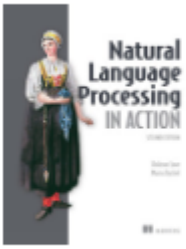

Natural Language Processing in Action, Second Edition

Cole Howard, Maria Dyshel, Hobson Lane, Hannes Hapke

1 Machines that read and write: A natural language processing overview

This chapter covers

- The power of human language
- How natural language processing 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. Words can change minds, and they can change the world. To harness the power of words, you need to understand how *natural language processing* (NLP) works and how you can make it work for you. Recent advancements in NLP have precipitated a technology explosion in almost every aspect of society and business. This chapter will open your eyes to the power of NLP and help you identify ways to employ it in your work and your life.

When you build machines that read and write words really well, they begin to seem like *artificial intelligence* (AI). In fact, usually, when people discuss the topic of "AI" on social media or in the news, they are actually referring to conversational NLP. After reading this chapter, you'll likely be a bit more discerning in how you use the term *AI* and see past some of the common misconceptions and "hype" often associated with the topic. You will learn how to build NLP software as well as how it can be integrated into larger systems to create intelligent, useful behavior that helps you achieve your goals. Most importantly, you will become a smart user of AI and NLP systems, ensuring your software delivers on its promises. As you learn how to build machines that read and write words, you will be plugging yourself into the powerful field of NLP and conversational AI that you can employ to build a better world for us all.

1.1 Programming languages vs. NLP

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. Both can communicate thoughts from human to human, human to machine, or even machine to machine. Both natural and programming languages feature the concept

of *tokens*, which you can think of as *words*, for now. Whether your text is written in a natural language or programming language, the first thing a machine does is split the text into tokens. In programming languages, the variety of these tokens is usually small—for example, the Python programming language uses just 33 reserved keywords. Conversely, there are hundreds of thousands of possible tokens in a natural language's vocabulary.

Both natural and programming languages also use *grammars*. A grammar is a set of rules that tell you how to combine words in a sequence to create an expression or statement that others will understand. And the words *expression* and *statement* mean similar things in both a computer science and an English grammar class; they give you a way to create grammar rules for processing text. You may have heard of *regular expressions* in computer science. In this book, you will use regular expressions to match patterns in all kinds of text, including natural language and computer programs. But regular expressions are just baby steps compared to the machine learning NLP approaches you will learn how to use.

Despite these similarities between programming and natural language, you need new skills and new tools to process natural language with a machine. 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, in a way that's understandable to humans. They are unambiguous—meaning there is only one way to understand a line of code. And programming languages only need to be *processed* by machines, rather than *understood*. Some programming languages are directly transformed into machine-readable code in a process called *compilation*. Others, like Python, are *interpreted*, meaning that another program, called an *interpreter*, goes over the code line by line and executes it. 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 does not need abstractions or mental models of the computer program to understand anything outside of the world of ones and zeros it is processing.

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.[1] As it was dying, Alex the parrot communicated with its owner in seemingly profound ways, saying, "You be good. … I love you."[2]

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 programs that can be run and acted on by machines. But that's exactly what you will learn how to do with this book. The machine learning process is a form of programming, and when used for NLP, those programs can 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* that allow the computer to break a sentence into its parts and understand the connections between those

parts. In this book, you will discover Python packages that allow you to analyze natural language text, compare it to other pieces of text, summarize it, and even generate new text from it. But there is no single algorithm or Python package that takes natural language text and turns it into machine instructions for automatic computation or execution. Stephen Wolfram, a scientist and the creator of Mathematica and Wolfram Alpha, has essentially spent his life trying to build a general-purpose, intelligent "computational" machine that can interact with us in plain English. He has even resorted to assembling a system out of many different NLP and AI algorithms that must be constantly expanded and evolved to handle new kinds of natural language instructions.[3]

With this book, you can build on the shoulders of giants. If you understand all the concepts in this book, you too will be able to combine these approaches to create remarkably intelligent conversational chatbots. You will have the skills to join the movement of people who create open source, ethical alternatives to ChatGPT—or whatever comes next in this world of rent-seeking AI apps.[4] And you will also learn how you can apply your newly acquired knowledge to build a fairer and more collaborative world.

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 statements, 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. Think about the sentence, "The chicken is ready to eat." This could mean that a live chicken is about to eat its breakfast—or that a cooked chicken is ready in the oven. This fuzziness of natural language leaves the interpretation of each word open to you and introduces interesting challenges in understanding and generating human language.

A natural language processing system is called a *pipeline* because natural language must be processed in several stages. Natural language text flows in one end, and text or data flows out of the other end, depending on what sections of "pipe" (Python code) you include in your pipeline. It's like a conga line of Python snakes passing the data along from one to the next.

This book will teach you how to write software that transforms simple text commands into applications that carry on a complete, human-like conversation. That 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

Natural language understanding (NLU) is a subfield of NLP that deals with machines understanding and analyzing the meaning of natural language. An important part of NLU is the automatic processing of text to extract a numerical representation of the *meaning* of that text. This is the *NLU* part of NLP. The numerical representation of the meaning of natural language usually takes the form of a row of numbers, or a vector. It's easy for computers to process vectors and do all kinds of operations with them.

You will learn various ways to represent natural language as vectors. The function that turns text into numerical vectors is called a *vectorizer*, or *encoder*, in the scikit-learn package (chapter 3). In chapters 3 and 4, you will get familiar with vector representations of the text, such as token count vectors and term frequency vectors. You will learn how to use term frequency vectors to implement keyword search, implement full text search, and even detect toxic social media messages. In chapter 6, you'll learn about a more advanced vector representation of language, called an *embedding*. Embeddings allow you to do math on the meaning or *semantics* of words, rather than just their occurrence counts. You will learn how search engines use embeddings to understand what your search query means, so they can help you find web pages that contain the information you are looking for. By the end of chapter 6, you will know how to create a hybrid search engine that combines the best of each of these approaches to natural language encoding.

Figure 1.1 shows how the NLU portion of an NLP pipeline takes in raw text and outputs a numerical representation of the meaning of that text.

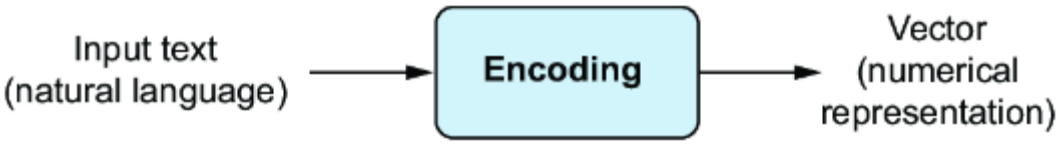

Figure 1.1 Encoding natural language

Once your natural language input is in numerical form, there are hundreds of ways to extract meaning from it. Machines have been able to accomplish many common NLU tasks with high accuracy for quite some time:

- Semantic search
- Paraphrase recognition
- Intent classification
- Sentiment analysis
- Topic modeling
- Authorship attribution

Over the years, advances in deep learning have made it possible to solve many NLU tasks that were impossible only 10 years ago:

- Analogy problem solving
- Reading comprehension
- Extractive summarization and question answering

However, there remain many NLU tasks at which humans significantly outperform machines. Some problems require the machine to have commonsense knowledge, learn the logical relationships between those commonsense facts, and use all of this on the context surrounding a particular piece of text. This makes the following problems much more difficult for machines:

- Euphemism and pun recognition

- Humor and sarcasm recognition
- Hate speech and troll detection
- Logical entailment and fallacy recognition
- Knowledge extraction

However, the most recent and advanced NLP programs, *large language models* (LLMs), are much better at completing these difficult tasks, and their effectiveness continues improving over time. In this book, you'll learn many of the state-of-the-art approaches to NLU that made solving problems like these possible. Armed with this knowledge, you will have the tools you need to create highly effective NLU pipelines that are optimized for your own use case and can tackle even those *extra* challenging problems.

1.1.2 Natural language generation

A decade before writing this book, the idea that machines could easily generate human-sounding text seemed futuristic. But at the time of writing, only about a year and a half after tools like ChatGPT were introduced to the mainstream, the fact that machines regularly create custom, readable text based on the numerical representation of a person's intent and sentiment feels more like a commonplace reality to most than something dreamed up in a sci-fi novel. This innovation comes from the *natural language generation* (NLG) side of NLP. Machines can generate text in several ways, but unless your algorithm does explicit string manipulation (e.g., pasting the user's name into a Hello {{name}}! template), it will probably represent its output as a sequence of numbers. To turn those numbers into human-readable language, *decoding*, a process inverse to *encoding*, is required, as shown in figure 1.2.

