Natural Language Processing IN ACTION

SECOND EDITION

Hobson Lane
Maria Dyshel

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Natural Language Processing in Action
Second Edition

Version 8

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

brief contents

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):[43]
  - Recursively enumerable
  - Context-sensitive
  - Context-free
  - Regular

Natural languages are:

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

1.5 A simple chatbot

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

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

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

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

1.6 Keyword-based greeting recognizer

Your first chatbot will be straight out of the 80's. Imagine you want a chatbot to help you select a game to play, like chess… or a Thermonuclear War. But first your chatbot must find out if you are professor Falken or General Beringer, or some other user that knows what they are doing.[47] It will only be able to recognize a few greetings. But this approach can be extended to help you implement simple keyword-based intent recognizers on projects similar to those mentioned earlier in this chapter.

Listing 1.1 Keyword detection using `str.split`

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

❶ "Hi", "Hello", and "Greetings" might be the keywords programmed into Joshua, running on a supercomputer called "WOPR" in *War Games*.

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

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

1.6.1 Pattern-based intent recognition

A keyword based chatbot would recognize "Hi", "Hello", and "Greetings", but it wouldn't recognize "Hiiii" or "Hiiiiiiiiiii" - the more excited renditions of "Hi". Perhaps you could hardcode the first 200 versions of "Hi", such as `["Hi", "Hii", "Hiii", …]`. Or you could programmatically create such a list of keywords. Or you could save yourself a lot of trouble and make your bot deal with literally infinite variations of "Hi" using *regular expressions*. Regular expression *patterns* can match text much more robustly than any hard-coded rules or lists of keywords.

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

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

```
>>> import re                                                    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>
```

❶ There are two "official" regular expression packages in Python. The `re` package is pre-installed with all versions of Python. The `regex` package includes additional features such as fuzzy pattern matching.

❷ `'|'` means "OR", '*' means the preceding characters can occur 0 or more times and still match.

❸ 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 them all out individually. So the regular expression `"[a-z]"` will match any single lowercase letter, "a" through "z". The star ("*") after a character class means that the regular expression will match any number of consecutive characters if they are all within that character class.

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

```
>>> r = r"[^a-z]*([y]o|[h']?ello|ok|hey|(good[ ])(morn[gin']{0,3}|"
>>> r += r"afternoon|even[gin']{0,3}))[\s,;:]{1,3}([a-z]{1,20})")
>>> re_greeting = re.compile(r, flags=re.IGNORECASE)             ❶
>>> 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")                       ❷
>>> re_greeting.match('Good evening Rosa Parks').groups()        ❸
('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'>
```

❶ You can compile regular expressions so you do not have to specify the options (`flags`) each time you use it.

❷ Notice that this regular expression cannot recognize (match) words with typos.

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

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

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

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

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

(1) We do not yet know who is chatting with the bot, and we will not 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 encourage politeness.[49] If you name someone else who might be monitoring the conversation on a party line or forum, the bot will keep quiet and allow you and whomever you are addressing to chat. Obviously, there is no one else out there watching our `input()` line, but if this were a function within a larger chatbot, you want to deal with these sorts of things.

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

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

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

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

1.6.2 Another way

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

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

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

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

We do not stay in this confusing binary world of logic for long, but let's imagine we're famous World War II-era code-breaker Mavis Batey at Bletchley Park and we have just been handed that binary, Morse code message intercepted from communication between two German military officers. It could hold the key to winning the war. Where would we start? Well the first layer of deciding would be to do something statistical with that stream of bits to see if we can find

patterns. We can first use the Morse code table (or ASCII table, in our case) to assign letters to each group of bits. Then, if the characters are gibberish to us, as they are to a computer or a cryptographer in WWII, we could start counting them up, looking up the short sequences in a dictionary of all the words we have seen before and putting a mark next to the entry every time it occurs. We might also make a mark in some other log book to indicate which message the word occurred in, creating an encyclopedic index to all the documents we have read before. This collection of documents is called a *corpus*, and the words or sequences we have listed in our index are called a *lexicon*.

