Natural Language Processing IN ACTION

Understanding, analyzing, and generating text with Python

Hobson Lane
Cole Howard
Hannes Max Hapke

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Natural Language Processing in Action
Understanding, analyzing, and generating text with Python
Version 10

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

welcome

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

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

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

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

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

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

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

—Hobson, Cole, and Hannes

brief contents

acknowledgments

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

I Wordy machines

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

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

In chapter 2, we give you the tools you need to teach machines to extract words from documents. There's more to it than you might guess, and we show you all the tricks. You'll learn how to automatically group natural language words together into groups of words with similar meaning without having to hand-craft synonym lists.

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

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

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

1 *Packets of thought (NLP overview)*

This chapter covers

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

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

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

1.1 Natural language vs. programming language

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

DEFINITION **Natural language processing**

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

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

DEFINITION **Pipeline**

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

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

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

-- Dave Magee Georgia Tech 1995

1.2 The magic

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

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

Footnote 2 See the web page titled "How Google's Site Crawlers Index Your Site - Google Search" (www.google.com/search/howsearchworks/crawling-indexing/).

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

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

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

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

Python was designed from the ground up to be a readable language. It also exposes a lot

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

Footnote 4 Mathematical notation is ambiguous. See the "Mathematical notation" section of the Wikipedia article "Ambguity" (en.wikipedia.org/wiki/Ambiguity#Mathematical_notation).

1.2.1 Machines that converse

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

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

SIDEBAR

Speech recognition systems

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

1.2.2 The math

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

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

Footnote 5 Gonzalo-Ibanez et al found that educated and trained human judges could not match the performance of their simple classification algorithm of 68% reported in their ACM paper. The Sarcasm Detector (github.com/MathieuCliche/Sarcasm_detector) and web app (www.thesarcasmdetector.com/) by Matthew Cliche at Cornell achieves similar accuracy (>70%).

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

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

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

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

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

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

1.3 Practical applications

Natural language processing is everywhere. It's so ubiquitous that some of the examples in table 1.1 may surprise you.

Table 1.1 Categorized NLP applications

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

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

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

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

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

Footnote 8 New York Times, Oct 18, 2016, automated-pro-trump-bots-overwhelmed-pro-clinton-messages-researchers-say and MIT Technology Review, Nov 2016, how-the-bot-y-politic-influenced-this-election/

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

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

Footnote 9 Google Blog May 2018 about their *Duplex* system advances-in-semantic-textual-similarity

NOTE With its *Duplex* demonstration at Google IO, engineers and managers overlooked concerns about the ethics of teaching chatbots to deceive humans. We all ignore this dilemma when we happily interact with chatbots on Twitter and other anonymous social networks, where bots do not share their pedigree. With bots that can so convincingly deceive us, the AI control problem [10] looms, and Yuval Harari's cautionary forecast of "Homo Deus"[11] may come sooner than we think.

Footnote 10 See the web page titled "AI control problem - Wikipedia" (en.wikipedia.org/wiki/AI_control_problem).

Footnote 11 WSJ Blog, March 10, 2017 authority-shifting-from-people-to-ai

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

1.4 Language through a computer's "eyes"

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

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

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

1.4.1 The language of locks

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

After finishing this section, you'll never think of your combination bicycle lock the same way again. A combination lock certainly can't read and understand the textbooks stored inside a school locker, but it can understand the language of locks. It can understand when you try to "tell" it a "password": a combination. A padlock combination is any sequence of symbols that matches the "grammar" (pattern) of lock language. Even more

importantly, the padlock can tell if a lock "statement" matches a particularly meaningful statement, the one for which there is only one correct "response," to release the catch holding the U-shaped hasp so you can get into your locker.

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

For example, we'd like our chatbot to recognize greetings such as "Hello Rosa," and respond to them appropriately. This kind of language, like the language of locks, is a formal language because it has strict rules about how an acceptable statement must be composed and interpreted. If you've ever written a math equation or coded a programming language expression, you've written a formal language statement.

Formal languages are a subset of natural languages. Many natural language statements can be matched or generated using a formal language grammar, like regular expressions. That's the reason for this diversion into the mechanical, "click, whirr"[12] language of locks.

