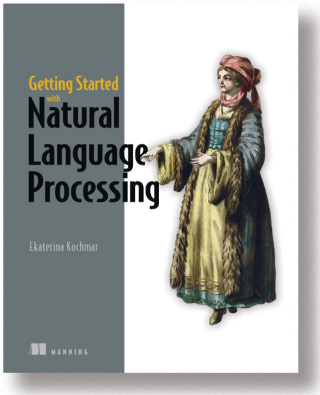
Natural Language Processing in Practice

Chapters selected by Ekaterina Kochmar

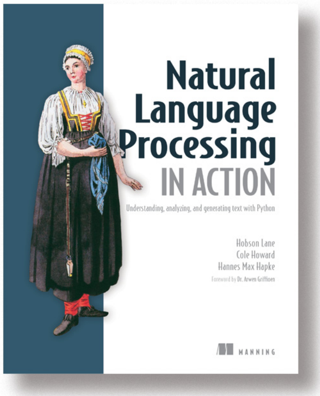
manning



Getting Started with Natural Language Processing

by Ekaterina Kochmar

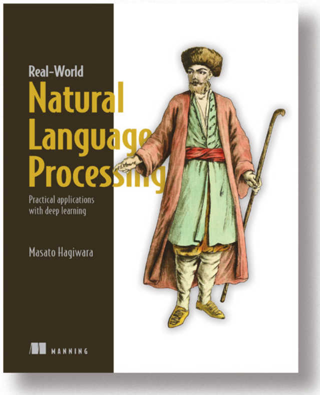
ISBN 9781617296765
325 pages
$31.99
Spring 2021


Natural
Language
Processing
IN ACTION
Understanding, analyzing, and generating text with Python
Hobson Lane
Cole Howard
Hannes Max Hapke
Foreword by Dr. Arwen Griffioen
MANNING

Natural Language Processing in Action

by Hobson Lane, Cole Howard, Hannes Hapke

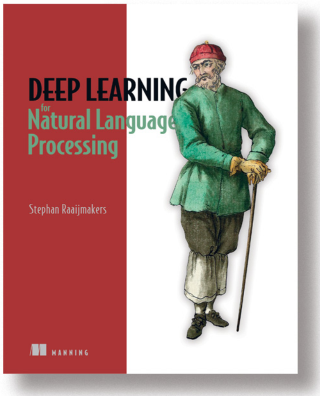
ISBN 9781617294631
544 pages
$39.99

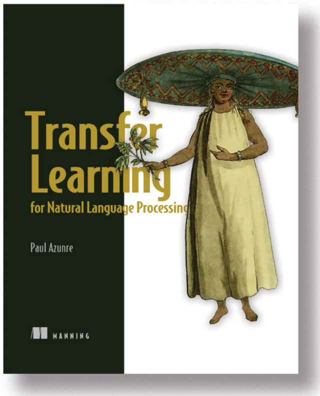


Real-World Natural Language Processing

by Masato Hagiwara

ISBN 9781617296420
500 pages
$47.99
Spring 2021


DEEP LEARNING
for
Natural Language
Processing
Stephan Raaijmakers
MANNING


Transfer
Learning
for Natural Language Processing
Paul Azunre
MANNING

Natural Language Processing in Practice

Chapters chosen by Ekaterina Kochmar

Manning Author Picks

To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Candace Gillhoolley, corp-sales@manning.com

♾ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617299360

contents

introduction

Language proficiency is a core metric of intelligence—not only in humans, but in machines as well. That's why the ability to efficiently and accurately process the information conveyed by human language is so important. Natural Language Processing (NLP) technology plays an integral role in many of the recent advances in Artificial Intelligence, so it's no surprise that it's earning more and more of the spotlight on the intelligent technology stage. And it's not only tech companies that are leveraging NLP; Finance, insurance, and other industries across the board are increasingly interested in the potential benefits NLP can offer. After all, language is the primary means of communication in all facets of life, and this rapidly advancing field promises to spawn countless new and exciting opportunities. There's really never been a better time to get started with NLP!

To jumpstart your NLP education, this mini ebook features two chapters from Manning's *Getting Started with Natural Language Processing*. In it, author and NLP expert Ekaterina Kochmar guides you step by step as you build two NLP applications from start to finish: one, a spam filter; the other, an information search algorithm; both, classic examples of NLP in practice that apply fundamental NLP skills and concepts. When you're done, you'll have valuable hands-on NLP experience that will help you hit the ground running as you continue your NLP learning.

Whether your interest in getting started with NLP is for your career, your personal passion project, or just for fun, as long as it involves textual information, you'll find great value in this quick and practical primer. All you need to benefit from these chapters is some basic Python programming skills. We hope the information here kindles your NLP curiosity even more and opens your eyes to its endless possibilities. If you'd like to dive deeper into this highly useful, interesting—and exploding!—field, we highly recommend the complete version of *Getting Started with Natural Language Processing*.

Chapter 2 from *Getting Started with Natural Language Processing*
by Ekaterina Kochmar

In this chapter, you'll build your own NLP application from beginning to end: a spam filter. You'll learn how to structure a typical NLP pipeline and apply a machine learning algorithm to solve your task as you implement this classic combination of NLP and machine learning.

2 Your first NLP example

This chapter covers:

- Implementing your first practical NLP application from scratch
- Structuring an NLP project from beginning to end
- Understanding useful NLP concepts, including tokenization and text normalization
- Applying a machine learning algorithm to textual data

In this chapter, you will learn how to implement your own NLP application from scratch. In doing so, you will also learn how to structure a typical NLP pipeline and how to apply a simple machine learning algorithm to solve your task. The particular application you will implement is *spam filtering*. We overviewed it in chapter 1 as one of the classic tasks on the intersection of NLP and machine learning.

2.1 Introducing NLP in practice: Spam filtering

In this book, you use the spam filtering as your first practical NLP application because it is an example of a very widely spread family of tasks: *text classification*. Text classification comprises a number of applications that we discuss in this book; for example, user profiling (chapter 5), sentiment analysis (chapter 6), and topic labeling (chapter 8), so this chapter will give you a good start for the rest of the book. First, let's see what exactly *classification* addresses.

We humans apply classification in our everyday lives pretty regularly: classifying things simply implies that we try to put them into clearly defined groups, classes, or categories. In fact, we tend to classify all sorts of things all the time. Here are some examples:

- Based on our level of engagement and interest in a movie, we may classify it as *interesting or boring.*
- Based on temperature, we classify water as *cold or hot.*
- Based on the amount of sunshine, humidity, wind strength, and air temperature, we classify the weather as *good or bad.*
- Based on the number of wheels, we classify vehicles into *unicycles, bicycles, tricycles, quadricycles, cars,* and so on.
- Based on the availability of the engine, we may classify two-wheeled vehicles into *bicycles and motorcycles.*

Figure 2.1 combines the two types of classification for vehicles into one illustration.

