the qualities of the text samples and the accuracy of the labels and perhaps ask yourself questions about the data. How long are the comments in general? Does sentence length or comment length have any relation to toxicity? Consider focusing on some of the *severe_toxic* comments. What sets them apart from the merely *toxic* ones? What is the class distribution? Do you need to potentially account for a class imbalance in your training techniques?

You want to get to the training, so let's split the data set into training and validation (evaluation) sets. With almost 160,000 samples available for model tuning, we elect to use an 80-20 train-test split.

Listing 9.32. Split data into training and validation sets

```
>>> from sklearn.model_selection import train_test_split  
>>> random_state=42  
>>> labels = ['toxic', 'severe', 'obscene', 'threat', 'insult', '  
>>> X = df[['comment_text']]  
>>> y = df[labels]  
>>> X_train, X_test, y_train, y_test = train_test_split(  
...     X, y, test_size=0.2,  
...     random_state=random_state) #1
```

Now you have your data in a Pandas DataFrame with descriptive column names you can use to interpret the test results for your model.

There's one last ETL task for you to deal with, you need a wrapper function to ensure the batches of examples passed to your transformer have the right shape and content. You are going to use the `simpletransformers` library which provides wrappers for various Hugging Face models designed for classification tasks including multilabel classification, not to be confused

with multiclass or multioutput classification models. [364] The Scikit-Learn package also contains a `MultiOutputClassifier` wrapper that you can use to create multiple estimators (models), one for each possible target label you want to assign to your texts.



Important

A multilabel classifier is a model that outputs multiple different predicted discrete classification labels ('toxic', 'severe', and 'obscene') for each input. This allows your text to be given multiple different labels. Like a fictional family in Tolstoy's *Anna Karenina*, a toxic comment can be toxic in many different ways, all at the same time. You can think of a multilabel classifier as applying hashtags or emojis to a text. To prevent confusion you can call your models "taggers" or "tagging models" so others don't misunderstand you.

Since each comment can be assigned multiple labels (zero or more) the `MultiLabelClassificationModel` is your best bet for this kind of problem. According to the documentation, [365] the `MultiLabelClassificationModel` model expects training samples in the format of `["text", [label1, label2, label3, ...]]`. This keeps the outer shape of the dataset the same no matter how many different kinds of toxicity you want to keep track of. The Hugging Face `transformers` models can handle any number of possible labels (tags) with this data structure, but you need to be consistent within your pipeline, using the same number of possible labels for each example. You need a *multihot* vector of zeros and ones with a constant number of dimensions so your model knows where to put the predictions for each kind of toxicity. The next listing shows how you can arrange the batches of data within a wrapper function that you run during training and evaluation of your model.

Listing 9.33. Create datasets for model

```
>>> def get_dataset(X, y):
...     data = [[X.iloc[i][0], y.iloc[i].values.tolist()] for i in
...             range(len(y))]
...     return pd.DataFrame(data, columns=['text', 'labels'])
...
>>> train_df = get_dataset(X_train, y_train)
>>> eval_df = get_dataset(X_test, y_test)
```

```

>>> train_df.shape, eval_df.shape
((127656, 2), (31915, 2))

>>> train_df.head() #1
                                                text
0 Grandma Terri Should Burn in Trash \nGrandma T... [1, 0, 0, 0
1 , 9 May 2009 (UTC)\nIt would be easiest if you... [0, 0, 0, 0
2 "\n\nThe Objectivity of this Discussion is dou... [0, 0, 0, 0
3 Shelly Shock\nShelly Shock is. . .( ) [0, 0, 0, 0
4 I do not care. Refer to Ong Teng Cheong talk p... [0, 0, 0, 0

```

You can now see that this dataset has a pretty low bar for toxicity if mothers and grandmothers are the target of bullies' insults. This means this dataset may be helpful even if you have extremely sensitive or young users that you are trying to protect. If you are trying to protect modern adults or digital natives that are used to experiencing cruelty online, you can augment this dataset with more extreme examples from other sources.

Detect toxic comments with simpletransformers

You now have a function for passing batches of labeled texts to the model and printing some messages to monitor your progress. So it's time to choose a BERT model to download. You need to set up just a few basic parameters and then you will be ready to load a pre-trained BERT for multi-label classification and kick off the fine-tuning (training).