Figure 1.2 Decoding: The final step for NLG

You will soon master many common NLG tasks that build on your NLU skills. The following tasks mainly rely on your ability to *encode* natural language into meaningful embedding vectors with NLU:

- Synonym substitution
- Answering frequently asked questions (information retrieval)
- Autocompleting sentences in emails and messages
- Retrieval-augmented generation
- Spelling and grammar correction

Once you understand how to accomplish these foundational tasks for honing your NLU skills, more advanced NLG tasks like these will be within your reach:

- Abstractive summarization and simplification
- Machine translation with neural networks

- Sentence paraphrasing
- Therapeutic conversational AI
- Factual question generation
- Discussion facilitation and moderation
- Argumentative essay writing

Finally, in chapter 10, you'll see how modern LLMs are able to leverage generation capabilities for the most advanced tasks, like the following:

- Participating in debate on social media
- Automatically summarizing long technical documents
- Composing natural-sounding poetry and song lyrics
- Composing jokes and sarcastic comments
- Composing programming language expressions from natural language descriptions

This last development in NLG is particularly powerful. Machines can now write correct code that comes close to matching your intent based only on a natural language description. Machines can't program themselves yet, but they may be able to soon, according to the latest (September 2024) consensus on Metaculus. The community predicts that by September, 2028, we will live in a world where "AIs program programs that can program AIs."[5]

The combination of NLU and NLG will give you the tools to create machines that interact with humans in surprising ways. You may have heard of Microsoft and OpenAI's Copilot project. GPT-J can do almost as well, and it's completely open source and open data.[6]

1.1.3 Plumbing it all together for positive-impact AI

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 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, spawning new industries and communities that thrive together.

It is our hope that by understanding how NLP works, you will become more cognizant of the ways machines are used in our daily lives—often without our knowledge—to mine our words for profit, gently guide us toward a particular outcome, or even train us to become more easily manipulated in the future. The good news is that by learning how NLP works,

you will better prepare yourself to recognize, protect yourself from, or even fight back against nefarious uses of NLP in a world filled with manipulative algorithms.

Machines that can understand and generate natural language harness the power of words. And because machines can now understand and generate text that seems human, in some situations, they are capable of acting on your behalf in the real world. One day soon, you'll likely be able to create bots that will automatically follow your wishes and accomplish the goals you program them to achieve. But beware of Aladdin's three-wishes trap. Your bots have the potential to create a tsunami of blowback for your business or your personal life. The same bots that were able to do the challenging tasks we saw in the previous section are also the ones that have caused lawyers to lose their jobs,[7] given people with eating disorders harmful dieting advice,[8] and deceived airline customers, resulting in massive reputational damage.[9] This is called the "AI control problem" or the "challenge of AI safety."[10]

The control problem and AI safety are not the only challenges you will face on your quest for positive-impact NLP. The danger of superintelligent AI that can manipulate us into giving it ever greater power and control may be decades away, but the danger of not-so-intelligent AI that deceives and manipulates us has been around for years.[11] The search and recommendation engine NLP that determines which posts you are allowed to see is not doing what *you* want; it is doing what the *platform's investors* want: stealing your attention, time, and money.

1.2 The magic of natural language

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, intentionally designed so that computers could understand them. Programming languages avoid ambiguity so that computers can always do exactly what you *tell* them to do, even if that is not always what you *want* them to do.

Computer languages can only be interpreted (or compiled) in one correct way. With NLP, you can allow your users talk to machines in their own language, rather than forcing them to learn "computerese." When software can process languages not designed for machines to understand, it is like magic—something we previously thought only humans could do.

Moreover, machines can access a massive amount of natural language text, such as from Wikipedia, to learn about the world and human thought. Google's index of natural language documents comprises well over 100 million GBs,[12] and that is just the index—and the index is incomplete! The size of the actual natural language content currently online probably exceeds 100 billion GBs.[13] This massive amount of natural language text makes NLP a useful tool.

Note Today, Wikipedia lists approximately 1,000 programming languages.[14] Wikipedia's lists of natural languages include more than 7,000 natural languages,[15] and those lists do not 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.[16,17,18]

For now, you only need to think about one natural language: English. You'll ease into more difficult languages, like Mandarin 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 that has yet to be deciphered by archaeologists and linguists. We will 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 an excellent choice for learning NLP. 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,[19] 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 vocabulary affects our thoughts.[20] For example, many Aboriginal Australian languages, such as Guugu Yimithirr and Kuuk Thaayorre, use words to describe the position of objects on their body according to the cardinal points of the compass. Aboriginal people from Chiapas in southern Mexico speak the language of Tzeltal, which also has words for the cardinal directions rather than egocentric relative directions. The Tzeltal people even use words for *uphill* and *downhill* or *altitude* to describe the location of events in time relative to the present.[21] Such people exhibit a more robust internal compass than those without this culture and language. Instead of saying an item is in their *right* hand, speakers say it is on the *north* side of their body. This use of cardinal directions in speech may even aid speakers when performing certain tasks. For example, languages that commonly refer to the directions on a compass are thought to help speakers regularly update their understanding of their orientation in the world, which, in turn, improves communication and orienteering during hunting expeditions.

Stephen Pinker flips that notion around, seeing 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."[22] Regardless of whether you think of words as affecting your thoughts or as helping you see and understand your thoughts, they are certainly 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*.

What about math? Humans can 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 thoughts, like mathematical 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* (Penguin Press 2021), about how he first "discovered" the commutative property of algebra while staring at a stereo speaker with a 6x8 grid of dots. 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 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 he had 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 to work with in your brain. You may not realize it, but as you are composing sentences, you are actually rethinking, manipulating, and repackaging these thoughts. 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 good old fashioned AI (GOFAI), researchers assumed that AI would need to learn to manipulate natural language symbols and logical statements the same way it compiles programming languages. In chapter 11, you'll learn how to teach a machine to do symbol manipulation on natural language.

But that's not the most impressive power of NLP. Think back to a time when you had a difficult email to send to someone close. Perhaps, you needed to apologize to a boss or a teacher or maybe to your partner or a close friend. Before you started typing, you probably started thinking about the words you would use or even the reasons or excuses for why you did what you did. And then, maybe 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 are 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 that they make more sense to our reader or even our future self. Every sentence in this second edition of the book has been edited several times—sometimes with the help of generous readers of the liveBook.[23] 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.[24]

But words and writing aren't the *only* way to think logically and deeply. Drawing, diagramming, and even dancing and acting out are all expressions of thought. And we imagine these drawings in our minds—sketching ideas, concepts, and thoughts in our head. And sometimes, we 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 we do almost constantly.

Reading and writing are special kinds of thought. These acts seem to compress our thoughts and make them easier to remember and manage. 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. 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*. Even though machines use very different tools to process and generate text, when we see them doing that, we associate it with the thinking processes that accompany our reading and writing. This is why people think of NLP as 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 inside your head, the real fun is when you use those words to interact with others. The act of conversation 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 say the right words to make them understand some of your thoughts. For example, when you are feeling great pain, frustration, or shock, you can use a curse word. This word selection is then likely to convey that shock and discomfort to your listener or reader. Though we cannot "program" another human with our words, we can use words to communicate extremely complex ideas.

Natural language cannot be directly translated into a precise set of mathematical operations, but it does contain information and instructions that can be extracted. These pieces of information and instruction can be stored, indexed, searched, or immediately acted upon. One of these actions could be, for example, to generate a sequence of words in response to a statement. This is the function of the "dialog engine," or chatbot, 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, using the example of Morse code. But apart from that, we focus on the words that have been put to paper ... or at least put to transistors in a computer. There are whole books on speech recognition as well as speech-to-text (STT) and text-to-speech (TTS) systems. There are ready-made open source projects for STT and TTS. If you are working on a mobile application, 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, we focus on what happens after the audio has been translated into text. This can help you build a smarter voice assistant when you add your *brains* to open source projects, such as Home Assistant[25] or Mycroft AI.[26] And you'll understand all the helpful NLP the big boys could be giving you with their voice assistants ... assuming commercial voice assistants wanted to help you with more than just lightening your wallet.