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

1.4.2 Regular expressions

Regular expressions use a special kind (class) of formal language grammar called a regular grammar. Regular grammars have predictable, provable behavior, and yet are flexible enough to power some of the most sophisticated dialog engines and chatbots on the market. Amazon Alexa and Google Now are mostly pattern-based engines that rely on regular grammars. Deep, complex regular grammar rules can often be expressed in a single line of code called a regular expression. There are successful chatbot frameworks in Python, like `Will`, that rely exclusively on this kind of language to produce some useful and interesting behavior. Amazon Echo, Google Home, and similarly complex and useful assistants use this kind of language to encode the logic for most of their user interaction.

NOTE Regular expressions implemented in Python and in Posix (Unix) applications such as `grep` are not true regular grammars. They have language and logic features such as look-ahead and look-back that make leaps of logic and recursion that aren't allowed in a regular grammar. As a result, regular expressions aren't provably halting; they can sometimes "crash" or run forever.[13]

Footnote 13 Stack Exchange Went Down for 30 minutes on July 20, 2016 when a regex "crashed" (stackstatus.net/post/147710624694/outage-postmortem-july-20-2016)

You may be saying to yourself, "I've heard of regular expressions. I use `grep`. But that's only for search!" And you're right. *R*egular *E*xpressions are indeed used mostly for search, for sequence matching. But anything that can find matches within text is also great for carrying out a dialog. Some chatbots, like `Will`, use "search" to find sequences of characters within a user statement that they know how to respond to. These recognized sequences then trigger a scripted response appropriate to that particular regular expression match. And that same regular expression can also be used to extract a useful piece of information from a statement. A chatbot can add that bit of information to its knowledge base about the user or about the world the user is describing.

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

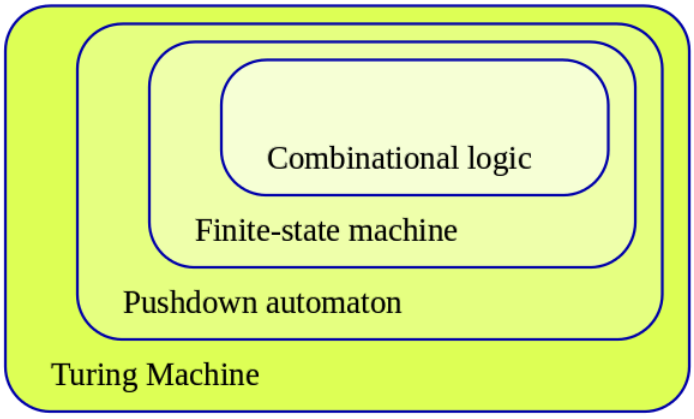

Figure 1.1 Kinds of automata

SIDEBAR

Formal mathematical explanation of formal languages

Kyle Gorman describes programming languages this way:

- Most (if not all) programming languages are drawn from the class of context-free languages.
- Context free languages are parsed with context-free grammars, which provide efficient parsing.
- The regular languages are also efficiently parsable and used extensively in computing for string matching.
- String matching applications rarely require the expressiveness of context-free.
- There are a number of formal language classes, a few of which are shown here (in decreasing complexity):[14]

  Footnote 14 See the web page titled "Chomsky hierarchy - Wikipedia" (en.wikipedia.org/wiki/Chomsky_hierarchy).

  - Recursively enumerable
  - Context-sensitive
  - Context-free
  - Regular

Natural languages are:

- Not regular [15]

  Footnote 15 "English is not a regular language" (cs.haifa.ac.il/../complexity.pdf#page=20) by Shuly Wintner

- Not context-free [16]

  Footnote 16 "Is English context-free?" (cs.haifa.ac.il/../complexity.pdf#page=24) by Shuly Wintner

- Can’t be defined by any formal grammar [17]

  Footnote 17 See the web page titled "1.11. Formal and Natural Languages — How to Think like a Computer Scientist: Interactive Edition" (interactivepython.org/../FormalandNaturalLanguages).