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

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

Figure 1.5 Canadian coin sorter

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

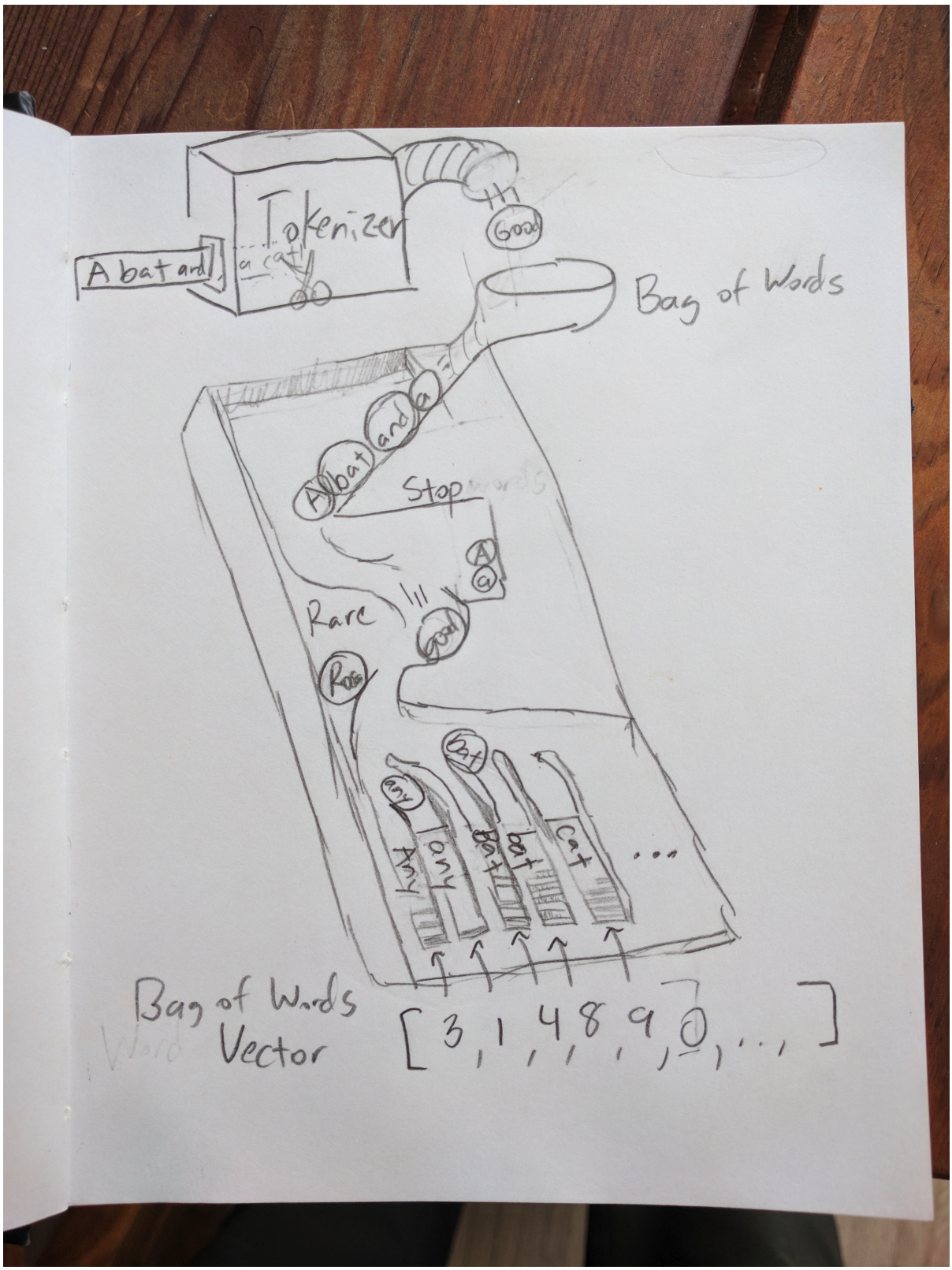

Figure 1.6 Token sorting tray

It turns out that machines can handle this bag of words quite well and glean most of the information content of even moderately long documents this way. Each document, after token sorting and counting, can be represented as a vector, a sequence of integers for each word or

token in that document. You see a crude example in figure 1.3, and then chapter 2 shows some more useful data structures for bag-of-word vectors.