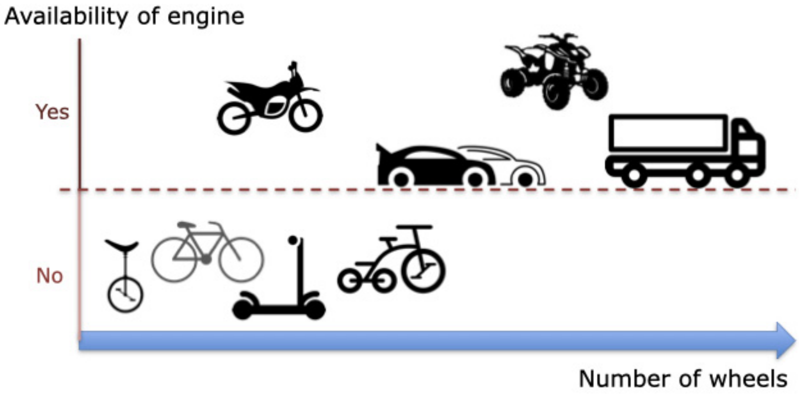


Figure 2.1 Classification of vehicles by two parameters: number of wheels and availability of an engine.

Classification is useful because it makes it easier for us to reason about things and adjust our behavior accordingly. For example, there might be more subtle characteristics to a movie than it being just interesting or just boring, but by defining our attitude toward a movie very concisely using these two categories, we might save a friend of ours (provided we have similar taste in movies!) a lot of time. By defining water as hot we know that we should be careful when we use it, without the need to think about what particular temperature it is and whether it is tolerable. Or take the different types of vehicles as an example: once we've done the grouping of vehicles, it becomes much easier to deal with any *instance* of each class. When we see any particular bicycle, we know what typical speed it can travel with and what types of actions can be performed with bicycles in general. We know what to expect and don't need to reconsider any of these facts for *each bicycle in question* because the class of bicycles defines the properties of each particular instance, too. We refer to the name of each class as a *class label.*

When classifying things, we often go for simple contrasts—good vs. bad, interesting vs. boring, hot vs. cold. When we are dealing with two labels only, this is called *binary classification.* For example, if we classify two-wheeled vehicles on the basis of whether they have an engine or not, we perform a binary classification and end up with two groups of objects—unmotorized two-wheeled vehicles like bicycles and kick-scooters, and motorized two-wheeled vehicles like electric bicycles, motorcycles, mopeds, and so on. But if we classify all vehicles based on the number of wheels, on size, or any other characteristics, we will end up with multiple classes; for example, two-wheeled unmotorized vehicles, two-wheeled motorized vehicles, three-wheeled unmotorized vehicles, and so on, as figure 2.2 illustrates. Classification that implies more than two classes is called *multi-class classification.*

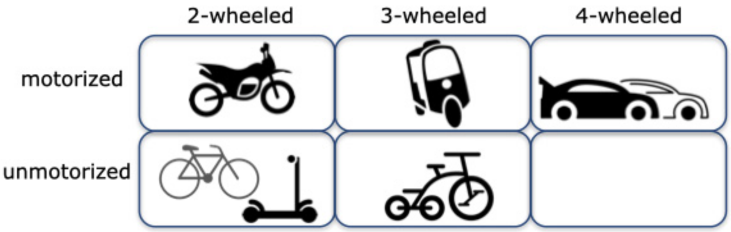


Figure 2.2 Multi-class classification of vehicles based on two parameters.

Finally, how do we actually perform classification? We rely on a number of characteristics of the classified concepts, which in some cases may include one type of information only. For example, to classify water into cold or hot, we may rely on a single value of water temperature, and, for example, call anything above 45°C (113°F) hot and anything below this value cold. The selection of such characteristics will depend on the particular task: for example, to classify weather into good or bad we may need to rely on a number of characteristics including air temperature, humidity, wind strength, and so on, rather than any single one. In machine learning terms, we call such characteristics *features.*

As we are used to classifying things on a regular basis, we can usually relatively easily define the number of classes, the labels and the features. This comes from our wide experience with classification and from our exposure to multiple examples of concepts from different classes. Machines can learn to classify things as well, with a little help from humans. Sometimes a simple rule would be sufficient. For example, you can make the machine print out a warning that water is hot based on a simple threshold of 45°C (113°F), as shown in the following listing.

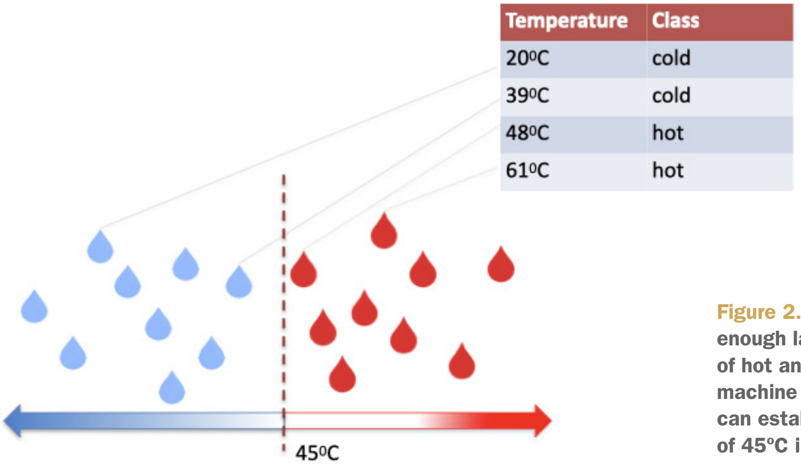
Listing 2.1 Simple code to tell whether water is cold or hot

```
def print_warning(temperature):
    if temperature>=45:
        print ("Caution: Hot water!")
    else:
        print ("You may use water as usual")
```

Define a simple function that takes water temperature as input and prints out water status.

B `If`-statement checks if temperature is above the threshold and prints out a warning message if it is.

However, when there are multiple factors to take into account, and these multiple factors may interact in various ways, a better strategy is to make the machine learn such rules and infer their correspondences from the data rather than hard-code them. After all, what the machines are good at is detecting the correspondences and patterns! This is what machine learning is about: it states that machines can learn to solve the task if they are provided with a sufficient number of examples and with the general outline of the task. For example, if we define the classes, labels, and features for the machine, it can then learn to assign concepts to the predefined classes based on these features. For the previous cold vs. hot water example, we can provide the machine with the samples of water labeled “hot” and samples of water labeled “cold”, tell it to use temperature as the predictive factor (feature), and this way let it learn independently from the provided data that the boundary between the two classes is around 45°C, as figure 2.3 shows. This type of machine learning approach, when we supervise the machine while it is learning by providing it with the labeled data, is called *supervised machine learning*.

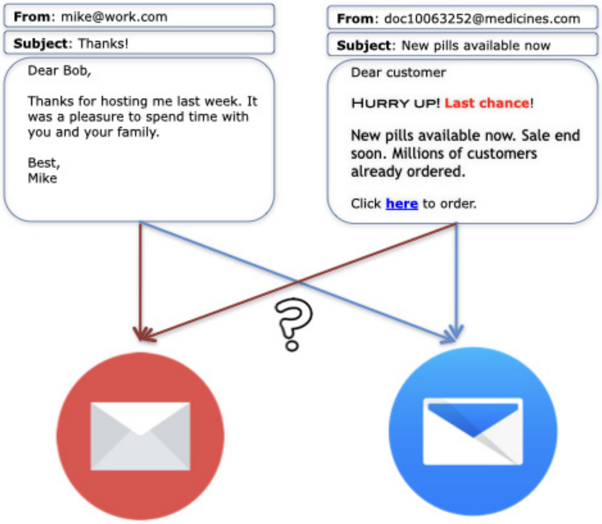


Temperature	Class
20ºC	cold
39ºC	cold
48ºC	hot
61ºC	hot

Figure 2.3 Provided with enough labeled examples of hot and cold water, the machine learning algorithm can establish the threshold of 45°C independently.