1.4.3 A simple chatbot

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

So if you run this little script and chat to our bot with a phrase like "Hello Rosa", it will respond by asking about your day. If you use a slightly rude name to address the chatbot,

she will be less responsive, but not inflammatory, to try to encourage politeness.[18] If you name someone else who might be monitoring the conversation on a party line or forum, the bot will keep quiet and allow you and whomever you are addressing to chat. Obviously there's no one else out there watching our `input()` line, but if this were a function within a larger chatbot, you want to deal with these sorts of things.

Footnote 18 The idea for this defusing response originated with Viktor Frankl's *Man's Search for Meaning*, his Logotherapy (en.wikipedia.org/wiki/Logotherapy) approach to psychology and the many popular novels where a child protagonist like Owen Meany has the wisdom to respond to an insult with a response like this.

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

TIP

Classical computational theory of mind

This classical NLP pattern-matching approach is based on the computational theory of mind (CTM). CTM assumes that human-like NLP can be accomplished with a finite set of logical rules that are processed in series.[19] Advancements in neuroscience and NLP led to the development of a "connectionist" theory of mind around the turn of the century, which allows for parallel pipelines processing natural language simultaneously, as is done in artificial neural networks.[20] [21]

Footnote 19 Stanford Encyclopedia of Philosophy, Computational Theory of Mind, plato.stanford.edu/entries/computational-mind/

Footnote 20 Stanford Encyclopedia of Philosophy, Connectionism, plato.stanford.edu/entries/connectionism/

Footnote 21 Christiansen and Chater, 1999, Southern Illinois University, crl.ucsd.edu/\~elman/Bulgaria/christiansen-chater-soa

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

datasets, to shortcut this laborious hand programming and refining.

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

1.4.4 Another way

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

But imagine how a single typo or variation in the statement would trip up our bot. Bit and character sequences are discrete. They either match or they don't. Instead, we'd like our bot to be able to measure the difference in *meaning* between character sequences.

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

Distance metrics designed for numerical sequences and vectors are useful for a few NLP applications, like spelling correctors and recognizing proper nouns. So we use these distance metrics when they make sense. But for NLP applications where we are more interested in the meaning of the natural language than its spelling, there are better

approaches. We use vector representations of natural language words and text and some distance metrics for those vectors for those NLP applications. We show you each approach, one by one, as we talk about these different applications and the kinds of vectors they are used with.

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

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

But let's think for a moment about what information has been lost in our effort to count all the words in the messages we receive. We assign the words to bins and store them away as bit vectors like a coin or token sorter (see figure 1.2) directing different kinds of tokens to one side or the other in a cascade of decisions that piles them in bins at the bottom. Our sorting machine must take into account hundreds of thousands if not millions of possible token "denominations," one for each possible word that a speaker or

author might use. Each phrase or sentence or document we feed into our token sorting machine will come out the bottom, where we have a "vector" with a count of the tokens in each slot. Most of our counts are zero, even for large documents with verbose vocabulary. But we haven’t lost any words yet. What have we lost? Could you, as a human understand a document that we presented you in this way, as a count of each possible word in your language, without any sequence or order associated with those words? I doubt it. But if it was a short sentence or tweet, you’d probably be able to rearrange them into their intended order and meaning most of the time.