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

```
>>> from collections import Counter

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

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

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

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

and some other properties that make high dimensional vectors difficult to work with.

1.7 A brief overflight of hyperspace

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

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

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

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

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

But how can a computer assign values to each of these vector dimensions? Well, if we simplified our vector dimension questions to things like, "Does it contain the word 'good'? Does it contain the word 'morning'?" And so on. You can see that we might be able to come up with a million or so questions resulting in numerical value assignments that a computer could make to a phrase.

This is the first practical vector space model, called a bit vector language model, or the sum of "one-hot encoded" vectors. You can see why computers are just now getting powerful enough to make sense of natural language. The millions of million-dimensional vectors that humans might generate simply "Does not compute!" on a supercomputer of the 80s, but is no problem on a commodity laptop in the 21st century. More than just raw hardware power and capacity made NLP practical; incremental, constant-RAM, linear algebra algorithms were the final piece of the puzzle that allowed machines to crack the code of natural language.

There is an even simpler, but much larger representation that can be used in a chatbot. What if our vector dimensions completely described the exact sequence of characters. The vector for each character would contain the answer to binary (yes/no) questions about every letter and punctuation mark in your alphabet:

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

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

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

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

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

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

meaning or topic vectors. We peek behind the magician's curtain in chapter 4 when we talk about latent semantic indexing and latent Dirichlet allocation, two techniques for creating much more dense and meaningful vector representations of statements and documents.

1.8 Word order and grammar

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