Now that you are familiar with the ideas behind classification tasks, you are all set to implement your first NLP classification algorithm in practice. Before you move on, test your understanding of the task with the exercise below.

Figure 2.4 shows examples of two emails.



From: mike@work.com

Subject: Thanks!

Dear Bob,

Thanks for hosting me last week. It was a pleasure to spend time with you and your family.

Best,
Mike

From: doc10063252@medicines.com

Subject: New pills available now

Dear customer

HURRY UP! Last chance!

New pills available now. Sale end soon. Millions of customers already ordered.

Click here to order.

Figure 2.4 Examples of two emails. Can you tell whether they should go to INBOX or SPAM box?

Exercise 1

Spam filtering is an example of text classification, which is usually addressed with supervised machine learning techniques.

1. What labels do we assign in the spam filtering task; for example, in the example above?
2. How many classes are there? What type of classification is this—binary or multi-class?
3. What features will help you distinguish between classes?

First, try solving this exercise yourself. Then compare your answers with the solution.

Solution:

1. We distinguish between *spam and normal* emails. Spam emails should end up in the spam box and normal emails should be kept in the INBOX.
2. This is an example of *binary classification* because we distinguish between two classes only.
3. We humans can relatively easily tell a spam email from a normal one, although some spammers use sophisticated techniques to disguise their intentions, and in some cases, it might be tricky to tell the difference. The format of the email (use of unusual fonts and colors), the information about the sender (unusual or unknown email address), and the list of addressees (spam emails are often mass emails), as well as attachments and links are all very indicative. However, some of the strongest clues are provided by the content of the email itself and

the language used: for example, you should be wary of emails that tell you that your account is unexpectedly blocked, that you need to provide sensitive personal information for suspicious reasons, or that you have won in a lottery, especially if you haven't participated in one!

2.2 Understanding the task

Scenario 1

You have a collection of spam and normal emails from the past. You are tasked with building a spam filter, which for any future incoming email can predict whether this email is spam or not.

- How can you use the provided data?
- What characteristics of the emails might be particularly useful and how will you extract them?
- What will be the sequence of steps in this application?

First, you need to ask yourself what format the email messages are delivered in for this task. For instance, in a real-life situation, you might need to extract the messages from the mail agent application. However, for simplicity, let's assume that someone has extracted the emails for you and stored them in text format. The normal emails are stored in a separate folder—let's call it "ham", and spam emails are stored in "spam" folder.[1] If someone has already pre-defined past spam and ham emails for you, for example, by extracting these emails from the INBOX and SPAM, you don't need to bother with labeling them yourself. However, you still need to point the machine-learning algorithm at the two folders by clearly defining which one of them is ham and which one is spam. This way, you will define the *class labels* and *identify the number of classes* for the algorithm. This should be the first step in your spam detection pipeline, after which you can preprocess the data, extract the relevant information, and then train and test your algorithm. In total, the pipeline will consist of five steps, visualized as a flow chart in figure 2.5.

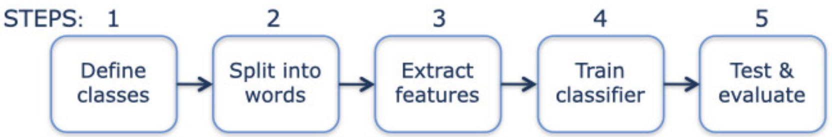
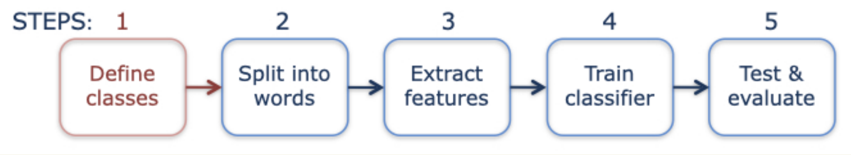


Figure 2.5 **Five steps of a machine-learning NLP project.**

[1] If you are wondering why "normal" emails are sometimes called "ham" in spam detection context, check out the history behind the term "spam" at https://en.wikipedia.org/wiki/Email_spam

Let's look into each of these steps in more detail. So you can set Step 1 of your algorithm as:

Step 1: Define which data represents "ham" class and which data represents "spam" class for the machine-learning algorithm.



You will need to define the *features* for the machine to know what type of information, or what properties of the emails to pay attention to, but before you can do that there is one more step to perform. As we've just discussed in the previous exercise, email content provides significant information as to whether an email is ham or spam. How can you extract the content? One solution would be to read in the whole email as a single textual property. For example, use "Minutes of the meeting on Friday, 20 June [. . .]"[2] as a single feature for "ham" emails, and "Low cost prescription medications [. . .]" as a single feature for "spam" emails. This will definitely help the algorithm identify the emails that contain *all* of the included phrases from these two emails as either "spam" or "ham", but how often would you expect to see precisely the same text of the email again? Any single character change may change the whole feature! This suggests that a better candidate for a feature in this task would be a smaller chunk of text, for example, a word. In addition, words are likely to carry spam-related information (for example, "lottery" might be a good clue for a spam email), while being repetitive enough to occur in multiple emails.

Exercise 2

For a machine, the text comes in as a sequence of symbols, so the machine does not have an idea of what a word is.

How would you define what a word is from the human perspective? How would you code this for a machine?

For example, how will you split a sentence "Define which data represents each class for the machine-learning algorithm" into words?

Solution:

The first solution that might come to your mind might be "Words are sequences of characters separated by whitespaces". This will work well for some examples, including:

[2] Assume the whole email instead of [. . .] here.

> Define which data represents each class for the machine-learning algorithm.

Let’s write simple code that uses whitespace to split text into words, as shown in the following listing.

Listing 2.2 Simple code to split text string into words by whitespaces

```
text = “Define which data represents each class for the machine-learning
     algorithm”
text.split(“ ”)
```

A You can rely on Python’s functionality to split strings of text by whitespaces.

This will split the sentence above into a list of words as [Define, which, data, represents, each, class, for, the, machine-learning, algorithm]. However, what happens to this strategy when we have punctuation marks? For example:

> Define which data represents “ham” class and which data represents “spam” class for the machine-learning algorithm.

Now you will end up with the words like [. . . , “ham”, . . . , “spam”, . . . , algorithm.] in the list of words. Are [“ham”], [“spam”] and [algorithm.] any different from [ham], [spam] and [algorithm]; that is, the same words but without the punctuation marks attached to them? The answer is, these words are exactly the same, but because you are only splitting by whitespaces at the moment, there is no way of taking the punctuation marks into account. However, each sentence will likely include one full stop (.), question (?) or exclamation mark (!) attached to the last word, and possibly more punctuation marks inside the sentence itself, so this is going to be a problem for extracting words from text properly. Ideally, you would like to be able to extract words and punctuation marks separately.