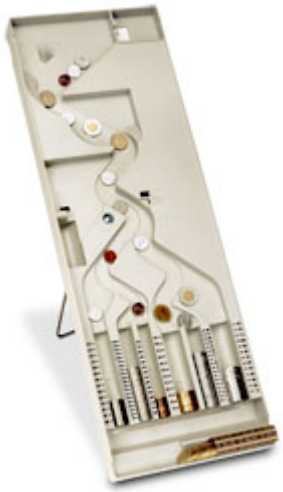

Figure 1.2 Canadian coin sorter

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

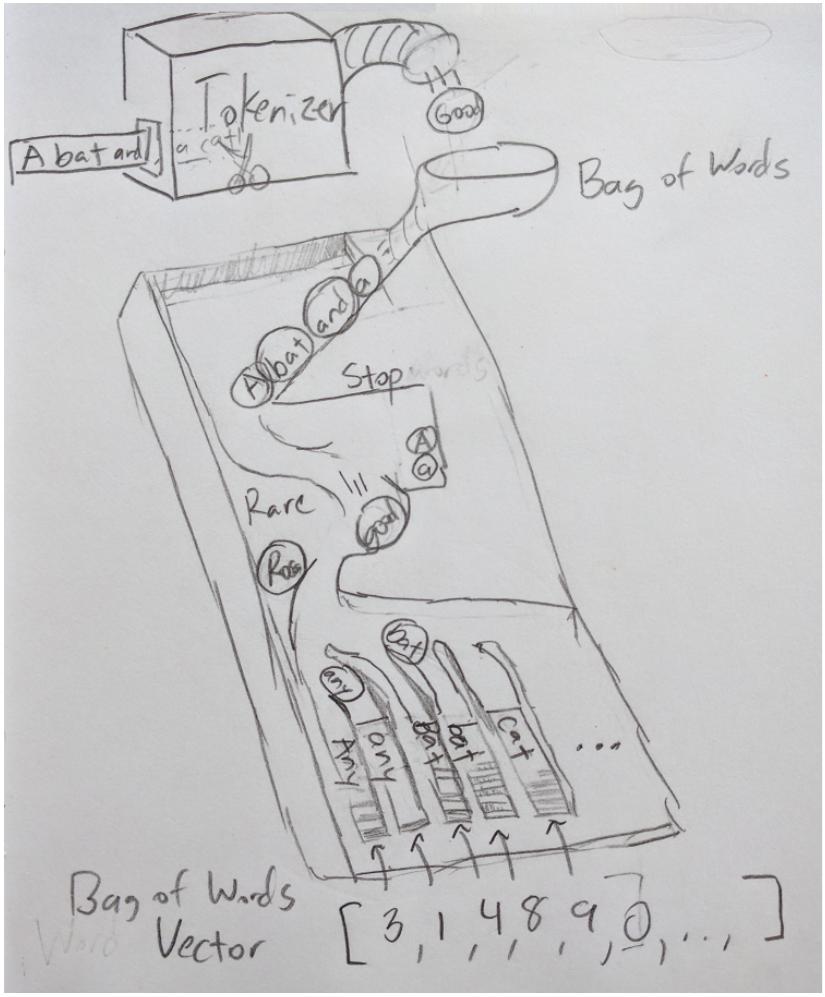

Figure 1.3 Token sorting tray

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

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

```
>>> from collections import Counter

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

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

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

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

1.5 A brief overflight of hyperspace

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

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

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

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

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

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

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

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

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

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

1.6 Word order and grammar

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