```
>>> from itertools import permutations

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

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

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

```
>>> s = """Find textbooks with titles containing 'NLP',
...     or 'natural' and 'language', or
...     'computational' and  'linguistics'."""
>>> len(set(s.split()))
12
>>> import numpy as np
>>> np.arange(1, 12 + 1).prod()  # factorial(12) = arange(1, 13).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%).[53] In later chapters, we show you how to use packages like `SyntaxNet` (Parsey McParseface) and `SpaCy` to identify these relationships.

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

1.9 A chatbot natural language pipeline

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

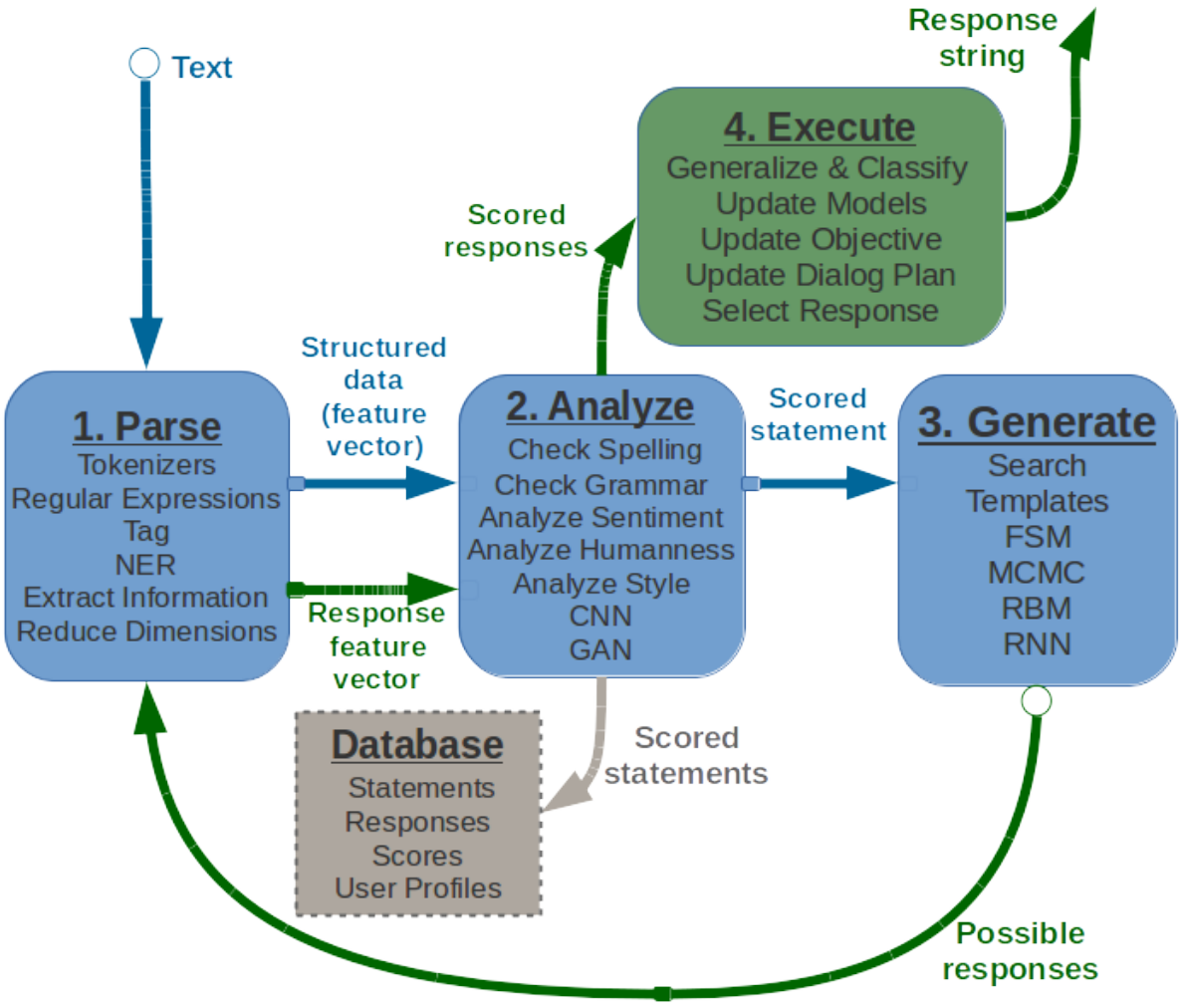

Figure 1.7 Chatbot recirculating (recurrent) pipeline

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

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

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

Most chatbots will contain elements of all five of these subsystems (the four processing stages as

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

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

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

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

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

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

1.10 Processing in depth

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

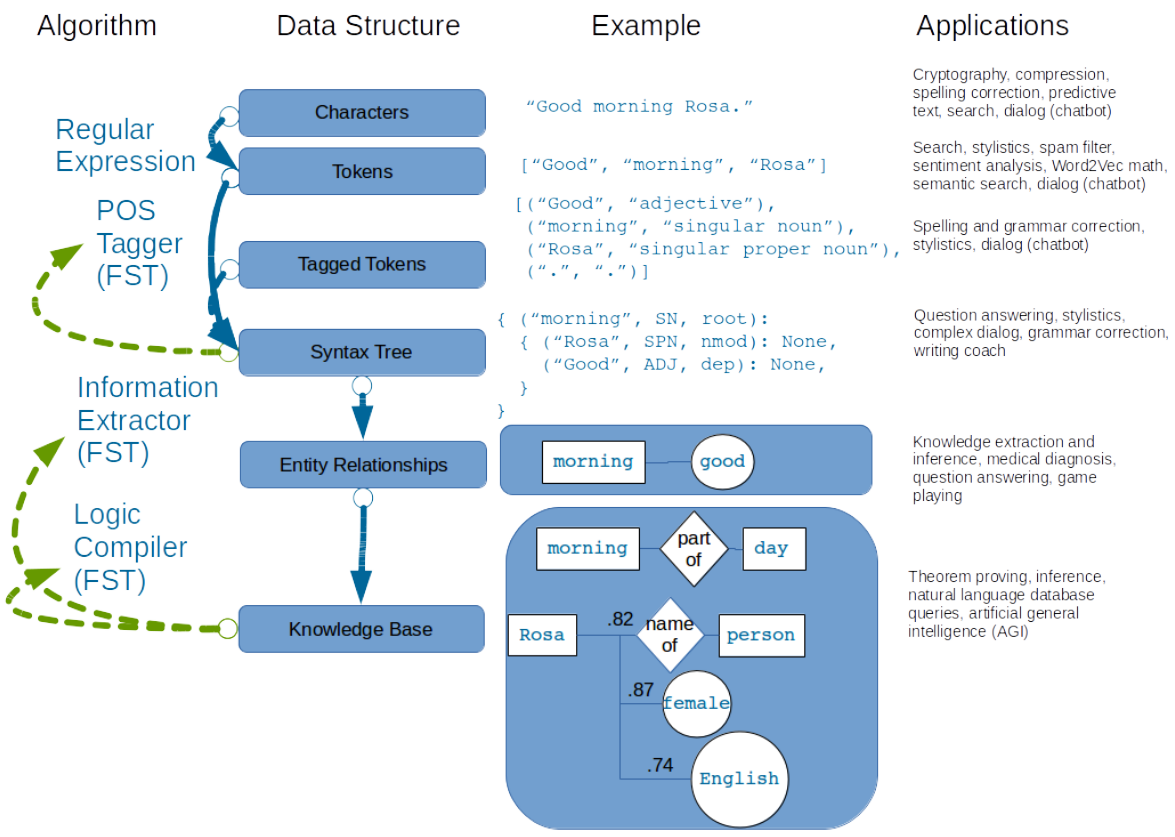

Figure 1.8 Example layers for an NLP pipeline

The top four layers in figure 1.8 correspond to the first two stages in the chatbot pipeline (feature extraction and feature analysis) in the previous section. For example the part-of-speech tagging (POS tagging), is one way to generate features within the Analyze stage of our chatbot pipeline. POS tags are generated automatically by the default `SpaCY` pipeline, which includes all the top four layers in this diagram. POS tagging is typically accomplished with a finite state transducer like the methods in the `nltk.tag` package.