Listing 9.34. Setup training parameters

```

>>> import logging
>>> logging.basicConfig(level=logging.INFO) #1

>>> model_type = 'bert' #2
>>> model_name = 'bert-base-cased'
>>> output_dir = f'{model_type}-example1-outputs'

>>> model_args = {
...     'output_dir': output_dir, # where to save results
...     'overwrite_output_dir': True, # allow re-run without havi
...     'manual_seed': random_state, #3
...     'no_cache': True,
... }

```

In the listing below you load the pre-trained `bert-base-cased` model configured to output the number of labels in our toxic comment data (6 total) and initialized for training with your `model_args` dictionary.[\[366\]](#)

Listing 9.35. Load pre-trained model and fine tune

```
>>> from sklearn.metrics import roc_auc_score
>>> from simpletransformers.classification import MultiLabelClassificationModel
>>> model = MultiLabelClassificationModel(
...     model_type, model_name, num_labels=len(labels),
...     args=model_args)
You should probably TRAIN this model on a downstream task to be a
for predictions and inference
>>> model.train_model(train_df=train_df) #1
```

The `train_model()` is doing the heavy lifting for you. It loads the pre-trained `BertTokenizer` for the pre-trained *bert-base-cased* model you selected and uses it to tokenize the `train_df['text']` to inputs for training the model. The function combines these inputs with the `train_df[labels]` to generate a `TensorDataset` which it wraps with a PyTorch `DataLoader`, that is then iterated over in batches to comprise the training loop.

In other words, with just a few lines of code and one pass through your data (one epoch) you've fine tuned a 12-layer transformer with 110 million parameters! The next question, is did it help or hurt the model's translation ability. Let's run inference on your evaluation set and check the results.

Listing 9.36. Evaluation

```
>>> result, model_outputs, wrong_predictions = model.eval_model(e
...     acc=roc_auc_score) #1
>>> result
{'LRAP': 0.9955934600588362,
 'acc': 0.9812396881786198,
 'eval_loss': 0.04415484298031397}
```

The ROC (Radio Operating Curve) AUC (Area Under the Curve) metric balances all the different ways a classifier can be wrong by computing the integral (area) under the precision vs recall plot (curve) for a classifier. This ensures that models which are confidently wrong are penalized more than models that are closer to the truth with their predicted probability values. And

the `roc_auc_score` within this `simpletransformers` package will give you the micro average of all the examples and all the different labels it could have chosen for each text.

The ROC AUC micro average score is essentially the sum of all the `predict_proba` error values, or how far the predicted probability values are from the 0 or 1 values that each example was given by a human labeler. It's always a good idea to have that mental model in mind when your are measuring model accuracy. Accuracy is just how close to what your human labelers thought the correct answer was, not some absolute truth about the meaning or intent or effects of the words that are being labeled. Toxicity is a very subjective quality.

A `roc_auc_score` of 0.981 is not too bad out of the gate. While it's not going to win you any accolades [\[367\]](#), it does provide encouraging feedback that your training simulation and inference is setup correctly.

The implementations for `eval_model()` and `train_model()` are found in the base class for both `MultiLabelClassificationModel` and `ClassificationModel`. The evaluation code will look familiar to you, as it uses the `with torch.no_grad()` context manager for doing inference, as one would expect. Taking the time to look at the method implementations is suggested. Particularly, `train_model()` is helpful for viewing exactly how the configuration options you select in the next section are employed during training and evaluation.

A better BERT

Now that you have a first cut at a model you can do some more fine tuning to help your BERT-based model do better. And "better" in this case simply means having a higher AUC score. Just like in the real world, you'll need to decide what better is in your particular case. So don't forget to pay attention to how the models predictions are affecting the user experience for the people or businesses using your model. If you can find a better metric that more directly measures what "better" means for your users you should use that in place of the AUC score for your application you should substitute it in this code.

Building upon the training code you executed in the previous example, you'll work on improving your model's accuracy. Cleaning the text a bit with some preprocessing is fairly straightforward. The book's example source code comes with a utility `TextPreprocessor` class we authored to replace common misspellings, expand contractions and perform other miscellaneous cleaning such as removing extra white-space characters. Go ahead and rename the `comment_text` column to `original_text` in the loaded `train.csv` dataframe. Apply the preprocessor to the original text and store the refined text back to a `comment_text` column.