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 perform 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. Well-trained human judges could not match the performance (68% accuracy) of a simple sarcasm-detection NLP algorithm.[27] Simple *bag-of-words* (BOW) models achieve 63% accuracy, and state of the art transformer models achieve 81% accuracy.[28] 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 have extracted structured numerical data, or 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 NLP 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.[29] 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 way 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 you have some knowledge about what makes up a morning, including that the morning comes before noon, afternoon, evening, and midnight. You need to know that 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 their 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 commonsense 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 commonsense knowledge to help interpret statements that rely on this knowledge.

1.3 Applications

NLP 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 in figure 1.3 may surprise you.

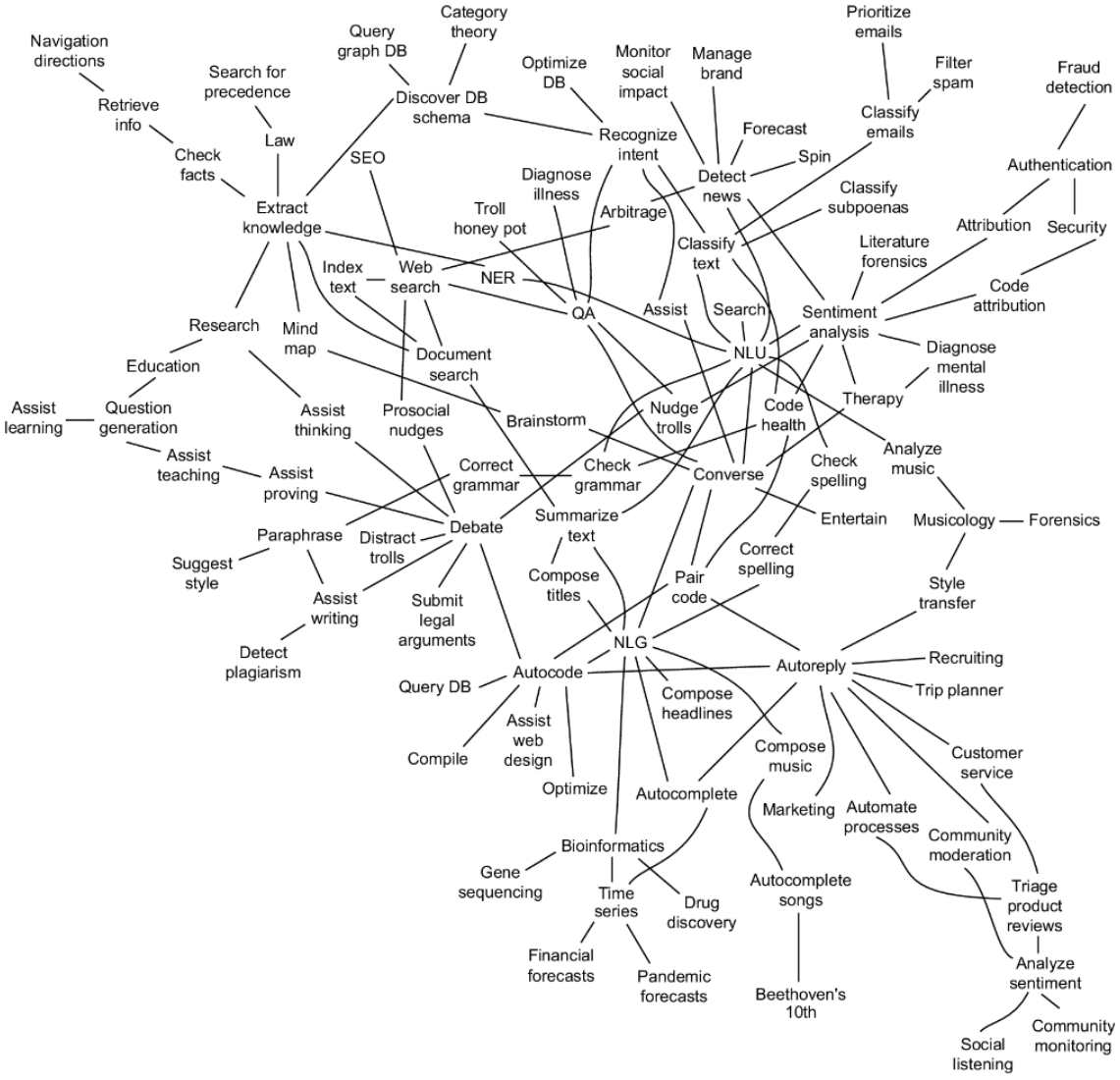

Figure 1.3 Graph of NLP applications

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 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 considers 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 even uses NLP "robot journalists" to write entire financial news articles and sporting event reports.[30] 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 and job satisfaction as well as the quality of the product. For example, chatbots can automate the responses to many customer service requests.[31] Additionally, 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 emails 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, where I am the CTO, we used NLP for the critical business process of finding developers to join our team, so we supervised our NLP pipeline carefully. NLP was allowed to filter out job applications only in situations where the candidate was nonresponsive or answered in ways that were not relevant to the questions. We also had rigorous quality control on the NLP pipeline with periodic random sampling of the model predictions and used simple models and sample-efficient NLP models[32] to focus human attention on those predictions where the machine learning was least confident—you'll learn to do it too with the predict_proba method of scikit-learn classifiers, starting with chapter 2. As a result, NLP for human relations (HR) 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, Asia, and South America. NLP helped us quickly evaluate English and technical skill before proceeding with interviews and paid take-home assignments.

Email spam filters have retained their edge in the cat-and-mouse game between spam filters and generators, but they 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.[33,34] These bots amplify the viewpoints of their owners and developers, often foreign governments or large corporations with the resources and motivation to influence popular opinion.

NLP systems can generate more than just short social network posts. They can be used to compose lengthy movie and product reviews on online shops 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. In fact, a large portion of all product reviews that bubble to the top of search results and appear on online retailers' product pages are fake. You can use NLP to help search engines and prosocial social media communities[35] detect and remove misleading or fake posts and reviews.[36]

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.[37] 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 I/O, engineers and managers overlooked concerns about the ethics of teaching chatbots to deceive humans. On most entertainment-focused 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 has been surpassed by the more urgent challenge of building AI that behaves ethically.[38] Yuval Harari's cautionary forecast of bots short-circuiting human decision making in "Homo Deus"[39] is already upon us.

NLP systems 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 customer relationship management (CRM) 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. Some inexperienced, daredevil NLP textbook authors are even letting bots author several sentences in their book—more on that later.

One of the most powerful applications of NLP is in psychology. The first conversational application in history, the ELIZA chatbot,[40] used simple pattern-matching methods on the user's input to create a surprisingly human-like conversation. Since then, chatbots that act like therapists have shown tremendous progress.[41] Commercial virtual companions, such as Xiaoice in China as well as Replika.AI and Woebot in the United States, helped hundreds of millions of lonely people survive the emotional impact of social isolation during COVID-19 lockdowns in 2020 and 2021.[42] Fortunately, you don't have to rely on engineers at large corporations to look out for your best interests. Many psychotherapy and cognitive assistants are completely free and open source.[43]