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

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

For example, the open source project `ChatterBot` simplifies this pipeline by merely computing the string "edit distance" (Levenshtein distance) between an input statement and the statements recorded in its database. If its database of statement-response pairs contains a matching statement, the corresponding reply (from a previously "learned" human or machine dialog) can be reused as the reply to the latest user statement. For this pipeline, all that is required is step 3 (Generate) of our chatbot pipeline. And within this stage, only a brute force search algorithm is required to find the best response. With this simple technique (no tokenization or feature generation required), `ChatterBot` can maintain a convincing conversion as the dialog engine for Salvius, a mechanical robot built from salvaged parts by Gunther Cox.[55]

`Will` is an open source Python chatbot framework by Steven Skoczen with a completely different approach.[56] `Will` can only be trained to respond to statements by programming it with regular expressions. This is the labor-intensive and data-light approach to NLP. This grammar-based approach is especially effective for question answering systems and task-execution assistant bots, like Lex, Siri, and Google Now. These kinds of systems overcome the "brittleness" of regular expressions by employing "fuzzy regular expressions"footnote:[The Python `regex` package is backward compatible with `re` and adds fuzziness among other features. The `regex` will replace the `re` package in future python versions (https://pypi.python.org/pypi/regex).

Similarly `TRE agrep`, or "approximate grep," (https://github.com/laurikari/tre) is an alternative to the UNIX command-line application `grep`.] and other techniques for finding approximate

