2 Build your vocabulary (word tokenization)

This chapter covers

- Tokenizing your text into words and *n*-grams (tokens)
- Dealing with nonstandard punctuation and emoticons, like social media posts
- Compressing your token vocabulary with stemming and lemmatization
- Building a vector representation of a statement
- Building a sentiment analyzer from handcrafted token scores

So you're ready to save the world with the power of natural language processing? Well the first thing you need is a powerful vocabulary. This chapter will help you split a document, any string, into discrete tokens of meaning. Our tokens are limited to words, punctuation marks, and numbers, but the techniques we use are easily extended to any other units of meaning contained in a sequence of characters, like ASCII emoticons, Unicode emojis, mathematical symbols, and so on.

Retrieving tokens from a document will require some string manipulation beyond just the `str.split()` method employed in chapter 1. You'll need to separate punctuation from words, like quotes at the beginning and end of a statement. And you'll need to split contractions like "we'll" into the words that were combined to form them. Once you've identified the tokens in a document that you'd like to include in your vocabulary, you'll return to the regular expression toolbox to try to combine words with similar meaning in a process called *stemming*. Then you'll assemble a vector representation of your documents called a bag of words, and you'll try to use this vector to see if it can help you improve upon the greeting recognizer sketched out at the end of chapter 1.

Think for a moment about what a word or token represents to you. Does it represent a single concept, or some blurry cloud of concepts? Could you be sure you could always recognize a word? Are natural language words like programming language keywords that have precise definitions and a set of grammatical usage rules? Could you write software that could recognize a word? Is "ice cream" one word or two to you? Don't both words have entries in your mental dictionary that are separate from the compound word "ice cream"? What about the contraction "don't"? Should that string of characters be split into one or two "packets of meaning?"

And words could be divided even further into smaller packets of meaning. Words themselves can be divided up into smaller meaningful parts. Syllables, prefixes, and suffixes, like "re," "pre," and "ing" have intrinsic meaning. And parts of words can be divided further into smaller packets of meaning. Letters or graphemes (https://en.wikipedia.org/wiki/Grapheme) carry sentiment and meaning.[1]

We'll talk about character-based vector space models in later chapters. But for now let's just try to resolve the question of what a word is and how to divide up text into words.

What about invisible or implied words? Can you think of additional words that are implied by the single-word command "Don't!"? If you can force yourself to think like a machine and then switch back to thinking like a human, you might realize that there are three invisible words in that command. The single statement "Don't!" means "Don't you do that!" or "You, do not do that!" That's three hidden packets of meaning for a total of five tokens you'd like your machine to know about. But don't worry about invisible words for now. All you need for this chapter is a tokenizer that can recognize words that are spelled out. You'll worry about implied words and connotation and even meaning itself in chapter 4 and beyond.[2]

---

[1] Morphemes are parts of words that contain meaning in and of themselves. Geoffrey Hinton and other deep learning deep thinkers have demonstrated that even graphemes (letters)—the smallest indivisible piece of written text—can be treated as if they are intrinsically meaningful.

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

In this chapter, we show you straightforward algorithms for separating a string into words. You'll also extract pairs, triplets, quadruplets, and even quintuplets of tokens. These are called *n*-grams. Pairs of words are 2-grams (bigrams), triplets are 3-grams (trigrams), quadruplets are 4-grams, and so on. Using *n*-grams enables your machine to know about "ice cream" as well as the "ice" and "cream" that comprise it. Another 2-gram that you'd like to keep together is "Mr. Smith." Your tokens and your vector representation of a document will have a place for "Mr. Smith" along with "Mr." and "Smith," too.

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

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

2.1 Challenges (a preview of stemming)

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

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

In this chapter we show you how to make your NLP pipeline a bit smarter by dealing with these word spelling challenges using conventional stemming approaches. Later, in chapter 5, we show you statistical clustering approaches that only require you

to amass a collection of natural language text containing the words you're interested in. From that collection of text, the statistics of word usage will reveal "semantic stems" (actually, more useful clusters of words like lemmas or synonyms), without any hand-crafted regular expressions or stemming rules.

2.2 Building your vocabulary with a tokenizer

In NLP, *tokenization* is a particular kind of document segmentation. *Segmentation* breaks up text into smaller chunks or segments, with more focused information content. Segmentation can include breaking a document into paragraphs, paragraphs into sentences, sentences into phrases, or phrases into tokens (usually words) and punctuation. In this chapter, we focus on segmenting text into *tokens*, which is called tokenization.

You may have heard of tokenizers before, if you took a computer science class where you learned about how compilers work. A tokenizer used for compiling computer languages is often called a *scanner* or *lexer*. The vocabulary (the set of all the valid tokens) for a computer language is often called a *lexicon*, and that term is still used in academic articles about NLP. If the tokenizer is incorporated into the computer language compiler's parser, the parser is often called a scannerless parser. And tokens are the end of the line for the context-free grammars (CFG) used to parse computer languages. They are called *terminals* because they terminate a path from the root to the leaf in CFG. You'll learn more about *formal* grammars like CFGs and regular expressions in chapter 11 when you will use them to match patterns and extract information from natural language.

For the fundamental building blocks of NLP, there are equivalents in a computer language compiler:

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

Tokenization is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline. A tokenizer breaks unstructured data, natural language text, into chunks of information that can be counted as discrete elements. These counts of token occurrences in a document can be used directly as a vector representing that document. This immediately turns an unstructured string (text document) into a numerical data structure suitable for machine learning. These counts can be used directly by a computer to trigger useful actions and responses. Or they might also be used in a machine learning pipeline as features that trigger more complex decisions or behavior. The most common use for bag-of-words vectors created this way is for document retrieval, or search.

The simplest way to tokenize a sentence is to use whitespace within a string as the "delimiter" of words. In Python, this can be accomplished with the standard library

method `split`, which is available on all `str` object instances as well as on the `str` built-in class itself. See the following listing and figure 2.1 for an example.

Listing 2.1 Example Monticello sentence split into tokens

```
>>> sentence = """Thomas Jefferson began building Monticello at the
...   age of 26."""
>>> sentence.split()
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26.']
>>> str.split(sentence)
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26.']
```

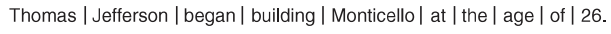

Figure 2.1 Tokenized phrase

As you can see, this built-in Python method already does a decent job tokenizing a simple sentence. Its only "mistake" was on the last word, where it included the sentence-ending punctuation with the token "26." Normally you'd like tokens to be separated from neighboring punctuation and other meaningful tokens in a sentence. The token "26." is a perfectly fine representation of a floating point number 26.0, but that would make this token different than another word "26" that occurred elsewhere in the corpus in the middle of sentences or the word "26?" that might occur at the end of a question. A good tokenizer should strip off this extra character to create the word "26" as an equivalent class for the words "26," "26!", "26?", and "26." And a more accurate tokenizer would also output a separate token for any sentence-ending punctuation so that a sentence segmenter or sentence boundary detector can find the end of that sentence.