```
>>> from itertools import permutations

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

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

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

```
>>> s = """Find textbooks with titles containing 'NLP',
...     or 'natural' and 'language', or
...     'computational' and  '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 bag-of-words processing, more complex statements can lose most of their meaning when thrown into a bag. A bag of words is not the best way to begin processing a database query, like the natural language query in the preceding example.

Whether a statement is written in a formal programming language like SQL, or in an informal natural language like English, word order and grammar are important when a statement intends to convey logical relationships between things. That's why computer languages depend on rigid grammar and syntax rule parsers. Fortunately, recent advances in natural language syntax tree parsers have made possible the extraction of syntactical and logical relationships from natural language with remarkable accuracy (greater than 90%).[22] In later chapters, we show you how to use packages like `SyntaxNet` (Parsey McParseface) and `SpaCy` to identify these relationships.

Footnote 22 A comparison of the syntax parsing accuracy of SpaCy (93%), SyntaxNet (94%), Stanford's CoreNLP (90%), and others is available at spacy.io/docs/api/

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

1.7 A chatbot natural language pipeline

The NLP pipeline required to build a dialog engine, or chatbot, is similar to the pipeline required to build a question answering system described in *Taming Text* (Manning, 2013).[23] However, some of the algorithms listed within the five subsystem blocks may be new to you. We help you implement these in Python to accomplish various NLP tasks essential for most applications, including chatbots.

Footnote 23 Ingersol, Morton, and Farris, www.manning.com/books/taming-text/?a_aid=totalgood

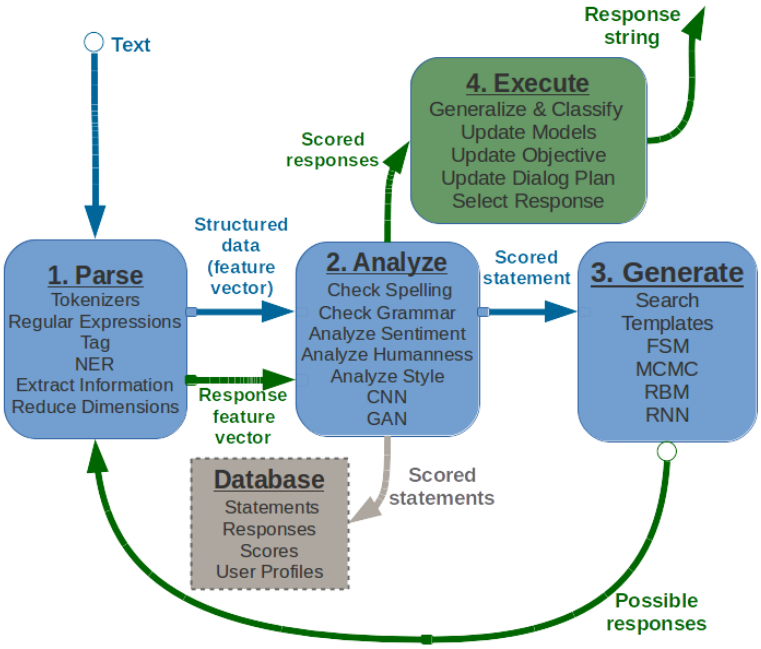

Figure 1.4 Chatbot recirculating (recurrent) pipeline

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

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

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

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

In addition, deep learning and data-driven programming (machine learning, or probabilistic language modeling) have rapidly diversified the possible applications for NLP and chatbots. This data-driven approach allows ever greater sophistication for an

NLP pipeline by providing it with greater and greater amounts of data in the domain you want to apply it to. And when a new machine learning approach is discovered that makes even better use of this data, with more efficient model generalization or regularization, then large jumps in capability are possible.

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

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

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

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

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

1.8 Processing in depth

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

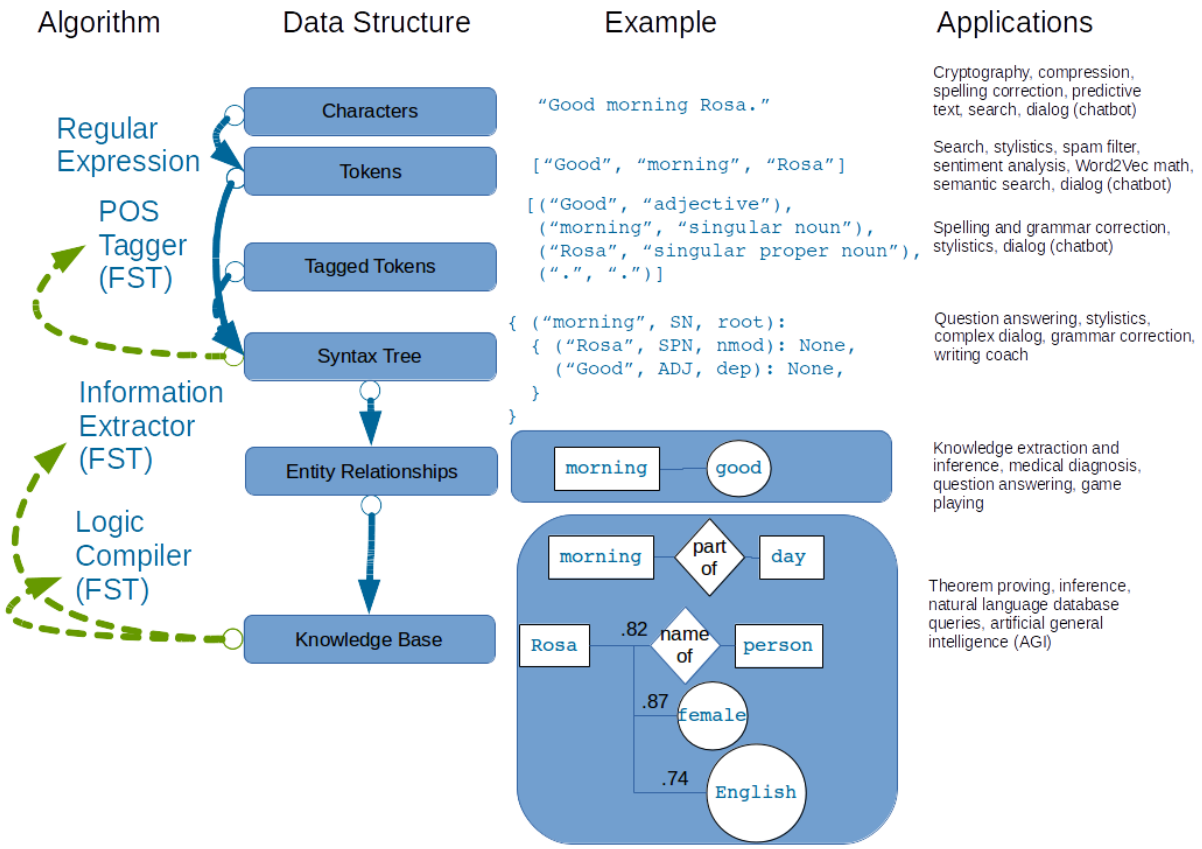

Figure 1.5 Example layers for an NLP pipeline