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 have proven helpful for simple tasks, like autocomplete and providing snippet suggestions. Additionally, users can often employ 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. Older code generation tools were *extractive*. Extractive text generation algorithms find the most relevant text in your history and simply regurgitate it verbatim, as a suggestion. So if the phrase *prosocial artificial intelligence* appears a lot in the text an algorithm was trained on, autocomplete will recommend the term *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 Visual Studio Code (VS-Code). It suggests entire function and class definitions, and all you have to supply is a short comment and the first line of the function definition. The following is the example TypeScript prompt shown on the Copilot home page:[44]

```
// 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. An algorithm is writing code for you to analyze the sentiment of natural language text, such as the text in your emails or personal essays. And the examples shown on the Copilot home page all favor Microsoft products and services. This means you will end up with an NLP pipeline that has a perspective on what is positive and what is not, based on Microsoft's NLP; in other words, it values what Microsoft has told it to value. Similar to how Google indirectly influenced the kind of code you wrote, 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 build a pipeline that helps you write blog posts and chatbots or even contribute to some open source datasets and algorithms. You 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 a huge influence on the direction of your code and the direction of your life.

1.4 Language through a computer's "eyes"

When you type something like, "Good morning 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 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 an 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 then use one of the responses other humans and authors have used for that statement in the past? But imagine 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 a similarity between two streams of bits that takes into account what they signify. The bits for "good" will be just as similar as those for "bad" and those for "OK."

But let's see how this approach would work before we show you a better way. We'll start by building a small, regular expression to recognize greetings and respond appropriately—our first tiny chatbot!

1.4.1 The language of locks

You might not know it, but the humble combination lock is actually a simple language processing machine with its own unique language. This section will teach you how to "speak" the language of locks as an analogy for regular expressions, but be forewarned: after reading it, you'll never see your bicycle lock the same way again!

As with natural languages, to "speak" the language of a padlock, you must follow its *grammar*, or rules and patterns of language. In the language of locks, if you "tell" your lock a password (by entering a combination) using the appropriate grammar (the correct sequence of symbols for your lock) it will immediately determine you've "told" it something particularly meaningful. And, quite impressively, it will understand its single correct response: to release the catch holding the U-shaped hasp, so you can get into your locker.

This language of locks uses regular expressions, making it a particularly simple one—but that doesn't mean it's too simple to use in a chatbot! We can, for example, use it to recognize a key phrase or command to unlock a particular action or behavior. Imagine you are building a chatbot that you want to recognize and respond to greetings (e.g., "Hello, Rosa"). This kind of language, like the language of locks, is a *formal language*, meaning 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 already 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. And that's the reason for our diversion into the mechanical (*click, whirr*)[45] language of locks.

1.4.2 Regular expressions

Regular expressions use a special class of formal language grammars, called *regular grammars*. Regular grammars have predictable, provable behavior, yet they 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[46] and qary,[47] which rely exclusively on this kind of language processing to produce some effective chatbots.

Note Regular expressions implemented in Python and POSIX (Unix) applications, such as grep, are not true regular grammars. They have language and logic features, such as lookahead and lookback, which 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.[48]

You may be saying to yourself, "I've heard of regular expressions. I use grep, but that's only for search!" And you would be 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 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 the world the user is describing.

A machine that processes this kind of language can be thought of as a formal mathematical object, an FSM, or a *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. Figure 1.4 shows how formal languages used to program finite state automata are nested inside each other, like Ukrainian matryoshka dolls.

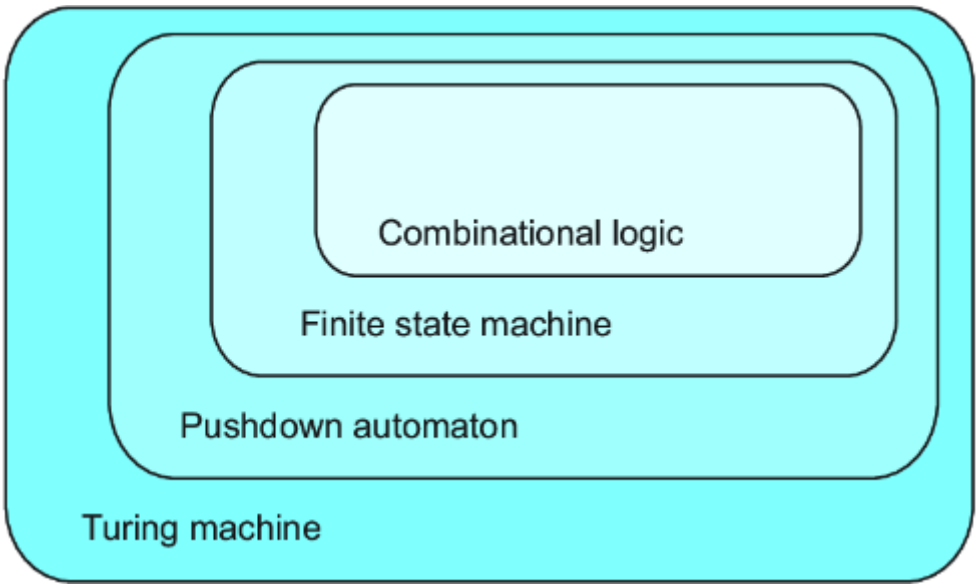

Figure 1.4 Kinds of automata

Combinatoric logic is the smallest, simplest language at the heart of this Venn diagram. *Finite state machines* encompass *combinatoric logic*, and *pushdown automata* are a superset of both. A *Turing machine* is capable of implementing the behaviors of all the other automata in this diagram.

Formal languages

Kyle Gorman describes programming languages and formal languages like this:

- 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 a context-free grammar.

- A few categories of formal language[a] are listed here in decreasing complexity, with the most complex category being recursively enumerable grammars:

a See "Chomsky Hierarchy," Wikipedia (https://en.wikipedia.org/wiki/Chomsky_hierarchy).

Natural languages

Natural languages are quite different from formal programming languages, mostly in what they are not:

- Not regular languages[a]
- Not context-free languages[b]
- Cannot be defined by any formal grammar[c]

a Shuly Wintner, "English Is Not a Regular Language" (http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=20).

b Shuly Wintner, "Is English Context-Free?" (http://cs.haifa.ac.il/~shuly/teaching/08/nlp/complexity.pdf#page=24).

c "Foundations of Python Programming" (https://runestone.academy/ns/books/published/fopp/GeneralIntro/FormalandNaturalLanguages.html).

Even though formal programming languages cannot directly implement very complex natural language pipelines, they are critical to any NLP pipeline. After all, you will use the formal programming language Python to create all of the NLP pipelines in this book. Sometimes, all you need are a few if statements and some Python string processing to accomplish what you need. This means you can create a simple chatbot without any machine learning or other nondeterministic processing.

1.5 Building a simple chatbot

Let's 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 won't have to try to think of all the different ways people can say something, like we did in this example. You won't even have to write regular expressions to build an awesome chatbot. Instead, we will guide you through building a chatbot of your own that can learn from reading (processing) a bunch of English text in later chapters, without hardcoding anything.

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. A variation of the pattern-matching approach we cover here is used in chatbots like Amazon Alexa and other virtual assistants.

For now, let's build an FSM, a regular expression that can speak a regular language. We'd like it to understand greetings—phrases like, "Open sesame" or "Hello, Rosa." Being able to respond to a greeting is an important feature of a prosocial chatbot. In high school, teachers

may chastise students for being impolite when they 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 (acknowledgment) signal to confirm receipt of each message. But our machines are going to be interacting with humans who say things like, "Good morning." 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. You will want a few different approaches in your toolbox.

Note An *intent* is a category of objectives the user may want the NLP system or chatbot to carry out in different contexts. Words like *hello* and *hi* might fall under the *greeting* intent, for example, so that the chatbot will use them when the user wants to start a conversation. Another intent might be to carry out some task or command, such as answering the query, "How do I say 'Hello' in Ukrainian?" or responding to a translation command. You'll learn about intent recognition throughout the book and put it to use in a chatbot in chapter 12.

1.5.1 Keyword-based greeting recognizer

Your first chatbot will be straight out of the '80s. If you watched the 1983 science fiction classic *WarGames*, you might remember Joshua, an AI chatbot running on the WOPR computer, programmed by Professor Steven Falken.[49] Imagine you want a chatbot to help you select a game to play, like chess ... or thermonuclear war. This approach can be extended to help you implement simple keyword-based intent recognizers, as shown in the following listing, 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 = "Thermonuclear War is a strange game. "
...     bot_reply += "The only winning move is NOT TO PLAY."
>>> else:
...     bot_reply = "Would you like to play a nice game of chess?"
>>> bot_reply
'Thermonuclear War is a strange game. The only winning move is NOT TO PLAY.'
```

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

Rule-based chatbots can be 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*, 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.