For now, let's forge ahead with your imperfect tokenizer. You'll deal with punctuation and other challenges later. With a bit more Python, you can create a numerical vector representation for each word. These vectors are called *one-hot vectors*, and soon you'll see why. A sequence of these one-hot vectors fully captures the original document text in a sequence of vectors, a table of numbers. That will solve the first problem of NLP, turning words into numbers:

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

str.split () is your quick-and-dirty tokenizer.

Your vocabulary lists all the unique tokens (words) that you want to keep track of.

Sorted lexographically (lexically) so numbers come before letters, and capital letters come before lowercase letters.

For each word in the sentence, mark the column for that word in your vocabulary with a 1.

The empty table is as wide as your count of unique vocabulary terms and as high as the length of your document, 10 rows by 10 columns.

If you have trouble quickly reading all those ones and zeros, you're not alone. Pandas DataFrames can help make this a little easier on the eyes and more informative. Pandas wraps a 1D array with some helper functionality in an object called a `Series`. And Pandas is particularly handy with tables of numbers like lists of lists, 2D numpy arrays, 2D numpy matrices, arrays of arrays, dictionaries of dictionaries, and so on.

A `DataFrame` keeps track of labels for each column, allowing you to label each column in our table with the token or word it represents. A `DataFrame` can also keep track of labels for each row in the `DataFrame.index`, for speedy lookup. But this is usually just a consecutive integer for most applications. For now you'll use the default index of integers for the rows in your table of one-hot word vectors for this sentence about Thomas Jefferson, shown in the following listing.

Listing 2.2 One-hot vector sequence for the Monticello sentence

```
>>> import pandas as pd
>>> pd.DataFrame(onehot_vectors, columns=vocab)
   26.  Jefferson  Monticello  Thomas  age  at  began  building  of  the
0    0          0           0       1    0   0      0         0   0    0
1    0          1           0       0    0   0      0         0   0    0
2    0          0           0       0    0   0      1         0   0    0
3    0          0           0       0    0   0      0         1   0    0
4    0          0           1       0    0   0      0         0   0    0
5    0          0           0       0    0   1      0         0   0    0
6    0          0           0       0    0   0      0         0   0    1
7    0          0           0       0    1   0      0         0   0    0
8    0          0           0       0    0   0      0         0   1    0
9    1          0           0       0    0   0      0         0   0    0
```

One-hot vectors are super-sparse, containing only one nonzero value in each row vector. So we can make that table of one-hot row vectors even prettier by replacing zeros with blanks. Don't do this with any `DataFrame` you intend to use in your machine learning pipeline, because it'll create a lot of non-numerical objects within your numpy array, mucking up the math. But if you just want to see how this one-hot vector sequence is like a mechanical music box cylinder, or a player piano drum, the following listing can be a handy view of your data.

Listing 2.3 Prettier one-hot vectors

```
>>> df = pd.DataFrame(onehot_vectors, columns=vocab)
>>> df[df == 0] = ''
>>> df
  26. Jefferson Monticello Thomas age at began building of the
0                              1
1           1
2                                        1
3                                             1
4                       1
5                                  1
6                                                  1
7                                 1
8                                                    1
9   1
```

In this representation of your one-sentence document, each row is a vector for a single word. The sentence has 10 words, all unique, and it doesn't reuse any words. The table has 10 columns (words in your vocabulary) and 10 rows (words in the document). A "1" in a column indicates a vocabulary word that was present at that position in the document. So if you wanted to know what the third word in a document was, you'd go to the third row in the table. And you'd look up at the column heading for the "1" value in the third row (the row labeled 2, because the row numbers start at 0). At the top of that column, the seventh column in the table, you can find the natural language representation of that word, "began."

Each row of the table is a binary row vector, and you can see why it's also called a one-hot vector: all but one of the positions (columns) in a row are `0` or blank. Only one column, or position in the vector, is "hot" ("1"). A one (`1`) means on, or hot. A zero (`0`) means off, or absent. And you can use the vector `[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]` to represent the word "began" in your NLP pipeline.

One nice feature of this vector representation of words and tabular representation of documents is that no information is lost.[3] As long as you keep track of which words are indicated by which column, you can reconstruct the original document from this table of one-hot vectors. And this reconstruction process is 100% accurate, even though your tokenizer was only 90% accurate at generating the tokens you thought would be useful. As a result, one-hot word vectors like this are typically used in neural nets, sequence-to-sequence language models, and generative language models. They're a good choice for any model or NLP pipeline that needs to retain all the meaning inherent in the original text.

This one-hot vector table is like a recording of the original text. If you squint hard enough you might be able to imagine that the matrix of ones and zeros above is a player piano paper roll.[4] Or maybe it's the bumps on the metal drum of a music box.[5] The vocabulary key at the top tells the machine which "note" or word to play for each row in the sequence of words or piano music. Unlike a player piano, your mechanical word recorder and player is only allowed to use one "finger" at a time. It can only play one "note" or word at a time. It's one-hot. And each note or word is played for the same amount of "time" with a consistent pace. There's no variation in the spacing of the words.

But this is just one way of thinking of one-hot word vectors. You can come up with whatever mental model makes sense for you. The important thing is that you've turned a sentence of natural language words into a sequence of numbers, or vectors. Now you can have the computer read and do math on the vectors just like any other vector or list of numbers. This allows your vectors to be input into any natural language processing pipeline that requires this kind of vector.

You could also play a sequence of one-hot encoded vectors back if you want to generate text for a chat bot, just like a player piano might play a song for a less artificial audience. Now all you need to do is figure out how to build a player piano that can "understand" and combine those word vectors in new ways. Ultimately, you'd like your chatbot or NLP pipeline to play us, or say something, you haven't heard before. We get to that in chapters 9 and 10 when we talk about LSTM models, and similar neural networks.

---

[3] Except for the distinction between various white spaces that were "split" with your tokenizer. If you wanted to get the original document back, unless your tokenizer keeps track of the white spaces it discarded during tokenization, you can't. If your tokenizer didn't preserve that information, there's no way to tell whether a space or a newline or a tab or even nothing should be inserted at each position between words. But the information content of whitespace is low, negligible in most English documents. And many modern NLP parsers and tokenizers retain that whitespace information for you, if you ever need it.

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

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

This representation of a sentence in one-hot word vectors retains all the detail, grammar, and order of the original sentence. And you've successfully turned words into numbers that a computer can "understand." They are also a particular kind of number that computers like a lot: binary numbers. But this is a big table for a short sentence. If you think about it, you've expanded the file size that would be required to store your document. For a long document this might not be practical. Your document size (the length of the vector table) would grow to be huge. The English language contains at least 20,000 common words, millions if you include names and other proper nouns. And your one-hot vector representation requires a new table (matrix) for every document you want to process. This is almost like a raw "image" of your document. If you've done any image processing, you know that you need to do dimension reduction if you want to extract useful information from the data.