Taking this into account, you might update your algorithm with a splitting strategy by punctuation marks. There are several possible ways to do that, including using Python’s regular expressions module `re`.[3] However, if you have never used regular expressions before, you may apply a simple iterative algorithm that will consider each character in the text string and decide whether it should be ignored (if it is a whitespace), added to a word list (if it is a punctuation mark), or added to the current word (otherwise). In other words, the algorithm may proceed as follows:

[3] If you have never used `re` module and regular expressions before, you can find more information about it on https://docs.python.org/3/library/re.html

Algorithm 1

(1) Store words list and a variable that keeps track of the current word—let’s call it current_word for simplicity.

(2) Read text character by character, and:

(2.1) *if* a character is a whitespace, add the current_word to the words list and update the current_word variable to be ready to start a new word.

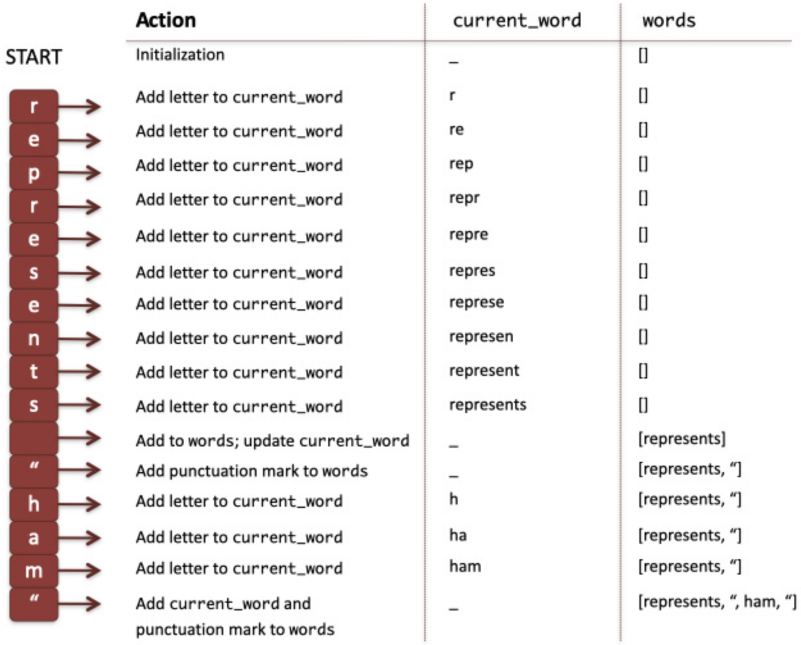
(2.2) *else if* a character is a punctuation mark, and:

(2.2.1) *if* the previous character is not a whitespace, add the current_word to the words list, then add the punctuation mark as a separate word token, and update the current_word variable.

(2.2.2) *else if* the previous character is a whitespace, just add the punctuation mark as a separate word token.

(2.3) *else if* a character is a letter other than a whitespace or punctuation mark, add it to the current_word.

Figure 2.6 shows how this algorithm will process the string ‘represents “ham”‘:



	Action	current_word	words
START	Initialization	_	[]
r	Add letter to current_word	r	[]
e	Add letter to current_word	re	[]
p	Add letter to current_word	rep	[]
r	Add letter to current_word	repr	[]
e	Add letter to current_word	repre	[]
s	Add letter to current_word	repres	[]
e	Add letter to current_word	represe	[]
n	Add letter to current_word	represen	[]
t	Add letter to current_word	represent	[]
s	Add letter to current_word	represents	[]
	Add to words; update current_word	_	[represents]
“	Add punctuation mark to words	_	[represents, “]
h	Add letter to current_word	h	[represents, “]
a	Add letter to current_word	ha	[represents, “]
m	Add letter to current_word	ham	[represents, “]
“	Add current_word and punctuation mark to words	_	[represents, “, ham, “]

Figure 2.6 Processing of the string ‘represents “ham”‘ with a tokenization algorithm.

And the following listing shows how you can implement this algorithm in Python.

Listing 2.3 Code to split text string into words by whitespaces and punctuation

Initialize a list of delimiters and populate it with some punctuation marks.

```
text = ‘Define which data represents "ham" class and which data represents
     "spam" class for the machine-learning algorithm.’
delimiters = [‘"‘, "."]
words = []
current_word = ""

for char in text:
    if char==" ":
        if not current_word=="":
            words.append(current_word)
            current_word = ""
    elif char in delimiters:
        if current_word=="":
            words.append(char)
        else:
            words.append(current_word)
            words.append(char)
            current_word = ""
    else:
        current_word += char

print(words)
```

Variable words will keep the list of processed words.

Variable current_word will keep track of the word currently being processed.

Iterate through text character by character.

E If the character is a whitespace and the current_word is not empty, add it to the words list and re-initialize current_word to keep track of the upcoming words.

If the character is one of the punctuation marks and there is nothing stored in the current_word yet, add this punctuation mark to the words list.

If the character is one of the punctuation marks and there is information stored in current_word, add both the current_word and the punctuation mark to the words list and re-initialize current_word to keep track of the upcoming words.

Otherwise, if the character is any other letter (not specified as a delimiter, and not a whitespace), add it to the current_word.

This code will work for the previous examples, but it will also split examples like “i.e.” and “e.g.” into [i, ., e, .] and [e, ., g, .], and “U.S.A.” and “U.K.” into [U, ., S, ., A, .] and [U, ., K, .]. This is problematic, because the algorithm will lose track of the correct interpretation of words like “i.e.” or “U.S.A.”, which should be treated as one word rather than a combination of characters. How can this be achieved?

This is where the NLP tools come in handy: the tool that helps you to split the running string of characters into meaningful words is called *tokenizer,* and it takes care of the cases like the ones we’ve just discussed. That is, it can recognize that “ham.” needs to be split into [ham, .] while “U.S.A.” needs to be kept as one word [U.S.A.]. Unlike the simpler solutions that you applied previously, tokenizers not only perform splitting by whitespaces and punctuation marks, but also keep track of the cases that should not be split by such methods. This helps make sure that the tokenization step results in a list of appropriate English words.

To check your understanding of what tokenization step achieves, try manually tokenizing strings of text in the following exercise before looking into the solution notes. Later, you will also be able to check whether your solutions coincide with those returned by a tokenizer.

Exercise 3

How will you tokenize the following strings into words?

(1) What’s the best way to split a sentence into words?

(2) We’re going to use NLP tools.

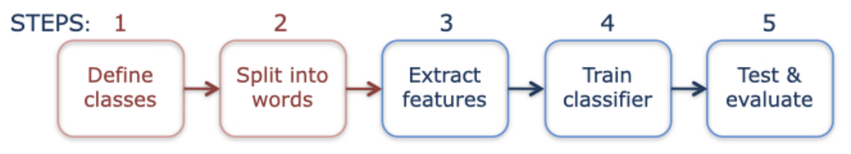
(3) I haven’t used tokenizers before.