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

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

1.11 Natural language IQ

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

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

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

You will use both approaches (neural networks and hand-coded algorithms) as you incrementally build an NLP pipeline for a chatbot capable of conversing within a focused knowledge domain.

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

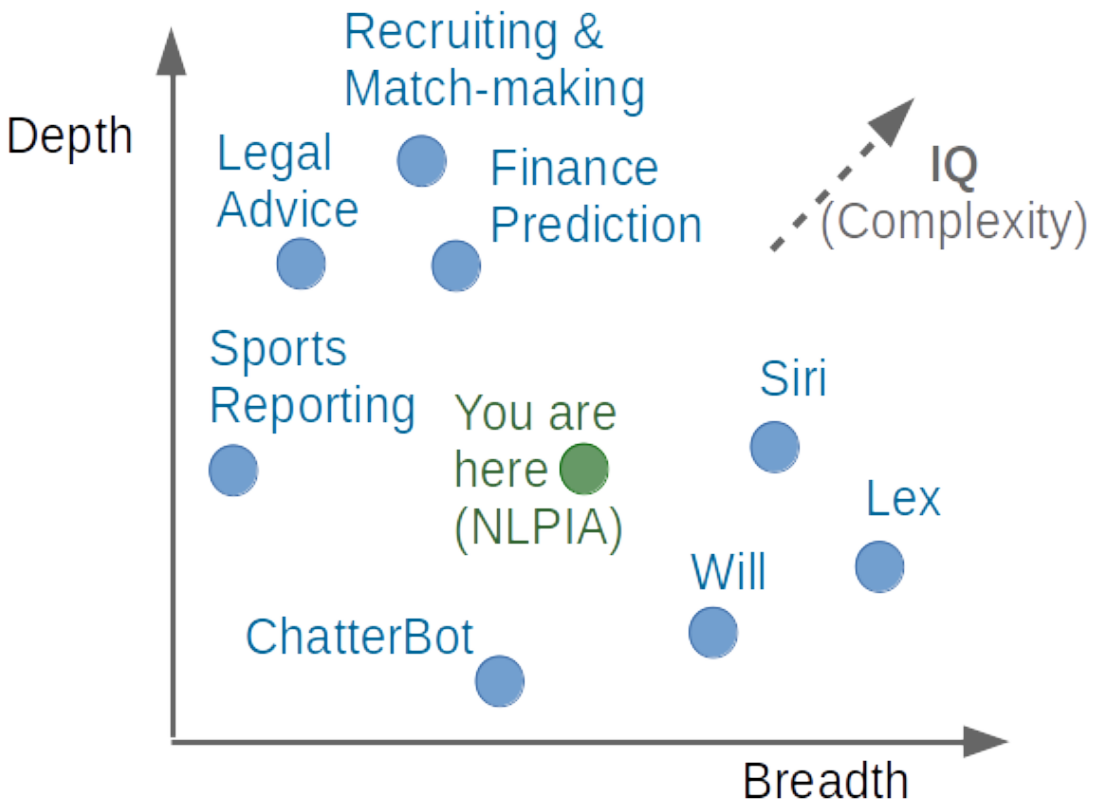

Figure 1.9 2D IQ of some natural language processing systems

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

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

Machine learning gives us a way to trick machines into behaving as if we had spent a lifetime programming them with hundreds of complex regular expressions or algorithms. We can teach a machine to respond to patterns similar to the patterns defined in regular expressions by merely

providing it examples of user statements and the responses we want the chatbot to mimic. And the "models" of language, the FSMs, produced by machine learning, are much better. They are less picky about mispelings and typoz.

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

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

This book is about using machine learning to build smart text reading machines without you having to anticipate all the ways people can say things. Each chapter incrementally improves on the basic NLP pipeline for the chatbot introduced in this chapter. As you learn the tools of natural language processing, you will be building an NLP pipeline that can not only carry on a conversation but help you accomplish your goals in business and in life.

1.12 Review

Chapter 1 review questions

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

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