Let's run through the math to give you an appreciation for just how big and unwieldy these "player piano paper rolls" are. In most cases, the vocabulary of tokens you'll use in an NLP pipeline will be much more than 10,000 or 20,000 tokens. Sometimes it can be hundreds of thousands or even millions of tokens. Let's assume you have a million tokens in your NLP pipeline vocabulary. And let's say you have a meager 3,000 books with 3,500 sentences each and 15 words per sentence—reasonable averages for short books. That's a whole lot of big tables (matrices):

```
>>> num_rows = 3000 * 3500 * 15
>>> num_rows
157500000
>>> num_bytes = num_rows * 1000000
>>> num_bytes
157500000000000
>>> num_bytes / 1e9
157500  # gigabytes
>>> _ / 1000
157.5  # terabytes
```

Number of rows in the table

Number of bytes, if you use only one byte for each cell in your table

In a python interactive console, the variable name "_" is automatically assigned the value of the previous output. This is handy if you forget to explicitly assign the output of a function or expression to a variable name like you did for num_bytes and num_rows.

You're talking more than a million million bits, even if you use a single bit for each cell in your matrix. At one bit per cell, you'd need nearly 20 terabytes of storage for a small bookshelf of books processed this way. Fortunately, you don't ever use this data structure for storing documents. You only use it temporarily, in RAM, while you're processing documents one word at a time.

So storing all those zeros, and trying to remember the order of the words in all your documents, doesn't make much sense. It's not practical. And what you really want to do is compress the meaning of a document down to its essence. You'd like to compress your document down to a single vector rather than a big table. And you're willing to give up perfect "recall." You just want to capture most of the meaning (information) in a document, not all of it.

What if you split your documents into much shorter chunks of meaning, say sentences. And what if you assumed that most of the meaning of a sentence can be gleaned

from just the words themselves. Let's assume you can ignore the order and grammar of the words, and jumble them all up together into a "bag," one bag for each sentence or short document. That turns out to be a reasonable assumption. Even for documents several pages long, a bag-of-words vector is still useful for summarizing the essence of a document. You can see that for your sentence about Jefferson, even after you sorted all the words lexically, a human can still guess what the sentence was about. So can a machine. You can use this new bag-of-words vector approach to compress the information content for each document into a data structure that's easier to work with.

If you summed all these one-hot vectors together, rather than "replaying" them one at a time, you'd get a bag-of-words vector. This is also called a word frequency vector, because it only counts the *frequency* of words, not their order. You could use this single vector to represent the whole document or sentence in a single, reasonable-length vector. It would only be as long as your vocabulary size (the number of unique tokens you want to keep track of).

Alternatively, if you're doing basic keyword search, you could *OR* the one-hot word vectors into a binary bag-of-words vector. And you could ignore a lot of words that wouldn't be interesting as search terms or keywords. This would be fine for a search engine index or the first filter for an information retrieval system. Search indexes only need to know the presence or absence of each word in each document to help you find those documents later.

Just like laying your arm on the piano, hitting all the notes (words) at once doesn't make for a pleasant, meaningful experience. Nonetheless this approach turns out to be critical to helping a machine "understand" a whole group of words as a unit. And if you limit your tokens to the 10,000 most important words, you can compress your numerical representation of your imaginary 3,500 sentence book down to 10 kilobytes, or about 30 megabytes for your imaginary 3,000-book corpus. One-hot vector sequences would require hundreds of gigabytes.

Fortunately, the words in your vocabulary are sparsely utilized in any given text. And for most bag-of-words applications, we keep the documents short; sometimes just a sentence will do. So rather than hitting all the notes on a piano at once, your bag-of-words vector is more like a broad and pleasant piano chord, a combination of notes (words) that work well together and contain meaning. Your chatbot can handle these chords even if there's a lot of "dissonance" from words in the same statement that aren't normally used together. Even dissonance (odd word usage) is useful information about a statement that a machine learning pipeline can make use of.

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

Here's what your single text document, the sentence about Thomas Jefferson, looks like as a binary bag-of-words vector:

```
>>> sentence_bow = {}
>>> for token in sentence.split():
...     sentence_bow[token] = 1
>>> sorted(sentence_bow.items())
[('26.', 1)
 ('Jefferson', 1),
 ('Monticello', 1),
 ('Thomas', 1),
 ('age', 1),
 ('at', 1),
 ('began', 1),
 ('building', 1),
 ('of', 1),
 ('the', 1)]
```

One thing you might notice is that Python's `sorted()` puts decimal numbers before characters, and capitalized words before lowercase words. This is the ordering of characters in the ASCII and Unicode character sets. Capital letters come before lowercase letters in the ASCII table. The order of your vocabulary is unimportant. As long as you are consistent across all the documents you tokenize this way, a machine learning pipeline will work equally well with any vocabulary order.

And you might also notice that using a `dict` (or any paired mapping of words to their 0/1 values) to store a binary vector shouldn't waste much space. Using a dictionary to represent your vector ensures that it only has to store a 1 when any one of the thousands, or even millions, of possible words in your dictionary appear in a particular document. You can see how it would be much less efficient to represent a bag of words as a continuous list of 0's and 1's with an assigned location in a "dense" vector for each of the words in a vocabulary of, say, 100,000 words. This dense binary vector representation of your "Thomas Jefferson" sentence would require 100 kB of storage. Because a dictionary "ignores" the absent words, the words labeled with a 0, the dictionary representation only requires a few bytes for each word in your 10-word sentence. And this dictionary could be made even more efficient if you represented each word as an integer pointer to each word's location within your lexicon—the list of words that makes up your vocabulary for a particular application.

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

```
>>> import pandas as pd
>>> df = pd.DataFrame(pd.Series(dict([(token, 1) for token in
...   sentence.split()])), columns=['sent']).T
>>> df
      26.  Jefferson  Monticello  Thomas  age  at  began  building  of  the
sent    1          1           1       1    1   1      1         1   1    1
```

Let's add a few more texts to your corpus to see how a DataFrame stacks up. A DataFrame indexes both the columns (documents) and rows (words) so it can be an "inverse index" for document retrieval, in case you want to find a Trivial Pursuit answer in a hurry.

Listing 2.4 Construct a `DataFrame` of bag-of-words vectors

```
>>> sentences = """Thomas Jefferson began building Monticello at the\
...   age of 26.\n"""
>>> sentences += """Construction was done mostly by local masons and\
...   carpenters.\n"""
>>> sentences += "He moved into the South Pavilion in 1770.\n"
>>> sentences += """Turning Monticello into a neoclassical masterpiece\
...   was Jefferson's obsession."""
>>> corpus = {}
>>> for i, sent in enumerate(sentences.split('\n')):
...     corpus['sent{}'.format(i)] = dict((tok, 1) for tok in
...         sent.split())
>>> df = pd.DataFrame.from_records(corpus).fillna(0).astype(int).T
>>> df[df.columns[:10]]
        1770.  26.  Construction   ...    Pavilion  South  Thomas
sent0       0    1             0   ...           0      0       1
sent1       0    0             1   ...           0      0       0
sent2       1    0             0   ...           1      1       0
sent3       0    0             0   ...           0      0       0
```

This is the original sentence defined in listing 2.1.