1.5.2 Pattern-based intent recognition

A keyword-based chatbot would recognize the words *Hi*, *Hello*, and *Greetings*, but it wouldn't recognize *Hiiii* or *Hiiiiiiiiii* —more excited renditions of *Hi*. Perhaps, you should hardcode the first 200 versions of *Hi*, such as ["Hi", "Hii", "Hiii", ...], or programmatically create such a list of keywords. But there's a better way, one that will allow your bot to recognize infinite variations of *Hi*: using regular expressions. Regular expression *patterns* can match text much more reliably than any hardcoded rules or lists of keywords. Regular expressions recognize patterns for any sequence of symbols or tokens. They can even be used to match sequences of symbols and other characters, such as words, part-of-speech tags, and even *n*-grams (several words in a row).[50]

With both regular expressions and keyword matchers, you need to anticipate all the kinds of words your users will use as well as how they will spell and capitalize them. So your pattern matchers will miss greetings like *Hey* or even *hi* if those strings aren't in your list of greeting words. And what if your user chooses a greeting that starts or ends with punctuation, such as *'sup* or *Hi,*? In that case, you could do *case folding* with the str.lower() method on both your greetings and the user statement, and you could add more words to your list of greetings. You could even add misspellings and typos to ensure they aren't missed. But that is a lot of manual data hardcoding in your NLP pipeline, and any time you have to fold the capitalization (case) or tokens or handcraft any preprocessing of text, you are changing the meaning of that text and destroying information that your NLP might need. For example, capitalization can be a clue to help an NLP pipeline recognize names or proper nouns. If you lowercase the name *John*, your NLP pipeline might mistakenly interpret *john* as the slang word for *toilet*.

Machine learning promises to let your text data speak for itself by relying more on the statistics in the text you are processing than on your own guesses about which patterns your pipeline needs to match. Amazingly, machine learning can be used to recognize patterns in the *meaning* (semantics) of words, not just their spelling. Once you learn how to use a machine learning approach to NLP in chapters 3–6, you will notice that much of the hard work of designing and evaluating NLP pipelines becomes automatic. And when you graduate to the much more complex and accurate *deep learning* models of chapter 7 and beyond, you will find that elements of modern NLP pipelines can still be quite brittle. You will learn to be clever about building the datasets of text you need to make deep learning NLP pipelines more robust.[51] For now, you can get started with the basics. When your user

wants to specify actions with precise patterns of characters similar to programming language commands, regular expressions shine:

```
>>> import re           #1
>>> r = "(hi|hello|hey)[ ,:.!]*([a-z]*)"     #2
>>> re.match(r, 'Hello Rosa', flags=re.IGNORECASE)    #3
<re.Match object; span=(0, 10), match='Hello Rosa'>
>>> re.match(r, "hi ho, hi ho, it's off to work ...", flags=re.IGNORECASE)
<re.Match object; span=(0, 5), match='hi ho'>
>>> re.match(r, "hey, what's up", flags=re.IGNORECASE)
<re.Match object; span=(0, 9), match='hey, what'>
```

#1 There are two "official" regular expression packages in Python. The re package is pre-installed with all versions of Python. The regular expression package includes additional features such as fuzzy pattern matching.
#2 | means "OR," and * means the preceding characters can occur 0 or more times and still match.
#3 Ignoring the character case means this regular expression will match "Hey" as well as "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 all of them 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')
<re.Match object; span=(0, 10), match='Hello Rosa'>
>>> re_greeting.match('Hello Rosa').groups()
('Hello', None, None, 'Rosa')
>>> re_greeting.match("Good morning Rosa")
<re.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")

<re.Match object; span=(0, 16), match="Good Morn'n Rosa">

>>> re_greeting.match("yo Rosa")

<re.Match object; span=(0, 7), match='yo Rosa'>
```

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

Tip The *r* before the apostrophe (r') indicates that the quoted string literal is a *raw* string. A Python raw string just makes it easier to use the backslashes used to escape special symbols within a regular expression. If you tell Python that a string is *raw*, it 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, and 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, or *false positives*, which is also a bad thing. Having both false positive and false negative errors means 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 refining the phrases it matches for the bot to behave more intelligently.

All this tedious work would still be highly unlikely to ever capture all the slang and misspellings common in human speech. Fortunately, composing regular expressions by hand isn't the only way to train a chatbot—more on that later (in the entire rest of the book). So we only use them when we need precise control over a chatbot's behavior, like 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 its response:

```
>>> my_names = set(['rosa', 'rose', 'chatty', 'chatbot', 'bot',
...     'chatterbot'])
>>> curt_names = set(['hal', 'you', 'u'])
>>> greeter_name = ''                #1
>>> match = re_greeting.match(input())
...
```