Listing 9.37. Preprocessing the comment text

```
>>> from preprocessing.preprocessing import TextPreprocessor
>>> tp = TextPreprocessor()
loaded ./inc/preprocessing/json/contractions.json
loaded ./inc/preprocessing/json/misc_replacements.json
loaded ./inc/preprocessing/json/misspellings.json
>>> df = df.rename(columns={'comment_text':'original_text'})
>>> df['comment_text'] = df['original_text'].apply(
...     lambda x: tp.preprocess(x)) #1
>>> pd.set_option('display.max_colwidth', 45)
>>> df[['original_text', 'comment_text']].head()
      original_text          comment_t
0  Explanation\nWhy the edits ...  Explanation Why the edits made
1    D'aww! He matches this back...  D'aww! He matches this backgro
2  Hey man, I'm really not try...  Hey man, i am really not tryin
3  "\nMore\nI can't make any r...  " More I cannot make any real
4    You, sir, are my hero. Any ...  You, sir, are my hero. Any cha
```

With the text cleaned, turn your focus to tuning the model initialization and training parameters. In your first training run, you accepted the default input sequence length (128) as an explicit value for `max_sequence_length` was not provided to the model. The BERT-base model can handle sequences of a maximum length of 512. As you increase `max_sequence_length` you may need to decrease `train_batch_size` and `eval_batch_size` to fit tensors into GPU memory, depending on the hardware available to you. You can do some exploration on the lengths of the comment text to find an optimal max length. Be mindful that at some point you'll get diminishing returns, where longer training and evaluation times incurred by using larger sequences do not yield a significant improvement in model accuracy. For this example pick a `max_sequence_length` of 300, which is between the default of 128 and the

model's capacity. Also explicitly select `train_batch_size` and `eval_batch_size` to fit into GPU memory.



Warning

You'll quickly realize your batch sizes are set too large if a GPU memory exception is displayed shortly after training or evaluation commences. And you don't necessarily want to maximize the batch size based on this warning. The warning may only appear late in your training runs and ruin a long running training session. And larger isn't always better for the `batch_size` parameter. Sometimes smaller batch sizes will help your training be a bit more stochastic (random) in its gradient descent. Being more random can sometimes help your model jump over ridges and saddle points in your the high dimensional nonconvex error surface it is trying to navigate.

Recall that in your first fine-tuning run, the model trained for exactly one epoch. Your hunch that the model could have trained longer to achieve better results is likely correct. You want to find the sweet spot for the amount of training to do before the model overfits on the training samples. Configure options to enable evaluation during training so you can also set up the parameters for early stopping. The evaluation scores during training are used to inform early stopping. So set `evaluation_during_training=True` to enable it, and set `use_early_stopping=True` also. As the model learns to generalize, we expect oscillations in performance between evaluation steps, so you don't want to stop training just because the accuracy declined from the previous value in the latest evaluation step. Configure the *patience* for early stopping, which is the number of consecutive evaluations without improvement (defined to be greater than some delta) at which to terminate the training. You're going to set `early_stopping_patience=4` because you're somewhat patient but you have your limits. Use `early_stopping_delta=0` because no amount of improvement is too small.

Saving these transformers models to disk repeatedly during training (e.g. after each evaluation phase or after each epoch) takes time and disk space. For this example, you're looking to keep the *best* model generated during training, so specify `best_model_dir` to save your best-performing model. It's convenient to save it to a location under the `output_dir` so all your training

results are organized as you run more experiments on your own.

Listing 9.38. Setup parameters for evaluation during training and early stopping

```
>>> model_type = 'bert'
>>> model_name = 'bert-base-cased'
>>> output_dir = f'{model_type}-example2-outputs' #1
>>> best_model_dir = f'{output_dir}/best_model'
>>> model_args = {
...     'output_dir': output_dir,
...     'overwrite_output_dir': True,
...     'manual_seed': random_state,
...     'no_cache': True,
...     'best_model_dir': best_model_dir,
...     'max_seq_length': 300,
...     'train_batch_size': 24,
...     'eval_batch_size': 24,
...     'gradient_accumulation_steps': 1,
...     'learning_rate': 5e-5,
...     'evaluate_during_training': True,
...     'evaluate_during_training_steps': 1000,
...     'save_eval_checkpoints': False,
...     "save_model_every_epoch": False,
...     'save_steps': -1, # saving model unnecessarily takes time
...     'reprocess_input_data': True,
...     'num_train_epochs': 5, #2
...     'use_early_stopping': True,
...     'early_stopping_patience': 4, #3
...     'early_stopping_delta': 0,
... }
```

Train the model by calling `model.train_train_model()`, as you did previously. One change is that you are now going to `evaluate_during_training` so you need to include an `eval_df` (your validation data set). This allows your model to estimate how well your model will perform in the real world while it is still training model. If the validation accuracy starts to degrade for several (`early_stoping_patience`) epochs in a row, your model will stop the training so it doesn't continue to get worse.