Normally you should use .splitlines() but here you explicitly add a single '\n' character to the end of each line/sentence, so you need to explicitly split on this character.

This shows only the first 10 tokens (DataFrame columns), to avoid wrapping.

With a quick scan, you can see little overlap in word usage for these sentences. Among the first seven words in your vocabulary, only the word "Monticello" appears in more than one sentence. Now you need to be able to compute this overlap within your pipeline whenever you want to compare documents or search for similar documents. One way to check for the similarities between sentences is to count the number of overlapping tokens using a *dot product*.

2.2.1 Dot product

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

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

The dot product is also called the *scalar product* because it produces a single scalar value as its output. This helps distinguish it from the *cross product*, which produces a vector as its output. Obviously, these names reflect the shape of the symbols used to indicate the dot product (`"·"`) and cross product (`"x"`) in formal mathematical notation.

The scalar value output by the scalar product can be calculated by multiplying all the elements of one vector by all the elements of a second vector, and then adding up those normal multiplication products.

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

Listing 2.5 Example dot product calculation

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

Multiplication of numpy arrays is a "vectorized" operation that is very efficient.

You shouldn't iterate through vectors this way unless you want to slow down your pipeline.

TIP The dot product is equivalent to the *matrix product*, which can be accomplished in numpy with the `np.matmul()` function or the `@` operator. Since all vectors can be turned into Nx1 or 1xN matrices, you can use this shorthand operator on two column vectors (Nx1) by transposing the first one so their inner dimensions line up, like this: `v1.reshape(-1, 1).T @ v2.reshape(-1, 1)`, which outputs your scalar product within a 1x1 matrix: `array([[20]])`.

2.2.2 Measuring bag-of-words overlap

If we can measure the bag of words overlap for two vectors, we can get a good estimate of how similar they are in the words they use. And this is a good estimate of how similar they are in meaning. So let's use your newfound dot product understanding to estimate the bag-of-words vector overlap between some new sentences and the original sentence about Thomas Jefferson (`sent0`).

Listing 2.6 Overlap of word counts for two bag-of-words vectors

```
>>> df = df.T
>>> df.sent0.dot(df.sent1)
0
>>> df.sent0.dot(df.sent2)
1
>>> df.sent0.dot(df.sent3)
1
```

From this you can tell that one word was used in both `sent0` and `sent2`. Likewise one of the words in your vocabulary was used in both `sent0` and `sent3`. This overlap of words is a measure of their similarity. Interestingly, that oddball sentence, `sent1`, was the only sentence that did not mention Jefferson or Monticello directly, but used a completely different set of words to convey information about other anonymous people.

Here's one way to find the word that is shared by `sent0` and `sent3`, the word that gave you that last dot product of 1:

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

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

2.2.3 A token improvement

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

In the preceding example, the last token in the sentence was corrupted by a period at the end of "26." The trailing period can be misleading for the subsequent sections of an NLP pipeline, like stemming, where you would like to group similar words together using rules that rely on consistent word spellings. The following listing shows one way.

Listing 2.7 Tokenize the Monticello sentence with a regular expression

```
>>> import re
>>> sentence = """Thomas Jefferson began building Monticello at the\
...   age of 26."""
>>> tokens = re.split(r'[-\s.,;!?]+', sentence)
>>> tokens
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26',
 '']
```

This splits the sentence on whitespace or punctuation that occurs at least once (note the '+' after the closing square bracket in the regular expression). See sidenote that follows.

We promised we'd use more regular expressions. Hopefully they're starting to make a little more sense than they did when we first used them. If not, the following sidenote will walk you through each character of the regular expression. And if you want to dig even deeper, check out appendix B.

How regular expressions work

Here's how the regular expression in listing 2.7 works. The square brackets (`[` and `]`) are used to indicate a *character class*, a set of characters. The plus sign after the closing square bracket (`]`) means that a match must contain one or more of the characters inside the square brackets. The `\s` within the character class is a shortcut to a predefined character class that includes all whitespace characters like those created when you press the `[space]`, `[tab]`, and `[return]` keys. The character class `r'[\s]'` is equivalent to `r' \t\n\r\x0b\x0c'`. The six whitespace characters are space (`' '`), tab (`'\t'`), return (`'\r'`), newline (`'\n'`), and form-feed (`'\f'`).

You didn't use any character ranges here, but you may want to later. A character range is a special kind of character class indicated within square brackets and a hyphen, like `r'[a-z]'` to match all lowercase letters. The character range `r'[0-9]'` matches any digit 0 through 9 and is equivalent to `r'[0123456789]'`. The regular expression `r'[_a-zA-Z]'` would match any underscore character (`'_'`) or letter of the English alphabet (upper- or lowercase).

The hyphen (`-`) right after the opening square bracket is a bit of a quirk of regexes. You can't put a hyphen just anywhere inside your square brackets, because the regex parser may think you mean a character range like `r'[0-9]'`. To let it know that you really mean a literal hyphen character, you have to put it right after the open square bracket for the character class. So whenever you want to indicate an actual hyphen (dash) character in your character class, you need to make sure it's the first character, or you need to escape it with a backslash.

The `re.split` function goes through each character in the input string (the second argument, `sentence`) left to right looking for any matches based on the "program" in the regular expression (the first argument, `r'[-\s.,;!?]+'`). When it finds a match, it breaks the string right before that matched character and right after it, skipping over the matched character or characters. So the `re.split` line will work just like `str.split`, but it will work for any kind of character or multicharacter sequence that matches your regular expression.

The parentheses ("(" and ")") are used to group regular expressions just like they're used to group mathematical, Python, and most other programming language expressions. These parentheses force the regular expression to match the entire expression within the parentheses before moving on to try to match the characters that follow the parentheses.

Improved regular expression for separating words

Let's compile our regular expression so that our tokenizer will run faster. Compiled regular expression objects are handy for a lot of reasons, not just speed.

When to compile your regex patterns

The regular expression module in Python allows you to precompile regular expressions,[a] which you then can reuse across your code base. For example, you might have a regex that extracts phone numbers. You could use `re.compile()` to precompile the expression and pass it along as an argument to a function or class doing tokenization. This is rarely a speed advantage, because Python caches the compiled objects for the last `MAXCACHE=100` regular expressions. But if you have more than 100 different regular expressions at work, or you want to call methods of the regular expression rather than the corresponding `re` functions, `re.compile` can be useful:

```
>>> pattern = re.compile(r"([-\s.,;!?])+")
>>> tokens = pattern.split(sentence)
>>> tokens[-10:]  # just the last 10 tokens
['the', ' ', 'age', ' ', 'of', ' ', '26', '.', '']
```

[a] See stack overflow or the latest Python documentation for more details (http://stackoverflow.com/a/452143/623735).