```
>>> if match:
...     at_name = match.groups()[-1]
...     if at_name in curt_names:
...         print("Good one.")
...     elif at_name.lower() in my_names:
...         print("Hi {}, How are you?".format(greeter_name))
```

#1 We do not yet know who is chatting with the bot, and we will not worry about that here.

If you run this little script and prompt the 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, it will be less responsive, but not inflammatory, to encourage politeness.[52] 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 in a scenario where this is more likely (e.g., with a larger chatbot), you should address these sorts of things.

Because of the limitations of computational resources, early NLP researchers had to use the computational power of their human brains 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—*parts of speech* —and other higher-level patterns. The core building blocks of NLP, like stemmers and tokenizers, as well as sophisticated end-to-end NLP dialog engines (i.e., chatbots) like ELIZA, were built this way, from regular expressions and pattern matching. The art of taking a pattern-matching approach to NLP discovers elegant patterns that capture just what you want, without using too many lines of regular expression code.

Tip 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.[53] Advancements in neuroscience and NLP led to the development of a *connectionist* theory of mind around the turn of the century. This new theory inspired deep learning's use of artificial neural networks, which process natural language sequences in many different ways, simultaneously and in parallel.[54,55]

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 exponentially greater computational resources as well as larger datasets to circumvent 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 regular expression examples, since they act as 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 we're

sure it will become clear by the end of this book. And it turns out machines can learn this way as well.

1.5.3 Another way to recognize greetings

Imagine you have access to an enormous database of human dialog sessions—one that contains statements paired with responses from thousands, or even millions, of conversations. In this scenario, you could certainly build a chatbot by copying your user's input and searching the database for the exact same string of characters. And then you could simply reuse one of the responses to that sentence or phrase that other humans have used in the past. That would result in a statistical, or data-driven, approach to chatbot design, and it could take the place of the quite tedious pattern-matching algorithm design.

Think about how a single typo or variation in the statement would trip up a pattern-matching bot or even a data-driven bot 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 that uses new patterns of characters never 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 often get it wrong. Words and phrases with similar meanings, like *good* and *okay*, often have different character sequences and large distances when we count character-by-character matches to measure distance. And sometimes, two words look almost the same but mean completely different things, like *bad* and *bag*. You can count the number of characters that change from one word to another with algorithms such as Jaccard and Levenshtein. But these distance or "change" counts fail to capture the essence of the relationship between two dissimilar strings of characters, as with *good* and *okay*. And they fail to account for how small spelling differences might not really be typos but, rather, completely different words, as with *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. In these cases, we use vector representations of natural language words and text and some distance metrics for those vectors. 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, so where should we start? Well, the first layer of decision 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 logbook to indicate which message the word occurred in, creating an encyclopedic index of all the

documents we have read. 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—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 1.5) 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 consider hundreds of thousands, if not millions, of possible token "denominations," one for each possible word that a speaker or author might use. Each phrase, sentence, or document we feed into our token sorting machine will come out at 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? We 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 shows a Canadian coin (or token) sorter.

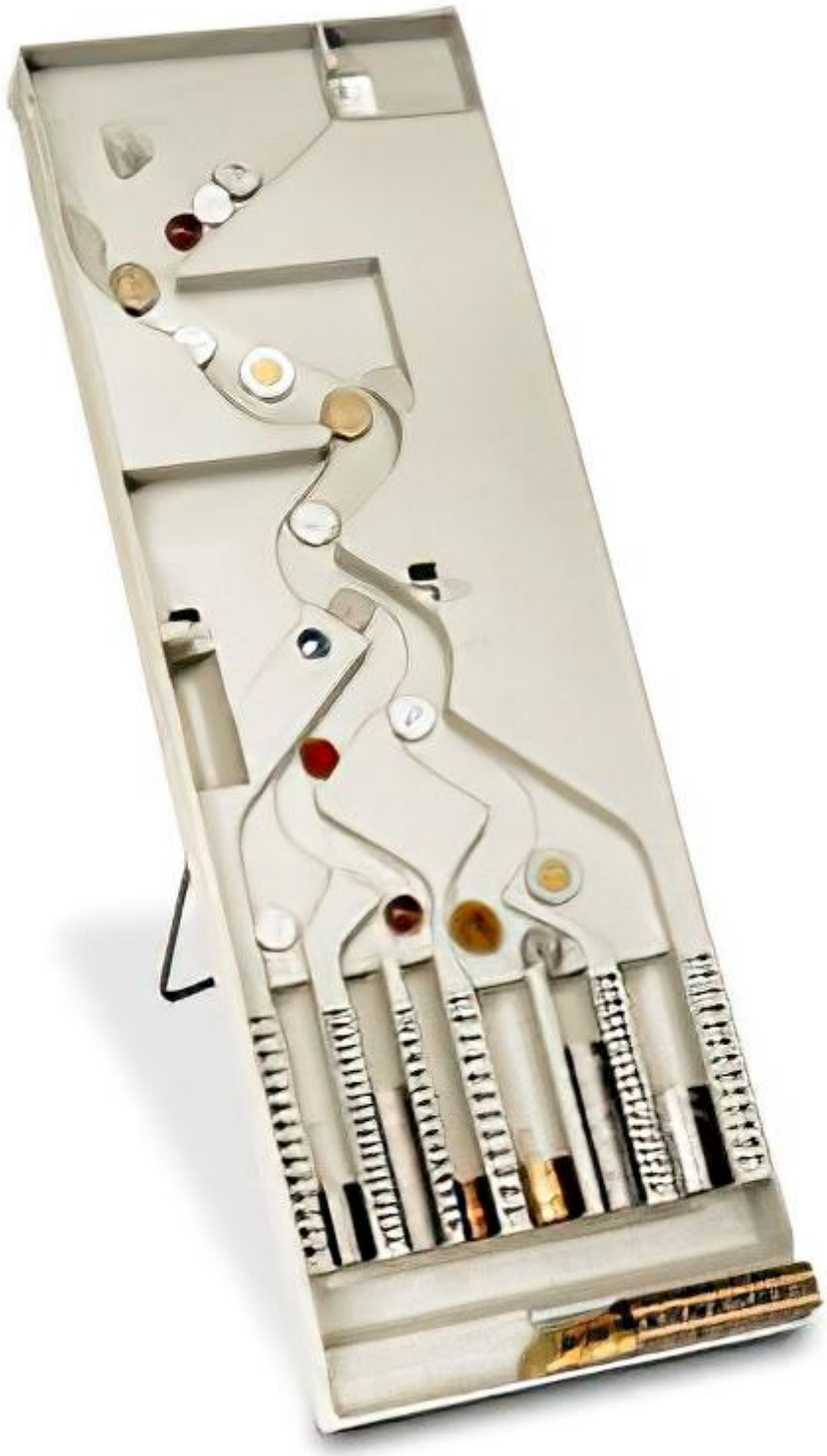

Figure 1.5 Canadian coin sorter

Here's how our token sorter fits into an NLP pipeline right after a tokenizer (see chapter 2). We have included a stop word filter as well as a rare word filter in our mechanical token sorter sketch illustrated in figure 1.6. Strings flow in from the top, and BOW vectors are created from the height profile of the token "stacks" at the bottom.

It turns out that machines can handle this BOW 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. Figure 1.6 shows a crude example, and in chapter 2, we will look at some more useful data structures for BOW vectors.

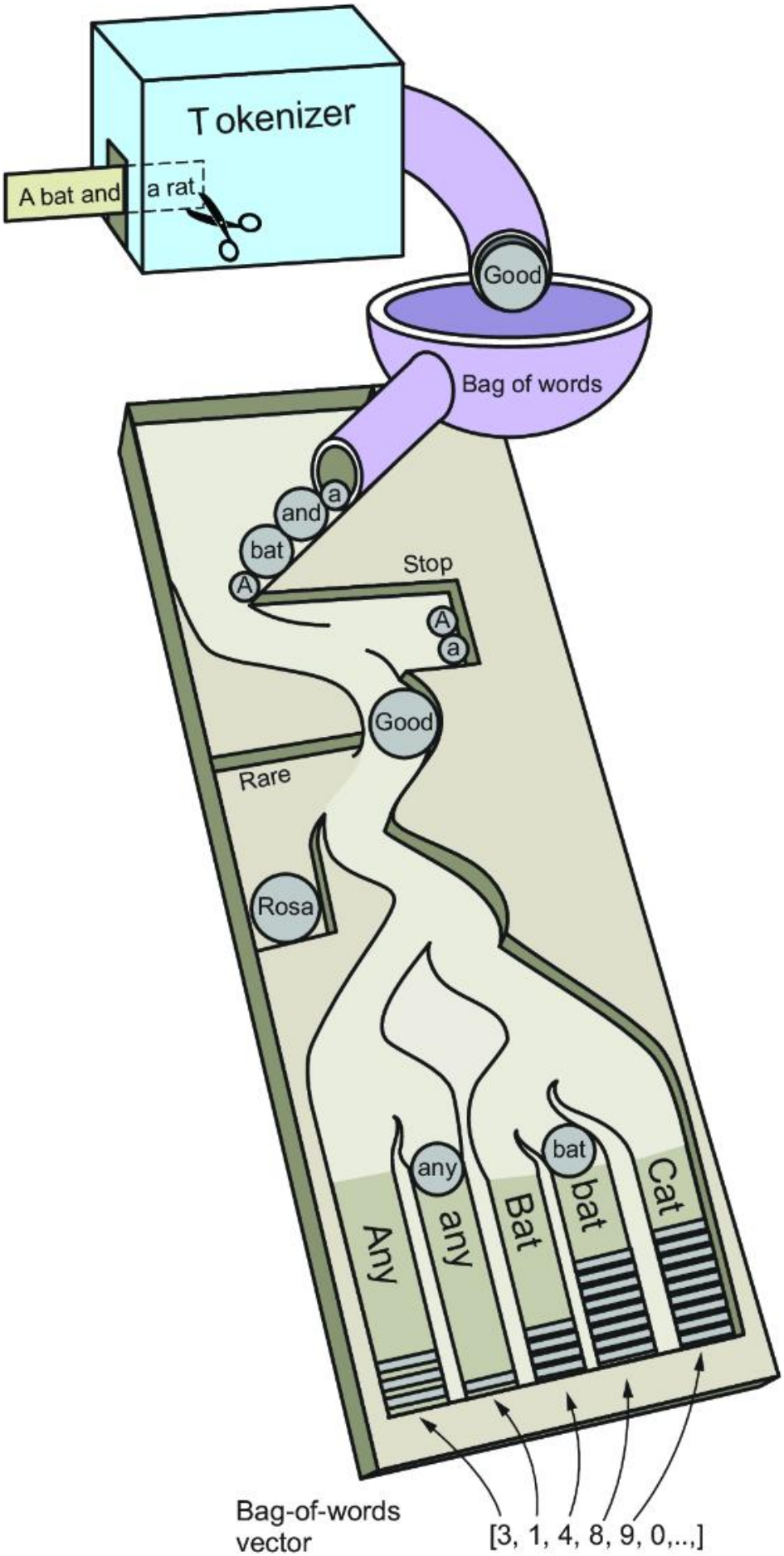
Tokenizer
A bat and
a rat
Good
Bag of words
a
and
bat
A
Stop
A
a
Good
Rare
Rosa
any
bat
Any
any
Bat
bat
Cat
Bag-of-words
vector
[3, 1, 4, 8, 9, 0,..,]

Figure 1.6 Token-sorting tray

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 many zeros and a few ones or twos scattered around wherever the word for that bin occurred. All the different ways words could be combined to create these vectors is called a *vector space*. 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*. 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, and we do just that in the next chapter. But you might also think there are better ways to represent a sentence such as, "Guten morgen Rosa." We'll learn about different ways to represent sequences of tokens in chapters 3, 4, and 6.

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 it a vector representation of that statement. This model of documents and statements and words is called a *vector space model*. Now, we can 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 brittleness in the NLP pipeline. One statistical question often asked of BOW 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, one might ask, "What is the closest BOW in our database to a BOW 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 BOW 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 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 trigram vocabulary computed from a large corpus. In chapter 3, we discuss the curse of dimensionality and some other properties that make high-dimensional vectors difficult to work with.

1.6 A brief overflight of hyperspace

In chapter 3, you will learn how to consolidate words into a smaller number of vector dimensions to deal with the *curse of dimensionality*. You may even be able to turn the curse into a blessing by using all those dimensions to identify the subtle things you want your NLU pipeline to understand. You project vectors onto each other to determine the distance between each pair. This gives you a reasonable estimate of the similarity in their *meaning*, rather than merely their statistical word usage. When you compute a vector distance this way, it is called a *cosine distance metric*. You will first use cosine distance in chapter 3, and then in chapter 4, you will uncover its true power by reducing the thousands of dimensions of topic vectors down to just a few. You can even project (*embed* is the more precise term) these vectors onto a 2D plane to have a "look" at them in plots and diagrams. This is one of the best ways to find patterns and clusters in high-dimensional data. You 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, messages, or sentences humans could write. Even though we repeat ourselves a lot, that's still a great number of possibilities. And when those tokens are each treated as separate, distinct dimensions, there is no reason we would believe "Good morning, Hobs" has any 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, ask questions like these:

- 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 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 farther apart on others.

But how can a computer assign values to each of these vector dimensions? Well, by simplifying our vector dimension questions like, "Does it contain the word *good*?" or "Does it contain the word *morning*?"—and so on. You can imagine we could 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*. You can see why computers are just now getting powerful enough to make sense of natural language. The mass of million-dimensional vectors humans can generate simply "Does not compute!" on a supercomputer of the '80s, but they are no problem on a modern commodity laptop. It's more than just raw hardware power and capacity that made NLP

practical; incremental, constant-RAM, linear-algebra algorithms were the final piece of the puzzle that allowed machines to crack the code of natural language.

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 a *B* ?"

...

"Is the first letter a *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 a *B* ?"

...

Despite all the "no" answers, or zeros, 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 is 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 pieces 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 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 or 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.

1.7 Word order and grammar

The order of words—grammar—matters. That's something that our BOW or word vector discarded in the earlier examples. Fortunately, in most short phrases, and even many complete sentences, this word vector approximation works fine. 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 likely 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 "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).prod()
479001600
```