The top four layers in figure 1.5 correspond to the first two stages in the chatbot pipeline (feature extraction and feature analysis) in the previous section. For example the part-of-speech tagging (POS tagging), is one way to generate features within the Analyze stage of our chatbot pipeline. POS tags are generated automatically by the default `spaCy` pipeline, which includes all the top four layers in this diagram. POS tagging is typically accomplished with a finite state transducer like the methods in the `nltk.tag` package.

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

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

For example, the open source project `ChatterBot` simplifies this pipeline by merely computing the string "edit distance" (Levenshtein distance) between an input statement and the statements recorded in its database. If its database of statement-response pairs contains a matching statement, the corresponding reply (from a previously "learned" human or machine dialog) can be reused as the reply to the latest user statement. For this pipeline, all that is required is step 3 (Generate) of our chatbot pipeline. And within this stage, only a brute force search algorithm is required to find the best response. With this simple technique (no tokenization or feature generation required), `ChatterBot` can maintain a convincing conversion as the dialog engine for Salvius, a mechanical robot built from salvaged parts by Gunther Cox.[24]

Footnote 24 ChatterBot by Gunther Cox and others at github.com/gunthercox/ChatterBot

`Will` is an open source Python chatbot framework by Steven Skoczen with a completely different approach.[25]. `Will` can only be trained to respond to statements by programming it with regular expressions. This is the labor-intensive and data-light approach to NLP. This grammar-based approach is especially effective for question answering systems and task-execution assistant bots, like Lex, Siri, and Google Now. These kinds of systems

overcome the "brittleness" of regular expressions by employing "fuzzy regular expressions"[26] and other techniques for finding approximate grammar matches. Fuzzy regular expressions find the closest grammar matches among a list of possible grammar rules (regular expressions) instead of exact matches by ignoring some maximum number of insertion, deletion, and substitution errors. However, expanding the breadth and complexity of behaviors for a grammar-based chatbot requires a lot of human development work. Even the most advanced grammar-based chatbots, built and maintained by some of the largest corporations on the planet (Google, Amazon, Apple, Microsoft), remain in the middle of the pack for depth and breadth of chatbot IQ.

Footnote 25 See the GitHub page for "Will", a chatbot for HipChat by Steven Skoczen and the HipChat community (github.com/skoczen/will). In 2018 it was updated to integrate with Slack.

Footnote 26 The Python `regex` package is backward compatible with `re` and adds fuzziness among other features. It will replace the `re` package in the future (pypi.python.org/pypi/regex). Similarly `TRE agrep`, or "approximate grep", (github.com/laurikari/tre) is an alternative to the UNIX command-line application `grep`.

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).[27] We show you how to do this in chapter 6.

Footnote 27 Simple neural networks are often used for unsupervised feature extraction from character and word sequences.

1.9 Natural language IQ

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