This simple regular expression is helping to split off the period from the end of the token "26." However, you have a new problem. You need to filter the whitespace and punctuation characters you don't want to include in your vocabulary. See the following code and figure 2.2:

```
>>> sentence = """Thomas Jefferson began building Monticello at the\
...   age of 26."""
>>> tokens = pattern.split(sentence)
>>> [x for x in tokens if x and x not in '- \t\n.,;!?']
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26']
```

If you want practice with lambda and filter(), use list(filter(lambda x: x if x and x not in '- \t\n.,;!?' else None, tokens)).

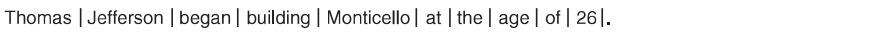

Figure 2.2 Tokenized phrase

So the built-in Python `re` package seems to do just fine on this example sentence, as long as you are careful to filter out undesirable tokens. There's really no reason to look elsewhere for regular expression packages, except…

When to use the new regex module in Python

There's a new regular expression package called `regex` that will eventually replace the `re` package. It's completely backward compatible and can be installed with `pip` from pypi. It's useful new features include support for

- Overlapping match sets
- Multithreading
- Feature-complete support for Unicode
- Approximate regular expression matches (similar to TRE's `agrep` on UNIX systems)
- Larger default MAXCACHE (500 regexes)

Even though `regex` will eventually replace the `re` package and is completely backward compatible with `re`, for now you must install it as an additional package using a package manager such as pip:

```
$ pip install regex
```

You can find more information about the *regex* module on the PyPI website (https://pypi.python.org/pypi/regex).

As you can imagine, tokenizers can easily become complex. In one case, you might want to split based on periods, but only if the period isn't followed by a number, in order to avoid splitting decimals. In another case, you might not want to split after a period that is part of "smiley" emoticon symbol, such as in a Twitter message.

Several Python libraries implement tokenizers, each with its own advantages and disadvantages:

- *spaCy*—Accurate , flexible, fast, Python
- *Stanford CoreNLP*—More accurate, less flexible, fast, depends on Java 8
- *NLTK*—Standard used by many NLP contests and comparisons, popular, Python

NLTK and Stanford CoreNLP have been around the longest and are the most widely used for comparison of NLP algorithms in academic papers. Even though the Stanford CoreNLP has a Python API, it relies on the Java 8 CoreNLP backend, which must be installed and configured separately. So you can use the Natural Language Toolkit (NLTK) tokenizer here to get you up and running quickly; it will help you duplicate the results you see in academic papers and blog posts.

You can use the NLTK function `RegexpTokenizer` to replicate your simple tokenizer example like this:

```
>>> from nltk.tokenize import RegexpTokenizer
>>> tokenizer = RegexpTokenizer(r'\w+|$[0-9.]+|\S+')
>>> tokenizer.tokenize(sentence)
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
```

```
 'at',
 'the',
 'age',
 'of',
 '26',
 '.']
```

This tokenizer is a bit better than the one you used originally, because it ignores whitespace tokens. It also separates sentence-ending trailing punctuation from tokens that do not contain any other punctuation characters.

An even better tokenizer is the Treebank Word Tokenizer from the NLTK package. It incorporates a variety of common rules for English word tokenization. For example, it separates phrase-terminating punctuation (?!.;,) from adjacent tokens and retains decimal numbers containing a period as a single token. In addition it contains rules for English contractions. For example "don't" is tokenized as `["do", "n't"]`. This tokenization will help with subsequent steps in the NLP pipeline, such as stemming. You can find all the rules for the Treebank Tokenizer at http://www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.treebank. See the following code and figure 2.3:

```
>>> from nltk.tokenize import TreebankWordTokenizer
>>> sentence = """Monticello wasn't designated as UNESCO World Heritage\
...   Site until 1987."""
>>> tokenizer = TreebankWordTokenizer()
>>> tokenizer.tokenize(sentence)
['Monticello',
 'was',
 "n't",
 'designated',
 'as',
 'UNESCO',
 'World',
 'Heritage',
 'Site',
 'until',
 '1987',
 '.']
```

Figure 2.3 Tokenized phrase

CONTRACTIONS

You might wonder why you would split the contraction `wasn't` into `was` and `n't`. For some applications, like grammar-based NLP models that use syntax trees, it's important to separate the words `was` and `not` to allow the syntax tree parser to have a consistent, predictable set of tokens with known grammar rules as its input. There are a variety of standard and nonstandard ways to contract words. By reducing contractions to their constituent words, a dependency tree parser or syntax parser only need be

programmed to anticipate the various spellings of individual words rather than all possible contractions.

Tokenize informal text from social networks such as Twitter and Facebook

The NLTK library includes a tokenizer—`casual_tokenize`—that was built to deal with short, informal, emoticon-laced texts from social networks where grammar and spelling conventions vary widely.

The `casual_tokenize` function allows you to strip usernames and reduce the number of repeated characters within a token:

```
>>> from nltk.tokenize.casual import casual_tokenize
>>> message = """RT @TJMonticello Best day everrrrrrr at Monticello.\
...   Awesommmmmmeeeeeeee day :*)"""
>>> casual_tokenize(message)
['RT', '@TJMonticello',
 'Best', 'day','everrrrrrr', 'at', 'Monticello', '.',
 'Awesommmmmmeeeeeeee', 'day', ':*)']
>>> casual_tokenize(message, reduce_len=True, strip_handles=True)
['RT',
 'Best', 'day', 'everrr', 'at', 'Monticello', '.',
 'Awesommmeee', 'day', ':*)']
```

2.2.4 Extending your vocabulary with n-grams

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

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

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

WE ALL GRAM FOR N-GRAMS

An *n*-gram is a sequence containing up to *n* elements that have been extracted from a sequence of those elements, usually a string. In general the "elements" of an *n*-gram can be characters, syllables, words, or even symbols like "A," "T," "G," and "C" used to represent a DNA sequence.[6]

In this book, we're only interested in *n*-grams of words, not characters.[7] So in this book, when we say 2-gram, we mean a pair of words, like "ice cream." When we say 3-gram, we mean a triplet of words like "beyond the pale" or "Johann Sebastian Bach"

[6] Linguistic and NLP techniques are often used to glean information from DNA and RNA. This site provides a list of nucleic acid symbols that can help you translate nucleic acid language into a human-readable language: "Nucleic Acid Sequence" (https://en.wikipedia.org/wiki/Nucleic_acid_sequence).