Solution:

1. You already know that the punctuation marks should be treated as a separate word, so the last bit of text in the sentence “What’s the best way to split a sentence into words?” should be split into “words” and “?”. The first bit “What’s” should also be split into two words: this is a contraction for “what” and “is”, and it is important that the classifier knows that these two are separate words. Therefore, the word list for this sentence will include [What, ‘s, the, best, way, to, split, a, sentence, into, words, ?].
2. The second sentence “We’re going to use NLP tools.” similarly contains a full stop at the end that should be separated from the previous word, and “we’re” should be split into “we” and “‘re” (= “are”). Therefore, the full word list will be [We, ‘re, going, to, use, NLP, tools, .].
3. Follow the same strategy as before: the third sentence “I haven’t used tokenizers before.” will produce [I, have, n’t, used, tokenizers, before, .]. Note that the contraction of “have” and “not” here results in an apostrophe inside the word “not”, however you should still be able to recognize that the proper English words in this sequence are “have” and “n’t” (= “not”) rather than “haven” and “‘t”. This is what the tokenizer will automatically do for you.[4]

Now you can define Step 2 of your algorithm as:

Step 2: Apply tokenization to split the running text into words, which are going to serve as features.



[4] Note that the tokenizers do not automatically map contracted forms like “n't” and “'re” to full form like “not” and “are”—although such mapping would be useful in some cases, this is beyond the functionality of tokenizers.

Next, let’s look into the extracted words closely and see whether they are all equally good to be used as features, that is, whether they are equally indicative of the spam-related content. Suppose two emails use a different format. One says:

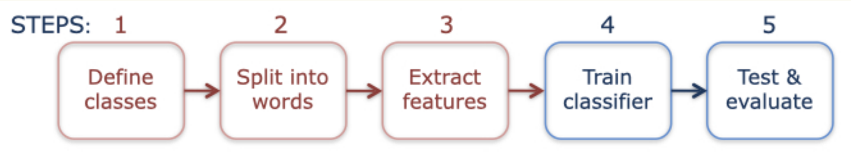
> Collect your lottery winnings.

while another one says

> Collect Your Lottery Winnings

The algorithm that splits these messages into words will end up with different word lists because, for instance, “lottery” ? ”Lottery”, but is it different in terms of the meaning? To get rid of such formatting issues like upper case vs. lower case you can put all the extracted words into lower case using Python functionality. Therefore, the third step in your algorithm should be defined as:

Step 3: Extract and normalize the features, for example, by putting all words to lower case.



At this point, you will end up with two sets of data—one linked to “spam” class and another one linked to “ham” class. Each data is preprocessed in the same way in steps 2 and 3, and the features are extracted. Next, you need to let the machine use this data to build the connection between the set of features (properties) that describe each type of email (spam or ham) and the labels attached to each type. Then, in step 4, the machine-learning algorithm tries to build a statistical model, a function, that helps it distinguish between the two classes. This is what happens during the learning (training) phase. Figure 2.7 is a refresher visualizing the training and test processes.

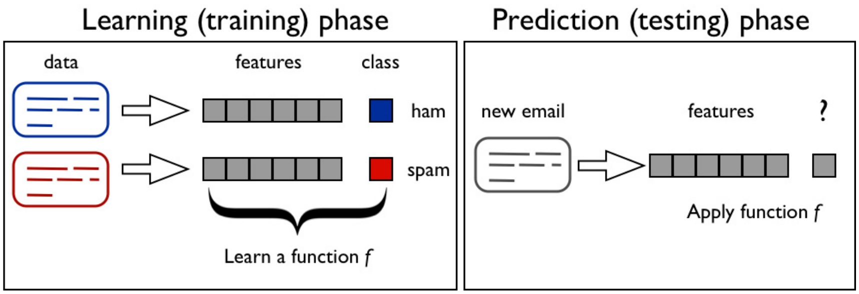
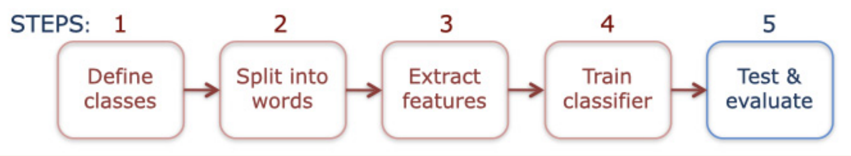


Figure 2.7 Learning (training) and prediction phases of spam filtering.

So, the next step of the algorithm should be defined as:

Step 4: Define the machine-learning model and train it on the data with the features predefined in the previous steps.



Your algorithm has now learned a function that can map the features from each class of emails to the "spam" and "ham" labels. During training, your algorithm will figure out which of the features matter more and should be trusted during prediction: for example, it might detect that occurrence of a word "lottery" in an email should be strongly associated with the label "spam", while occurrence of the word "meeting" should strongly suggest "ham" label. The final step in this process is to make sure the algorithm is doing such predictions well. How will you do that?

Remember that you were originally provided with a set of emails pre-labeled for you as "spam" and "ham". That means you know the correct answer for these emails. Why not use some of them to check how well your algorithm performs? In fact, this is exactly how it is done in machine learning: you use some of your labeled data to test classifier's performance—this bit of data is predictably called a *test set*. There is one caveat, though: if you've already used this data to train the classifier, that is, to let it figure out the correspondence between the features and the classes, it already knows the right answers. To avoid that, you need to make sure that the bit of data you used in Step 4 for training is separate and non-overlapping with the test set—this bit of data is called *training set*. Therefore, before training your classifier in Step 4 you need to split your full dataset into training and test sets. Here is the set of rules for that:

- Shuffle your data to avoid any bias.
- Split it randomly into a larger proportion for the training phase and set the rest aside for the test phase. The typical proportions for the sets are 80% to 20%.
- Train your classifier in Step 4 using training set *only*. Your test set is there to provide you with a realistic and fair estimate of your classifier's performance, so don't let your classifier peek into it. Use it at the final step for evaluation only.

Figure 2.8 visualizes these steps.

Suppose you trained your classifier in Step 4, and then applied it to the test data. How will you measure the performance? One approach would be to check what proportion of the test emails the algorithm classifies correctly, that is, assigns a "spam" label to a spam email, and classifies ham emails as "ham". This proportion is called *accuracy*, and its calculation is pretty straightforward:

```
Accuracy = (number_of_correct_predictions)/(number_of_all_test_instances)
```

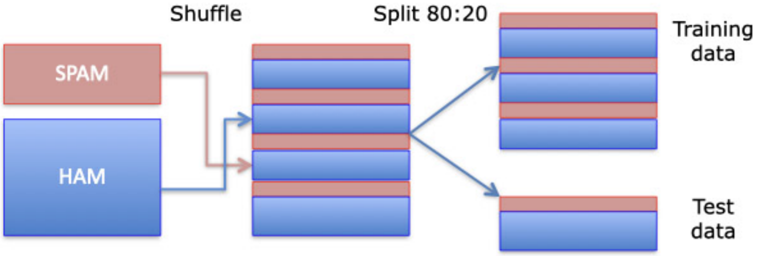


Figure 2.8 **Before training the classifier, shuffle the data and split it into training and test sets.**

Now check your understanding with the following exercise.