Listing 9.39. Load pre-trained model and fine tune with early stopping

```
>>> model = MultiLabelClassificationModel(
...     model_type, model_name, num_labels=len(labels),
...     args=model_args)
```

```
>>> model.train_model(  
...     train_df=train_df, eval_df=eval_df, acc=roc_auc_score,  
...     show_running_loss=False, verbose=False)
```

Your *best* model was saved during training in the `best_model_dir`. It should go without saying that this is the model you want to use for inference. The evaluation code segment is updated to load the model by passing `best_model_dir` for the `model_name` parameter in the `model` class' constructor.

Listing 9.40. Evaluation with the best model

```
>>> best_model = MultiLabelClassificationModel(  
...     model_type, best_model_dir,  
...     num_labels=len(labels), args=model_args)  
>>> result, model_outputs, wrong_predictions = best_model.eval_mo  
...     eval_df, acc=roc_auc_score)  
>>> result  
{'LRAP': 0.996060542761153,  
 'acc': 0.9893854727083252,  
 'eval_loss': 0.040633044850540305}
```

Now that's looking better. A 0.989 accuracy puts us in contention with the top challenge solutions of early 2018. And perhaps you think that 98.9% accuracy may be a little too good to be true. You'd be right. Someone fluent in German would need to dig into several of the translations to find all the translation errors your model is making. And the false negatives—test examples incorrectly marked as correct—would be even harder to find.

If you're like me, you probably don't have a fluent German translator lying around. So here's a quick example of a more English-focused translation application that you may be able to appreciate more, grammar checking and correcting. And even if you are still an English learner, you can appreciate the benefit of having a personalized tool for helping you write. A personalized grammar checker may be your personal killer app that helps you develop strong communication skills and advance your NLP career.

[357] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding: <https://arxiv.org/abs/1810.04805> (Devlin, Jacob et al. 2018)

[358] tensor2tensor library: <https://github.com/tensorflow/tensor2tensor>

[359] BERT Fine-tuning With Cloud TPUS:
https://colab.research.google.com/github/tensorflow/tpu/blob/master/tools/colab_tutorial.ipynb

[360] pytorch-pre-trained-bert: <https://pypi.org/project/pytorch-pre-trained-bert/>

[361] HuggingFace transformers: <https://huggingface.co/transformers/>

[362] Conversation AI: (<https://conversationai.github.io/>)

[363] Jigsaw toxic comment classification challenge on Kaggle
(<https://www.kaggle.com/datasets/julian3833/jigsaw-toxic-comment-classification-challenge>)

[364] SciKit Learn documentation on multiclass and multioutput models(<https://scikit-learn.org/stable/modules/multiclass.html#multilabel-classification>)

[365] Simpletransformers Multi-Label Classification documentation
(<https://simpletransformers.ai/docs/multi-label-classification/>)

[366] See "Configuring a Simple Transformers Model" section of the following webpage for full list of options and their defaults:
<https://simpletransformers.ai/docs/usage/>

[367] Final leader board from the Kaggle Toxic Comment Classification Challenge: <https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/leaderboard>

9.6 Test Yourself

1. How is the input and output dimensionality of a transformer layer different from any other deep learning layer like CNN, RNN, or LSTM layers?
2. How could you expand the information capacity of an transformer network like BERT or GPT-2?

3. What is a rule of thumb for estimating the information capacity required to get high accuracy on a particular labeled dataset?
4. What is a good measure of the relative information capacity of 2 deep learning networks?
5. What are some techniques for reducing the amount of labeled data required to train a transformer for a problem like summarization?
6. How do you measure the accuracy or loss of a summarizer or translator where there can be more than one right answer?

9.7 Summary

- By keeping the inputs and outputs of each layer consistent, transformers gained their key superpower—*infinite stackability*
- Transformers combine three key innovations to achieve world-changing NLP power: BPE tokenization, multi-head attention, and positional encoding.
- The GPT transformer architecture is the best choice for most text generation tasks such as translation and conversational chatbots
- Despite being more than 5 years old (when GPT-4 was released) the BERT transformer model is still the right choice for most NLU problems.
- If you chose a pretrained model that is efficient, you can fine-tune it to achieve competitive results for many difficult Kaggle problems, using only affordable hardware such as a laptop or free online GPU resources