[7] You may have learned about trigram indexes in your database class or the documentation for PostgreSQL (`postgres`). But these are triplets of characters. They help you quickly retrieve fuzzy matches for strings in a massive database of strings using the "%," "~," and "*" symbols in SQL full text search queries.

or "riddle me this." *n*-grams don't have to mean something special together, like compound words. They merely have to be frequent enough together to catch the attention of your token counters.

Why bother with *n*-grams? As you saw earlier, when a sequence of tokens is vectorized into a bag-of-words vector, it loses a lot of the meaning inherent in the order of those words. By extending your concept of a token to include multiword tokens, *n*-grams, your NLP pipeline can retain much of the meaning inherent in the order of words in your statements. For example, the meaning-inverting word "not" will remain attached to its neighboring words, where it belongs. Without *n*-gram tokenization, it would be free floating. Its meaning would be associated with the entire sentence or document rather than its neighboring words. The 2-gram "was not" retains much more of the meaning of the individual words "not" and "was" than those 1-grams alone in a bag-of-words vector. A bit of the context of a word is retained when you tie it to its neighbor(s) in your pipeline.

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

Let's use your original sentence about Thomas Jefferson to show what a 2-gram tokenizer should output, so you know what you're trying to build:

```
>>> tokenize_2grams("Thomas Jefferson began building Monticello at the\
...   age of 26.")
['Thomas Jefferson',
 'Jefferson began',
 'began building',
 'building Monticello',
 'Monticello at',
 'at the',
 'the age',
 'age of',
 'of 26']
```

I bet you can see how this sequence of 2-grams retains a bit more information than if you'd just tokenized the sentence into words. The later stages of your NLP pipeline will only have access to whatever tokens your tokenizer generates. So you need to let those later stages know that "Thomas" wasn't about "Isaiah Thomas" or the "Thomas & Friends" cartoon. *n*-grams are one of the ways to maintain context information as data passes through your pipeline.

Here's the original 1-gram tokenizer:

```
>>> sentence = """Thomas Jefferson began building Monticello at the\
...   age of 26."""
```

```
>>> pattern = re.compile(r"([-\s.,;!?])+")
>>> tokens = pattern.split(sentence)
>>> tokens = [x for x in tokens if x and x not in '- \t\n.,;!?']
>>> tokens
['Thomas',
 'Jefferson',
 'began',
 'building',
 'Monticello',
 'at',
 'the',
 'age',
 'of',
 '26']
```

And this is the *n*-gram tokenizer from `nltk` in action:

```
>>> from nltk.util import ngrams
>>> list(ngrams(tokens, 2))
[('Thomas', 'Jefferson'),
 ('Jefferson', 'began'),
 ('began', 'building'),
 ('building', 'Monticello'),
 ('Monticello', 'at'),
 ('at', 'the'),
 ('the', 'age'),
 ('age', 'of'),
 ('of', '26')]
>>> list(ngrams(tokens, 3))
[('Thomas', 'Jefferson', 'began'),
 ('Jefferson', 'began', 'building'),
 ('began', 'building', 'Monticello'),
 ('building', 'Monticello', 'at'),
 ('Monticello', 'at', 'the'),
 ('at', 'the', 'age'),
 ('the', 'age', 'of'),
 ('age', 'of', '26')]
```

TIP In order to be more memory efficient, the `ngrams` function of the NLTK library returns a Python generator. Python generators are "smart" functions that behave like iterators, yielding only one element at a time instead of returning the entire sequence at once. This is useful within `for` loops, where the generator will load each individual item instead of loading the whole item list into memory. However, if you want to inspect all the returned *n*-grams at once, convert the generator to a list as you did in the earlier example. Keep in mind that you should only do this in an interactive session, not within a long-running task tokenizing large texts.

The *n*-grams are provided in the previous listing as tuples, but they can easily be joined together if you'd like all the tokens in your pipeline to be strings. This will allow the later stages of the pipeline to expect a consistent datatype as input, string sequences:

```
>>> two_grams = list(ngrams(tokens, 2))
>>> [" ".join(x) for x in two_grams]
```

```
['Thomas Jefferson',
 'Jefferson began',
 'began building',
 'building Monticello',
 'Monticello at',
 'at the',
 'the age',
 'age of',
 'of 26']
```

You might be able to sense a problem here. Looking at your earlier example, you can imagine that the token "Thomas Jefferson" will occur across quite a few documents. However the 2-grams "of 26" or even "Jefferson began" will likely be extremely rare. If tokens or *n*-grams are extremely rare, they don't carry any correlation with other words that you can use to help identify topics or themes that connect documents or classes of documents. So rare *n*-grams won't be helpful for classification problems. You can imagine that most 2-grams are pretty rare—even more so for 3- and 4-grams.

Because word combinations are rarer than individual words, your vocabulary size is exponentially approaching the number of *n*-grams in all the documents in your corpus. If your feature vector dimensionality exceeds the length of all your documents, your feature extraction step is counterproductive. It'll be virtually impossible to avoid overfitting a machine learning model to your vectors; your vectors have more dimensions than there are documents in your corpus. In chapter 3, you'll use document frequency statistics to identify *n*-grams so rare that they are not useful for machine learning. Typically, *n*-grams are filtered out that occur too infrequently (for example, in three or fewer different documents). This scenario is represented by the "rare token" filter in the coin-sorting machine of chapter 1.

Now consider the opposite problem. Consider the 2-gram "at the" in the previous phrase. That's probably not a rare combination of words. In fact it might be so common, spread among most of your documents, that it loses its utility for discriminating between the meanings of your documents. It has little predictive power. Just like words and other tokens, *n*-grams are usually filtered out if they occur too often. For example, if a token or *n*-gram occurs in more than 25% of all the documents in your corpus, you usually ignore it. This is equivalent to the "stop words" filter in the coin-sorting machine of chapter 1. These filters are as useful for *n*-grams as they are for individual tokens. In fact, they're even more useful.

STOP WORDS

Stop words are common words in any language that occur with a high frequency but carry much less substantive information about the meaning of a phrase. Examples of some common stop words include[8]

- a, an
- the, this

[8] A more comprehensive list of stop words for various languages can be found in NLTK's corpora (https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/packages/corpora/stopwords.zip).

- and, or
- of, on

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

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

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

Designing a filter for stop words depends on your particular application. Vocabulary size will drive the computational complexity and memory requirements of all subsequent steps in the NLP pipeline. But stop words are only a small portion of your total vocabulary size. A typical stop word list has only 100 or so frequent and unimportant words listed in it. But a vocabulary size of 20,000 words would be required to keep track of 95% of the words seen in a large corpus of tweets, blog posts, and news articles.[9] And that's just for 1-grams or single-word tokens. A 2-gram vocabulary designed to catch 95% of the 2-grams in a large English corpus will generally have more than 1 million unique 2-gram tokens in it.