The number of permutations exploded from factorial(3) == 6 in our simple greeting to factorial(12) == 479001600 in our longer statement! And it's clear that the logic contained in the order of the words is important to any machine that would like to reply with the correct response. Even though common greetings are not usually garbled by BOW processing, more complex statements can lose most of their meaning when thrown into a bag. A BOW 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 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%).[56] 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 NLP are discussed in the next section. Chapter 2 shows you a trick for incorporating some of the information conveyed by word order into our word vector representation. It also shows you how to refine the crude tokenizer used in the previous examples (str.split()) to more accurately bin words into more appropriate slots within the word vector so that strings like *good* and *Good* are assigned the same bin, and separate bins can be allocated for tokens like rosa and Rosa but not *Rosa!*.

1.8 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*.[57] However, some of the algorithms listed within the five subsystem blocks may be new to you. In fact, some of the most promising generative approaches, such as LLMs, have only recently been invented. Each chapter will help you implement and test one or more of the algorithms in this diagram so that you can assemble the right pipeline for your application. The four rounded blocks in figure 1.7 show the four kinds of processing required to analyze and generate text.

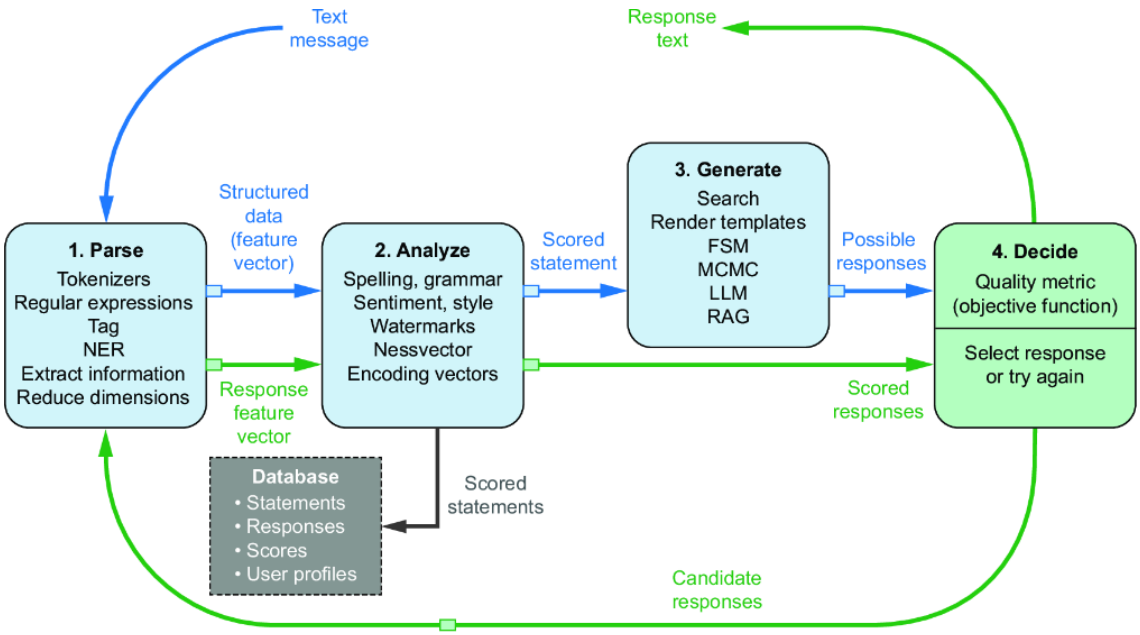

Figure 1.7 Chatbot recirculating (recurrent) pipeline

You will need a database to maintain a memory of past statements and responses as well as build a structured knowledge base to help inform the business logic and decision logic of your pipeline. Each of the five subsystems shown in figure 1.7 can be implemented with one or more algorithms:

1. *Parse* —Extract features, structured numerical data, from natural language text.
2. *Analyze* —Generate and combine features by scoring text for sentiment, grammaticality, and semantics.
3. *Generate* —Compose possible responses using templates, search, and language models.
4. *Decide* —Decide which generated response is most likely to move the conversation closer to the user's conversational goal.
5. *Database* —Store conversation history, user information, and general world knowledge for use in the Decide subsystem.

Each of these four stages can be implemented using one or more of the algorithms listed within the corresponding boxes in the block diagram. And you can combine the Python examples from this book in your own NLP system to accomplish state-of-the-art performance for most applications. By the end of this book, you will have mastered several alternative approaches to implementing these five subsystems. Most chatbots will contain elements of all five of these subsystems, but many applications require only simple algorithms that can be implemented in a few lines of Python. Some chatbots are better at answering factual questions, while others are better at information retrieval or search. And some chatbots can even generate lengthy, complex, plausibly human-sounding responses. As you might suspect, plausible responses are not necessarily correct or helpful. You will learn the advantages and disadvantages of all of the most popular approaches to NLP.

Machine learning, deep learning, and probabilistic language models have rapidly broadened the range of applications in which you can apply NLP successfully. The data-

driven approach of machine learning 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. Nonetheless, data isn't all you need. With a more efficient machine learning approach, you can often leapfrog over the competition. This book will give you the understanding you need to take advantage of these advances.

The chatbot pipeline in figure 1.7 contains all of the building blocks for most of the NLP applications described at the start of this chapter. We have also 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 1.7 that is not typically employed in search, forecasting, or question-answering systems is natural language *generation*. This the central feature of chatbots. 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 (e.g., 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 add sophistication and capability to each stage of the NLP pipeline.

1.9 Processing in depth

The stages of an NLP 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 more 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.

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, part-of-speech (POS) tagging is one way to generate features within the analysis stage of our chatbot pipeline. POS tags are generated automatically by the default spaCy pipeline, which

includes the top four layers in this diagram. POS tagging is typically accomplished with an FST, like the methods in the nltk.tag package.