Exercise 4

Suppose your algorithm predicts the following labels for some small dataset of test examples:

Correct label	*Predicted label*
Spam	Ham
Spam	Spam
Ham	Ham
Ham	Spam
Ham	Ham

1. What is the accuracy of your classifier on this small dataset?
2. Is this a good accuracy, that is, does it suggest that the classifier performs well? What if you know that the ratio of ham to spam emails in your set of emails is 50%-50%? What if it is 60% ham emails and 40% spam—does it change your assessment of how well the classifier performs?
3. Does it perform better in identifying ham emails or spam emails?

Solution:

1. Using the formula, we can estimate that the accuracy of this algorithm is 3/5, or 60%: it got 3 out of 5 examples correctly (spam-spam, ham-ham, and ham-ham), and it made 2 mistakes mislabeling one spam email as "ham", and one ham email as "spam".
2. An accuracy of 60% seems to not be very high, but how exactly can you interpret it? Note that the distribution of classes helps you to put the performance of your classifier in context because it tells you how challenging the problem itself is. For example, with the 50%-50% split, there is no majority class in the data and the classifier's random guess will be at 50%, so the classifier's accuracy is higher than this random guess. In the second case, however, the classifier performs on a par with a majority class guesser: the 60% to 40% distribution of

classes suggests that if some dummy "classifier" always selected the majority class, it would get 60% of the cases correctly—just like the classifier you trained.

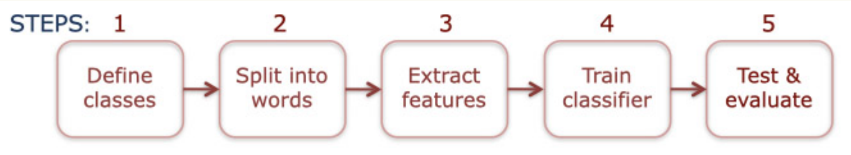
3. The single accuracy value of 60% does not tell you anything about the performance of the classifier on each class, so it is a bit hard to interpret. However, if you look into each class separately, you can tell that the classifier is better at classifying ham emails (it got 2/3 of those right) than at classifying spam emails (only 1/2 are correct).

The prediction of the classifier based on the distribution of classes that you came across in this exercise is called *baseline*. In an equal class distribution case, the baseline is 50%, and if your classifier yields an accuracy of 60%, it outperforms this baseline. In the case of 60%:40% split, the baseline, which can also be called the *majority class baseline*, is 60%. This means that if a dummy "classifier" does no learning at all and simply predicts "ham" label for all emails, it will not filter out any spam emails from the inbox, but its accuracy will also be 60%—just like your classifier that is actually trained and performs some classification! This makes the classifier in the second case in this exercise much less useful because it does not outperform the majority class baseline.

In summary, accuracy is a good overall measure of performance, but you need to keep in mind: (1) the distribution of classes to have a comparison point for the classifier's performance, and (2) the performance on each class which is hidden within a single accuracy value but might suggest what the strengths and weaknesses of your classifier are.

Therefore, the final step in your algorithm is:

Step 5: Apply your classifier to the test data and evaluate its performance.



2.3 Implementing your own spam filter

Now let's implement each of the five steps. It's time you open Jupyter and create a new notebook to start coding your own spam filter.

2.3.1 Step 1: Define the data and classes

Quite often when working on NLP and machine learning applications, you might find out that the problem has been previously described or someone has already collected some data that you may use to build an initial version of your algorithm. For example, if you want to build a machine learning classifier for spam detection, you need to provide your algorithm with a sufficient number of spam and ham emails. The best way to build such a classifier would be to collect your own ham and spam emails and train your algorithm to detect *what you personally would consider spam*—that would make your classifier personalized and tuned toward your needs, because you might consider certain content spam even when other users might see it as a harmless although unsolicited email. However, if you don't have enough examples in your own spam box (for instance, some mail agents automatically empty spam folders on a regular basis), there are collections of spam and ham emails collected from other users that you can use to train your classifier.

One of such publicly available collections is Enron email dataset.[5] This is a large dataset of emails (the original dataset contains about 0.5M messages), including both ham and spam emails, for about 150 users, mostly senior management of Enron.[6] To make processing more manageable, we are going to use a subset of this large dataset, although you can use the full dataset later if you wish. For your convenience, this subset is available together with the code for the book.[7] We are going to use enron1/ folder for training. All folders in Enron dataset contain spam and ham emails in separate subfolders, so you don't need to worry about pre-defining them. Each email is stored as a text file in these subfolders. Let's read in the contents of these text files in each subfolder, store the spam emails contents and the ham emails contents as two separate data structures and point our algorithm at each, clearly defining which one is spam and which one is ham.

To that end, let's define a function `read_in` that will take a folder as an input, read the files in this folder and store their contents as a Python list data structure:

Listing 2.4 Code to read in the contents of the files

```
import os
import codecs

def read_in(folder):
    files = os.listdir(folder)
    a_list = []
    for a_file in files:
        if not a_file.startswith("."):
            f = codecs.open(folder + a_file,
                "r", encoding = "ISO-8859-1", errors="ignore")
            a_list.append(f.read())
            f.close()
    return a_list
```

Import Python's os module that helps iterating through the folders.

Import Python's codecs module that helps with different text encodings.

Using os functionality, list all the files in the specified folder.

Iterate through the files in the folder.

Skip hidden files, that are sometimes automatically created by the operating systems. They can be easily identified because their names start with "."

Read the contents of each file. The encoding and errors arguments of codecs.open function will help you avoid errors in reading files that are related to text encoding.

Add the content of each file to the list data structure.

Don't forget to close the file after you've read the contents.

Return Python list that contains the contents of the files from the specified folder.

[5] You can read more about the dataset on this webpage https://www.cs.cmu.edu/~enron/, and download the subsets of the data here: http://nlp.cs.aueb.gr/software_and_datasets/Enron-Spam/index.html. The subsets are described in more detail in V. Metsis, I. Androutsopoulos and G. Paliouras, "Spam Filtering with Naive Bayes—Which Naive Bayes?". Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS 2006), Mountain View, CA, USA, 2006.

[6] See https://en.wikipedia.org/wiki/Enron.

[7] The dataset and the code are available at https://github.com/ekochmar/Essential-NLP.

Now you can define two such lists—spam_list and ham_list, letting the machine know what data to use as examples of spam emails and what data represents ham emails. Let's check if the data is uploaded correctly: for example, you can print out the lengths of the lists or check any particular member of the list. Since you are using publicly available dataset, you can easily check whether what your code put into the lists is correct: the length of the spam_list should equal the number of spam emails in the enron1/spam/ folder, which should be 1500, while the length of the ham_list should equal the number of emails in the enron1/ham/, or 3672. If you get these numbers, your data is uploaded and read in correctly. Similarly, you can check the contents of the very first instance in the spam_list and verify that it is exactly the same as the content of the first text file in the enron1/spam/ folder, as shown in the following listing.

Listing 2.5 Code to verify that the data is uploaded and read in correctly

```
spam_list = read_in("enron1/spam/")
ham_list = read_in("enron1/ham/")
print(len(spam_list))
print(len(ham_list))
print(spam_list[0])
print(ham_list[0])
```

Initialize spam_list and ham_list.

Check the lengths of the lists: for spam it should be 1500 and for ham—3672.