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

So if you have sufficient memory and processing bandwidth to run all the NLP steps in your pipeline on the larger vocabulary, you probably don't want to worry

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

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

If you do decide to arbitrarily filter out a set of stop words during tokenization, a Python list comprehension is sufficient. Here you take a few stop words and ignore them when you iterate through your token list:

```
>>> stop_words = ['a', 'an', 'the', 'on', 'of', 'off', 'this', 'is']
>>> tokens = ['the', 'house', 'is', 'on', 'fire']
>>> tokens_without_stopwords = [x for x in tokens if x not in stop_words]
>>> print(tokens_without_stopwords)
['house', 'fire']
```

You can see that some words carry a lot more meaning than others. And you can lose more than half the words in some sentences without significantly affecting their meaning. You can often get your point across without articles, prepositions, or even forms of the verb "to be." Imagine someone doing sign language or in a hurry to write a note to themselves. Which words would they chose to always skip? That's how stop words are chosen.

To get a complete list of "canonical" stop words, NLTK is probably the most generally applicable list. See the following listing.

Listing 2.8 NLTK list of stop words

```
>>> import nltk
>>> nltk.download('stopwords')
>>> stop_words = nltk.corpus.stopwords.words('english')
>>> len(stop_words)
153
>>> stop_words[:7]
['i', 'me', 'my', 'myself', 'we', 'our', 'ours']
>>> [sw for sw in stopwords if len(sw) == 1]
['i', 'a', 's', 't', 'd', 'm', 'o', 'y']
```

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

WARNING The set of English stop words that `sklearn` uses is quite different from those in NLTK. At the time of this writing, `sklearn` has 318 stop words. Even NLTK upgrades its corpora periodically, including the stop words list.

When we reran listing 2.8 to count the NLTK stop words with `nltk` version 3.2.5 in Python 3.6, we got 179 stop words instead of 153 from an earlier version.

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

Depending on how much natural language information you want to discard ;), you can take the union or the intersection of multiple stop word lists for your pipeline. Here's a comparison of `sklearn` stop words (version 0.19.2) and `nltk` stop words (version 3.2.5).

Listing 2.9 NLTK list of stop words

```
>>> from sklearn.feature_extraction.text import\
...   ENGLISH_STOP_WORDS as sklearn_stop_words
>>> len(sklearn_stop_words)
318
>>> len(stop_words)
179
>>> len(stop_words.union(sklearn_stop_words))
378
>>> len(stop_words.intersection(sklearn_stop_words))
119
```

NTLK's list contains 60 stop words that aren't in the larger sklearn set.

NLTK and sklearn agree on fewer than a third of their stop words (119 out of 378).

2.2.5 Normalizing your vocabulary

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

CASE FOLDING

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

However, some information is often communicated by capitalization of a word—for example, 'doctor' and 'Doctor' often have different meanings. Often capitalization is used to indicate that a word is a proper noun, the name of a person, place, or thing. You'll want to be able to recognize proper nouns as distinct from other words, if

named entity recognition is important to your pipeline. However, if tokens aren't case normalized, your vocabulary will be approximately twice as large, consume twice as much memory and processing time, and might increase the amount of training data you need to label for your machine learning pipeline to converge to an accurate, general solution. Just as in any other machine learning pipeline, your labeled dataset used for training must be "representative" of the space of all possible feature vectors your model must deal with, including variations in capitalization. For 100,000-D bag-of-words vectors, you usually must have 100,000 labeled examples, and sometimes even more than that, to train a supervised machine learning pipeline without overfitting. In some situations, cutting your vocabulary size by half can be worth the loss of information content.

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

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

And if you're certain that you want to normalize the case for an entire document, you can `lower()` the text string in one operation, before tokenization. But this will prevent advanced tokenizers that can split *camel case* words like "WordPerfect," "FedEx," or "stringVariableName."[10] Maybe you want WordPerfect to be it's own unique thing (token), or maybe you want to reminisce about a more perfect word processing era. It's up to you to decide when and how to apply case folding.

With case normalization, you are attempting to return these tokens to their "normal" state before grammar rules and their position in a sentence affected their capitalization. The simplest and most common way to normalize the case of a text string is to lowercase all the characters with a function like Python's built-in `str.lower()`.[11] Unfortunately this approach will also "normalize" away a lot of meaningful capitalization in addition to the less meaningful first-word-in-sentence capitalization you intended to normalize away. A better approach for case normalization is to lowercase only the first word of a sentence and allow all other words to retain their capitalization.

Lowercasing on the first word in a sentence preserves the meaning of proper nouns in the middle of a sentence, like "Joe" and "Smith" in "Joe Smith." And it properly groups words together that belong together, because they're only capitalized when they are at the beginning of a sentence, since they aren't proper nouns. This prevents "Joe" from being confused with "coffee" ("joe")[12] during tokenization. And

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

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

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

this approach prevents the blacksmith connotation of "smith" being confused with the proper name "Smith" in a sentence like "A word smith had a cup of joe." Even with this careful approach to case normalization, where you lowercase words only at the start of a sentence, you will still introduce capitalization errors for the rare proper nouns that start a sentence. "Joe Smith, the word smith, with a cup of joe." will produce a different set of tokens than "Smith the word with a cup of joe, Joe Smith." And you may not want that. In addition, case normalization is useless for languages that don't have a concept of capitalization.

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

By generalizing your model to work with text that has odd capitalization, case normalization can reduce overfitting for your machine learning pipeline. Case normalization is particularly useful for a search engine. For search, normalization increases the number of matches found for a particular query. This is often called the "recall" performance metric for a search engine (or any other classification model).[13]

For a search engine without normalization, if you searched for "Age" you would get a different set of documents than if you searched for "age." "Age" would likely occur in phrases like "New Age" or "Age of Reason." In contrast, "age" would more likely occur in phrases like "at the age of" in your sentence about Thomas Jefferson. By normalizing the vocabulary in your search index (as well as the query), you can ensure that both kinds of documents about "age" are returned, regardless of the capitalization in the query from the user.

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

---

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

STEMMING

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

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

Stemming is important for keyword search or information retrieval. It allows you to search for "developing houses in Portland" and get web pages or documents that use both the word "house" and "houses" and even the word "housing," because these words are all stemmed to the "hous" token. Likewise you might receive pages with the words "developer" and "development" rather than "developing," because all these words typically reduce to the stem "develop." As you can see, this is a "broadening" of your search, ensuring that you are less likely to miss a relevant document or web page. This broadening of your search results would be a big improvement in the "recall" score for how well your search engine is doing its job at returning all the relevant documents.[14]

But stemming could greatly reduce the "precision" score for your search engine, because it might return many more irrelevant documents along with the relevant ones. In some applications this "false-positive rate" (proportion of the pages returned that you don't find useful) can be a problem. So most search engines allow you to turn off stemming and even case normalization by putting quotes around a word or phrase. Quoting indicates that you only want pages containing the exact spelling of a phrase, such as "'Portland Housing Development software.'" That would return a different sort of document than one that talks about a "'a Portland software developer's house'". And there are times when you want to search for "Dr. House's calls" and not "dr house call," which might be the effective query if you used a stemmer on that query.