A consumer product chatbot or virtual assistant like Alexa or Allo is usually designed to have extremely broad knowledge and capabilities. However, the logic used to respond to requests tends to be shallow, often consisting of a set of trigger phrases that all produce the same response with a single if-then decision branch. Alexa (and the underlying Lex engine) behave like a single layer, flat tree of (if, elif, elif, …) statements.[28] Google

Dialogflow (which was developed independently of Google's Allo and Google Assistant) has similar capability to Amazon Lex, Contact Flow, and Lambda, but without the drag-and-drop user interface for designing your dialog tree.

Footnote 28 More complicated logic and behaviors are now possible when you incorporate Lambdas into an AWS Contact Flow dialog tree. See "Creating Call Center Bot with AWS Connect" (creating-call-center-bot-aws-connect-amazon-lex-can-speak-understand).

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

You will use both approaches (neural networks and hand-coded algorithms) as you incrementally build an NLP pipeline for a chatbot capable of conversing within a focused knowledge domain. This will give you the skills you need to accomplish the natural language processing tasks within your industry or business domain. Along the way you'll probably get ideas about how to expand the breadth of things this NLP pipeline can do. Figure 1.6 puts the chatbot in its place among the natural language processing systems that are already out there. Imagine the chatbots you have interacted with. Where do you think they might fit on a plot like this? Have you attempted to gauge their intelligence by probing them with difficult questions or something like an IQ test?[29] You'll get a chance to do exactly that in later chapters, to help you decide how your chatbot stacks up against some of the others in this diagram.

Footnote 29 A good question suggested by Byron Reese is: "What's larger? The sun or a nickel?" (gigaom.com/2017/11/20/voices-in-ai-episode-20-a-conversation-with-marie-des-jardins), Here are a couple more (github.com/../iq_test.csv) to get you started.

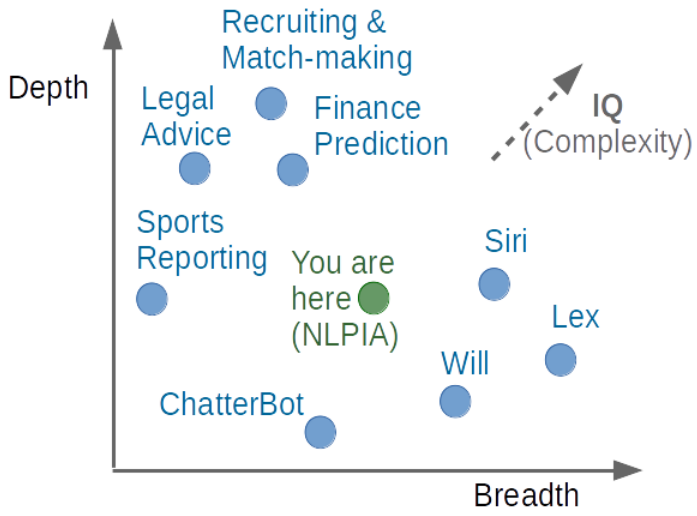

Figure 1.6 2D IQ of some natural language processing systems

As you progress through this book, you'll be building the elements of a chatbot. Chatbots require all the tools of NLP to work well:

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

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

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

We've given you some exciting reasons to learn about natural language processing. You want to help save the world, don't you? And we've attempted to pique your interest with some practical NLP applications that are revolutionizing the way we communicate, learn, do business, and even think. It won't be long before you're able to build a system that approaches human-like conversational behavior. And you should be able to see in

upcoming chapters how to train a chatbot or NLP pipeline with any domain knowledge that interests you—from finance and sports to psychology and literature. If you can find a corpus of writing about it, then you can train a machine to understand it.

The rest of this book is about using machine learning to save us from having to anticipate all the ways people can say things in natural language. Each chapter incrementally improves on the basic NLP pipeline for the chatbot introduced in this chapter. As you learn the tools of natural language processing, you'll be building an NLP pipeline that can not only carry on a conversation but help you accomplish your goals in business and in life.

1.10 Summary

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