Print out the contents of the first entry. In both cases, it should coincide with the contents of the first file in each correspondent subfolder.

Next, you'll need to preprocess the data (for example, by splitting text strings into words) and extracting the features. Won't it be easier if you could run all preprocessing steps over a single data structure rather than over two separate lists? The listing 2.6 shows how you can merge the two lists together keeping their respective labels. This time, instead of using for-loop, let's use the compact code style that is provided by Python's list comprehensions.[8] Instead of lengthy for-loops that do updates to the lists, we are going to update list contents as we go. Finally, remember that you will need to split the data randomly into the training and test sets. Let's shuffle the resulting list of emails with their labels, and make sure that the shuffle is reproducible by fixing the way in which the data is shuffled, as shown in the following listing.

Listing 2.6 Code to combine the data into a single structure

```
import random

all_emails = [(email_content, "spam") for email_content in spam_list]
all_emails += [(email_content, "ham") for email_content in ham_list]
random.seed(42)
random.shuffle(all_emails)
print (f"Dataset size = {str(len(all_emails))} emails")
```

Python's random module will help you shuffle the data randomly.

Let's use list comprehensions to create all_emails list that will keep all emails with their labels: for each member of the ham_list and spam_list it stores a tuple with the content and associated label.

By defining the seed for the random operator, you can make sure that all future runs will shuffle the data in the same way.

You can check the size of the dataset (length of the list) —it should be equal to 1500 + 3672[9].

[8] Refresher on Python's list comprehensions: https://docs.python.org/3/tutorial/datastructures.html.

[9] This kind of strings is called *formatted string literals* or *f-strings*, and it is a new feature in Python 3.6. If you are unfamiliar with this type of string literals, you can check Python documentation at https://docs.python.org/3/reference/lexical_analysis.html#f-strings.

2.3.2 Step 2: Split the text into words

Remember, that the email contents that you’ve read in so far each come as a single string of symbols. The first step of text preprocessing involves splitting the running text into words.

Several NLP toolkits will be introduced in this book. One of them, Natural Language Processing Toolkit, or NLTK for short, you are going to start using straight away.[10] One of the benefits of this toolkit is that it comes with a thorough documentation and description of its functionality.

You are going to use NLTK’s tokenizer.[11] It takes running text as input and returns a list of words based on a number of customized regular expressions, which help to delimit the text by whitespaces and punctuation marks, keeping common words like “U.S.A.” unsplit. The code in listing 2.7 shows how to import the toolkit and the tokenizer and run it over the examples you’ve looked into in this chapter.

Listing 2.7 Code to run a tokenizer over text

```
import nltk
from nltk import word_tokenize

def tokenize(input):
    word_list = []
    for word in word_tokenize(input):
        word_list.append(word)
    return word_list

input = “What’s the best way to split a sentence into words?”
print(tokenize(input))
```

Import nltk library, and specifically import NLTK’s word tokenizer.

Let’s define a function tokenize that will take a string as input and split it into words.

This loop appends each identified word from the tokenized string to the output word list. Can you see how to present this code in a more compact and elegant way using list comprehensions?

Given the input, the function prints out a list of words. You can test your intuitions about the words and check your answers to previous exercises by changing the input to any string of your choice!

If you run the code from listing 2.7 on the suggested example, it will print out [‘What’, ‘‘s’, ‘the’, ‘best’, ‘way’, ‘to’, ‘split’, ‘a’, ‘sentence’, ‘into’, ‘words’, ‘?’] as the output.

2.3.3 Step 3: Extract and normalize the features

Once the words are extracted from running text, you need to convert them into features. In particular, you need to put all words into lower case to make your algorithm establish the connection between different formats like “Lottery” and “lottery”.

Putting all strings to lower case can be achieved with Python’s string functionality. To extract the features (words) from the text, you need to iterate through the recognized words and put all words to lower case. In fact, both tokenization and

[10] Install the toolkit from https://www.nltk.org/install.html.

[11] Check the documentation at https://www.nltk.org/api/nltk.tokenize.html.

converting text to lower case can be achieved using a single line of code with list comprehensions. See if you can come up with this line of code before you look at the next code listing.

Listing 2.8 Code to extract the features

Let's define a function that will extract the features from the text of email passed in as input.

You can combine the two steps—tokenization and converting strings to lower case—in one line using list comprehensions. Compare this to a much longer piece of code that performs tokenization in listing 2.7.

```
def get_features(text):
    features = {}
    word_list = [word for word in word_tokenize(text.lower())]
    for word in word_list:
        features[word] = True
    return features

all_features = [(get_features(email), label)
       for (email, label) in all_emails]

print(get_features("Participate In Our New Lottery NOW!"))
print(len(all_features))
print(len(all_features[0][0]))
print(len(all_features[99][0]))
```

For each word in the email let's switch on the "flag" that this word is contained in the email.

You can check what features are extracted from an input text.

The list data structure all_features will keep tuples containing the list of features matched with the "spam" or "ham" label for each email.

You can also check what all_features list data structure contains, for example, by printing out its length and the number of features detected in the first or any other email in the set.

With this bit of code, you iterate over the emails in your collection (all_emails) and store the list of features extracted from each email matched with the label. For example, if a spam email consists of a single sentence "Participate In Our New Lottery NOW!" your algorithm will first extract the list of features present in this email and assign a 'True' value to each of them. The list of features will be represented using the following format: ['participate': True, 'in': True, . . . , 'now': True, '!': True]. Then, the algorithm will add this list of features to `all_features` together with the "spam" label, that is, (['participate': True, 'in': True, . . . , 'now': True, '!': True], "spam"). Figure 2.9 visualizes the steps performed in this code listing.

Now check your understanding of the data processing with the following exercise.

Exercise 5

Imagine your whole dataset contained only one spam text "Participate In Our New Lottery NOW!" and one ham text "Participate in the Staff Survey". What features will be extracted from this dataset with the code from listing 2.8?

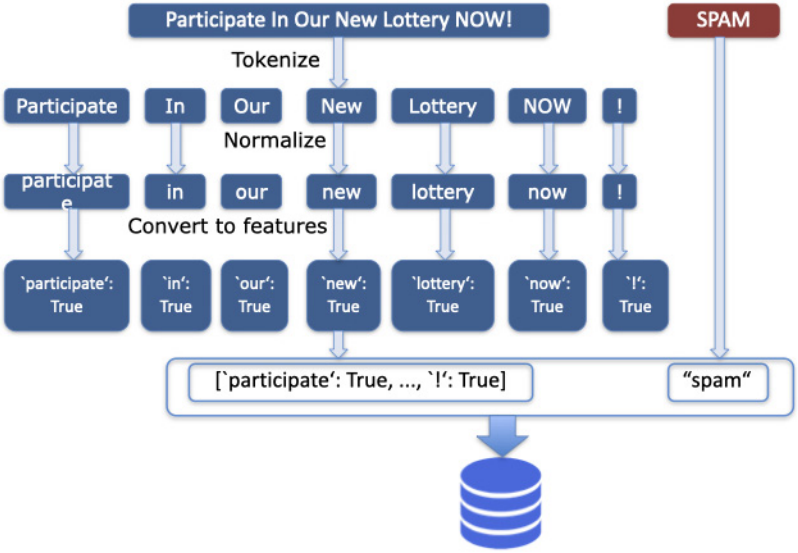
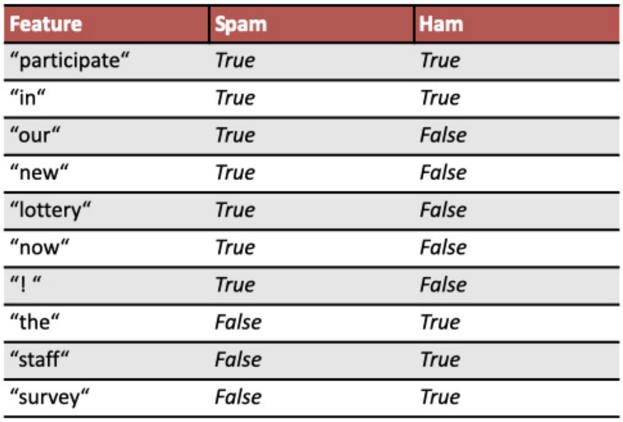