---

[14] Review appendix D if you've forgotten how to measure recall or visit the Wikipedia page to learn more (https://en.wikipedia.org/wiki/Precision_and_recall).

Here's a simple stemmer implementation in pure Python that can handle trailing S's:

```
>>> def stem(phrase):
...     return ' '.join([re.findall('^(.*ss|.*?)(s)?$',
...         word)[0][0].strip("'") for word in phrase.lower().split()])
>>> stem('houses')
'house'
>>> stem("Doctor House's calls")
'doctor house call'
```

The preceding stemmer function follows a few simple rules within that one short regular expression:

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

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

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

It also doesn't handing the "housing" example from your "Portland Housing" search.

Two of the most popular stemming algorithms are the Porter and Snowball stemmers. The Porter stemmer is named for the computer scientist Martin Porter.[15] Porter is also responsible for enhancing the Porter stemmer to create the Snowball stemmer.[16] Porter dedicated much of his lengthy career to documenting and improving stemmers, due to their value in information retrieval (keyword search). These stemmers implement more complex rules than our simple regular expression. This enables the stemmer to handle the complexities of English spelling and word ending rules:

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

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

[15] See "An algorithm for suffix stripping," 1993 (http://www.cs.odu.edu/~jbollen/IR04/readings/readings5.pdf) by M.F. Porter.

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

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

More on the Porter stemmer

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

There are eight steps to the Porter stemmer algorithm: 1a, 1b, 1c, 2, 3, 4, 5a, and 5b. Step 1a is a bit like your regular expression for dealing with trailing S's:[a]

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

This isn't at all like str.replace(). Julia's self.replace() modifies only the ending of a word.

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

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

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

[b] Sorry Chick, Porter doesn't like your `obsfucate` username ;).

LEMMATIZATION

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

In chapter 12, we show how you can use lemmatization to reduce the complexity of the logic required to respond to a statement with a chatbot. Any NLP pipeline that wants to "react" the same for multiple different spellings of the same basic root word

can benefit from a lemmatizer. It reduces the number of words you have to respond to, the dimensionality of your language model. Using it can make your model more general, but it can also make your model less precise, because it will treat all spelling variations of a given root word the same. For example "chat," "chatter," "chatty," "chatting," and perhaps even "chatbot" would all be treated the same in an NLP pipeline with lemmatization, even though they have different meanings. Likewise "bank," "banked," and "banking" would be treated the same by a stemming pipeline, despite the river meaning of "bank," the motorcycle meaning of "banked," and the finance meaning of "banking."

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

Lemmatization is a potentially more accurate way to normalize a word than stemming or case normalization because it takes into account a word's meaning. A lemmatizer uses a knowledge base of word synonyms and word endings to ensure that only words that mean similar things are consolidated into a single token.

Some lemmatizers use the word's part of speech (POS) tag in addition to its spelling to help improve accuracy. The POS tag for a word indicates its role in the grammar of a phrase or sentence. For example, the noun POS is for words that refer to "people, places, or things" within a phrase. An adjective POS is for a word that modifies or describes a noun. A verb refers to an action. The POS of a word in isolation cannot be determined. The context of a word must be known for its POS to be identified. So some advanced lemmatizers can't be run-on words in isolation.

Can you think of ways you can use the part of speech to identify a better "root" of a word than stemming could? Consider the word `better`. Stemmers would strip the "er" ending from "better" and return the stem "bett" or "bet." However, this would lump the word "better" with words like "betting," "bets," and "Bet's," rather than more similar words like "betterment," "best," or even "good" and "goods."

So lemmatizers are better than stemmers for most applications. Stemmers are only really used in large-scale information retrieval applications (keyword search). And if you really want the dimension reduction and recall improvement of a stemmer in your information retrieval pipeline, you should probably also use a lemmatizer right before the stemmer. Because the lemma of a word is a valid English word, stemmers work well on the output of a lemmatizer. This trick will reduce your dimensionality and increase your information retrieval recall even more than a stemmer alone.[17]

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

[17] Thank you Kyle Gorman for pointing this out.

```
>>> nltk.download('wordnet')
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> lemmatizer.lemmatize("better")
'better'
>>> lemmatizer.lemmatize("better", pos="a")
'good'
>>> lemmatizer.lemmatize("good", pos="a")
'good'
>>> lemmatizer.lemmatize("goods", pos="a")
'goods'
>>> lemmatizer.lemmatize("goods", pos="n")
'good'
>>> lemmatizer.lemmatize("goodness", pos="n")
'goodness'
>>> lemmatizer.lemmatize("best", pos="a")
'best'
```

The default part of speech is "n" for noun.

"a" indicates the adjective part of speech.

You might be surprised that the first attempt to lemmatize the word "better" didn't change it at all. This is because the part of speech of a word can have a big effect on its meaning. If a POS isn't specified for a word, then the NLTK lemmatizer assumes it's a noun. Once you specify the correct POS, "a" for adjective, the lemmatizer returns the correct lemma. Unfortunately, the NLTK lemmatizer is restricted to the connections within the Princeton WordNet graph of word meanings. So the word "best" doesn't lemmatize to the same root as "better." This graph is also missing the connection between "goodness" and "good." A Porter stemmer, on the other hand, would make this connection by blindly stripping off the "ness" ending of all words:

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

Use cases

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

If your application involves search, stemming and lemmatization will improve the recall of your searches by associating more documents with the same query words. However, stemming, lemmatization, and even case folding will significantly reduce the precision and accuracy of your search results. These vocabulary compression approaches will cause an information retrieval system (search engine) to return many documents not relevant to the words' original meanings. Because search results can be ranked according to relevance, search engines and document indexes often use

stemming or lemmatization to increase the likelihood that the search results include the documents a user is looking for. But they combine search results for stemmed and unstemmed versions of words to rank the search results that they present to you.[18]

For a search-based chatbot, however, accuracy is more important. As a result, a chatbot should first search for the closest match using unstemmed, unnormalized words before falling back to stemmed or filtered token matches to find matches. It should rank such matches of normalized tokens lower than the unnormalized token matches.

> **IMPORTANT** Bottom line, try to avoid stemming and lemmatization unless you have a limited amount of text that contains usages and capitalizations of the words you are interested in. And with the explosion of NLP datasets, this is rarely the case for English documents, unless your documents use a lot of jargon or are from a very small subfield of science, technology, or literature. Nonetheless, for languages other than English, you may still find uses for lemmatization. The Stanford information retrieval course dismisses stemming and lemmatization entirely, due to the negligible recall accuracy improvement and the significant reduction in precision.[19]

2.3 Sentiment

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

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

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

---

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

[19] See the web page titled "Stemming and lemmatization" (https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html).