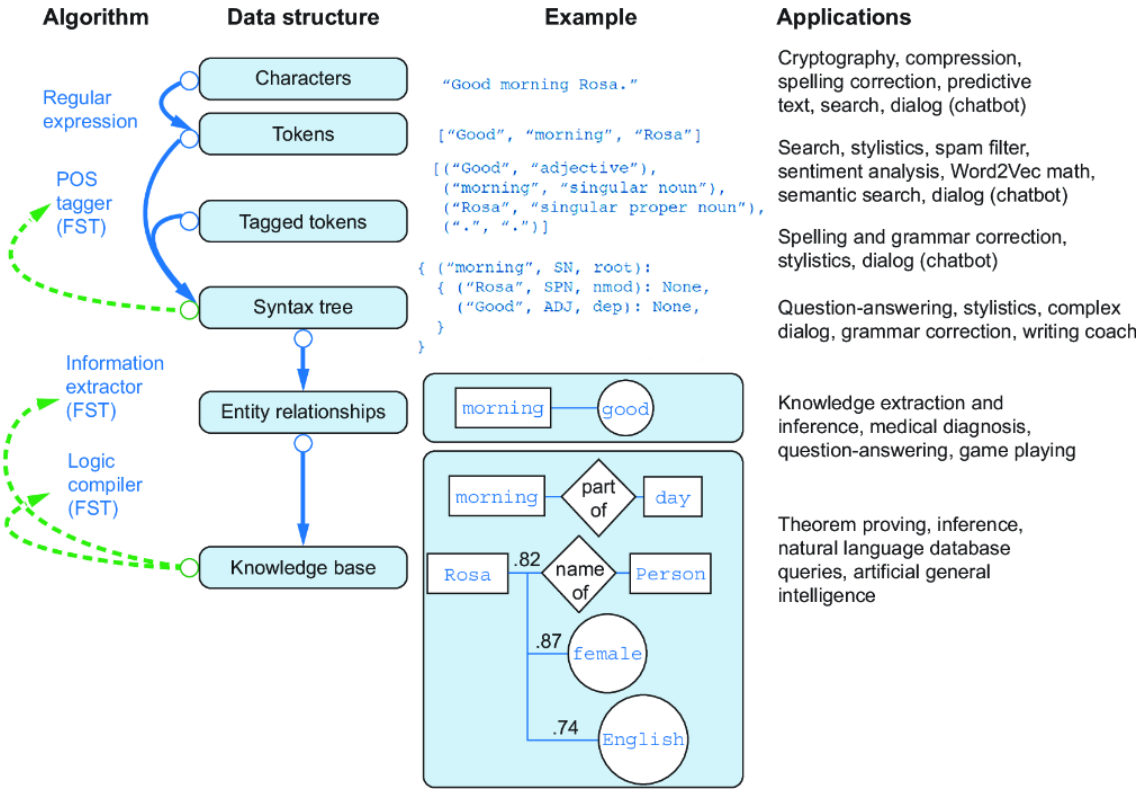

Figure 1.8 Example layers for an NLP pipeline

The bottom two layers (*entity relationships* and *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 is considered the domain of AI, 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.

In the next few chapters, we will 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 as well as to build human-mimicking chatbots. In fact, it's possible to build a useful and interesting chatbot with only a single layer of processing, directly using the text (character sequences) 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, 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 conversation as the dialog engine for Salvius, a mechanical robot built from salvaged parts by Gunther Cox.[58]

Will is an open source Python chatbot framework by Steven Skoczen with a completely different approach.[59] Will can only be trained to respond to statements by programming it with regular expressions—a 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"[60] 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 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 (e.g., Google, Amazon, Apple, and 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).[61] We show you how to do this in chapter 7.

1.10 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 behavior complexity and the degree of human supervision required. But for an NLP 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 Amazon Alexa or Google 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.[62] Google Dialogflow (which was developed independently of Google's Allo and Google Assistant) has capabilities similar to Amazon Lex, Amazon Contact Flow,[63] 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 1.9. Another approach rapidly overtaking this hardcoded pipeline is the deep learning

data-driven approach. Feature extractors for deep neural networks are learned rather than hardcoded, 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 NLP 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 1.9 puts the chatbot in its place among the NLP systems that are already out there. Imagine the chatbots you have interacted with. Where do you think they might fit in a plot like this? Have you attempted to gauge their intelligence by probing them with difficult questions or something like an IQ test? Try asking a chatbot something ambiguous that requires commonsense logic and the ability to ask clarifying questions, such as, "What's larger, the sun or a nickel?"[64] 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.

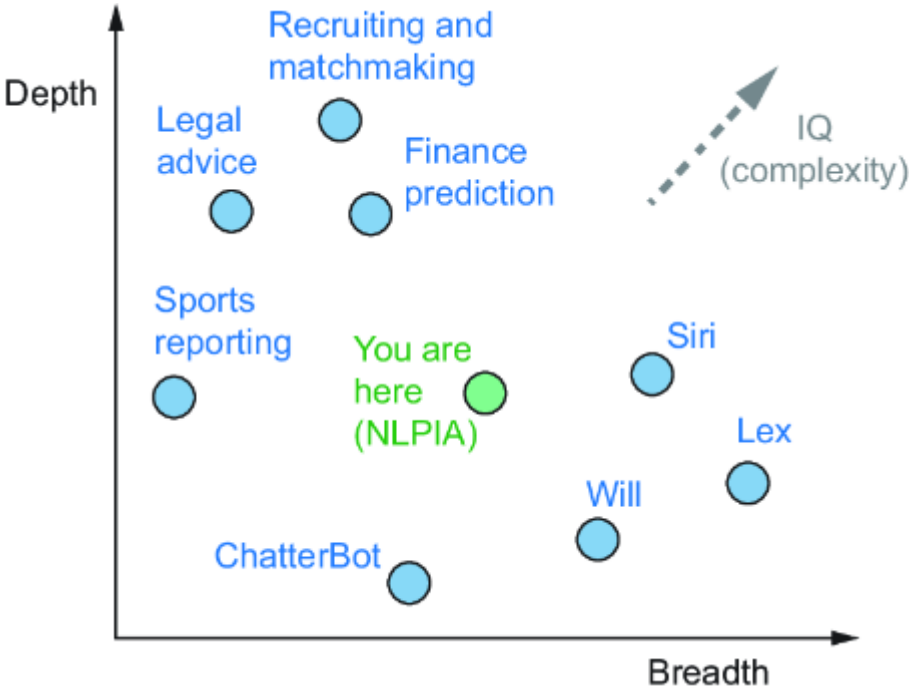

Figure 1.9 IQ of NLP systems

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 be able to retrieve relevant knowledge from its document database.
- *NLG* —To compose new, meaningful statements

Machine learning gives you a shortcut to quickly build machines that behave as if a team of programmers had spent years programming them with hundreds of complex hardcoded

algorithms and decision tree branches. With a modern NLP pipeline, you can teach a machine to respond effectively to patterns in text without entering regular expression hell; all you need are examples of user statements and the responses you want your chatbot to imitate. Machine learning is much less picky about *mispelings* and *typoz* than any handcrafted expert system or regular expression, and the *models* of language produced by machine learning are much better, more general, and more statistically accurate.

Machine learning NLP pipelines are easier to "program." You no longer have to anticipate every possible use of symbols in every language your users speak; you just have to feed the training pipeline with labeled examples. And the quality of your chatbot responses will depend mainly on the quality of the labeled dataset, making NLP accessible to more people on your team.

NLP is revolutionizing the way we communicate, learn, do business, and even think. We are witnessing machine-generated content encroach on more and more of the collective intelligence of society right before our eyes. You too will soon be building, training, and tweaking NLP systems that simulate human-like conversational behavior. In upcoming chapters, you will learn 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, you can train a machine to interact with that content.

This book is about using machine learning to build smart text-reading machines without requiring you to anticipate all the ways people can say things. Each chapter incrementally improves on the basic NLP pipeline for the chatbot architecture introduced in figure 1.7 and the inside cover of this book. As you learn the tools of NLP, 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.

1.11 Test yourself

1. Why is NLP considered a core enabling feature for AGI (human-like AI)?
2. How would you build a prosocial chatbot if much of your training data included antisocial examples?
3. What are the five main subsystems in a chatbot?
4. How is NLP used within a search engine?
5. Write a regular expression to recognize your name and all the variations on its spelling (including nicknames) you've seen.
6. Write a regular expression to try to recognize a sentence boundary (usually a period, question mark, or exclamation mark).

Tip Active learning (e.g., by quizzing yourself with questions like the ones in this subsection) can accelerate your understanding of any new topic.[65] It turns out this approach is effective for machine learning as well as model evaluation.[66]

Summary

- By building prosocial NLP software, you can help make the world a better place.
- The meaning and intent of words can be deciphered by machines.

- A smart NLP pipeline can deal with ambiguity and help correct human mistakes.
- Machines can build a knowledge base of information about the world by processing only unlabeled text.
- Chatbots can be thought of as semantic search engines, retrieving the most relevant response from the documents used for training.
- Regular expressions are useful for more than just search.