Figure 2.9 Preprocessing and feature extraction steps.

Solution:
You will end up with the following feature set.



Feature	Spam	Ham
“participate“	*True*	*True*
“in“	*True*	*True*
“our“	*True*	*False*
“new“	*True*	*False*
“lottery“	*True*	*False*
“now“	*True*	*False*
“! “	*True*	*False*
“the“	*False*	*True*
“staff“	*False*	*True*
“survey“	*False*	*True*

Again, it is a good idea to make sure you know how your data is represented, so the code in listing 2.8 uses print function to help you check some parameters of your data: for example, you can check how many emails have been processed and put into the feature list (this number should be equal to the number of emails you have started with), as well as the number of features present in each email (that is, with the ‘True’ flag assigned to them). The data structure that you have just created with the code from Listing 2.8, `all_features`, is a list of tuples (pairs), where each tuple represents an individual email, so the total length of `all_features` is equal to the number of emails in your dataset.

As each tuple in this list corresponds to an individual email, you can access each one of them by the index in the list using `all_features[index]`: for example, you can access the first email in the dataset as `all_features[0]` (remember, that Python's indexing starts with 0), and the 100th as `all_features[99]` (for the same reason).

Let's now clarify what each tuple structure representing an email contains. Tuples pair up two information fields: in this case a list of features extracted from the email and its label; that is, each tuple in `all_features` contains a pair (*list_of_features, label*). So if you'd like to access first email in the list, you call on `all_features[0]`, to access its list of features you use `all_features[0][0]`, and to access its label you use `all_features[0][1]`. Figure 2.10 visualizes the `all_features` data structure and the extraction process.

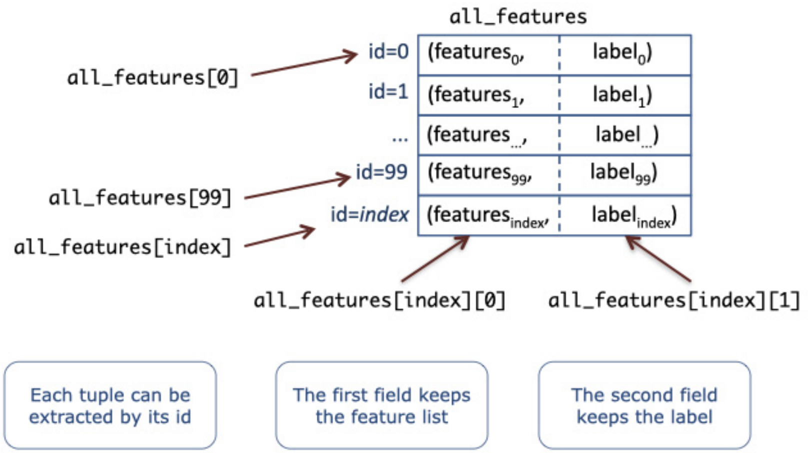


Figure 2.10 The all_features data structure.

For example, if the very first email in your collection is a spam email with the content "Participate in our lottery now!", `all_features[0]` will return the tuple (['participate': True, 'in': True, …, 'now': True, '!': True], "spam"), `all_features[0][0]` will return the list ['participate': True, 'in': True, …, 'now': True, '!': True], and `all_features[0][1]` will return the value "spam".

2.3.4 Step 4: Train the classifier

Next, let's apply machine learning and teach the machine to distinguish between the features that describe each of the two classes. There are a number of classification algorithms that you can use, and you will come across many of them in this book. But since you are at the beginning of your journey, let's start with one of the most interpretable ones—an algorithm called Naïve Bayes. Don't be misled by the word "Naïve" in its title, though: despite relative simplicity of the approach compared to other ones, this algorithm often works well in practice and sets a competitive performance baseline that is hard to beat with more sophisticated approaches. For the spam filtering algorithm that you are building in this chapter, you will rely on the Naïve Bayes implementation provided with the NLTK library, so don't worry if some details of the algorithm seem challenging to you. However, if you would like to see what is happening "under the hood", this section will walk you through the details of the algorithm.

Naïve Bayes is a *probabilistic* classifier, which means that it makes the class prediction based on the estimate of which outcome is most likely. That is, it assesses the probability of an email being spam and compares it with the probability of it being ham, and then selects the outcome that is *most probable* between the two. In fact, this is quite similar to how humans assess whether an email is spam or ham: when you receive an email that says, "Participate in our lottery now! Click on this link", before clicking on the (potentially harmful) link you assess *how likely* (that is, *what is the probability*) that it is a ham email and compare it to how likely it is that this email is spam. Based on your previous experience and all the previous spam and ham emails you have seen before, you might judge that it is *much more likely* (*more probable*) that it is a spam email. By the point the machine makes a prediction, it has also accumulated some experience in distinguishing spam from ham that is based on processing a dataset of labeled spam and ham emails.

Now let's formalize this step a bit further. In the previous step, you extracted the content of the email and converted it into a list of individual words (features). In this step, the machine will try to predict whether the email content represents spam or ham. In other words, it will try to predict whether the email is spam or ham *given* or *conditioned on* its content. This type of probability, when the outcome (class of "spam" or "ham") depends on the condition (words used as features), is called *conditional probability*. For spam detection, you estimate `P(spam | email content)` and `P(ham | email content)`, or generally `P(outcome | (given) condition)`.[12] Then you compare one estimate to another and return the most probable class. For example:

```
If P(spam | content) = 0.58 and P(ham | content) = 0.42, predict spam
If P(spam | content) = 0.37 and P(ham | content) = 0.63, predict ham
```

In summary, this boils down to the following set of actions illustrated in figure 2.11.

How can you estimate these probabilities in practice? Your own prediction of whether an email content like "Participate in our lottery now!" signifies that an email is spam or ham is based on how often in the past an email with the same content was spam or ham. Similarly, a machine can estimate the probability that an email is spam or ham conditioned on its content taking the number of times it has seen this content leading to a particular outcome. That is:

```
P(spam | "Participate in our lottery now!") = (number of emails "Participate
in our lottery now!" that are spam) / (total number of emails "Participate in
our lottery now!", either spam or ham)
```

[12] Reminder on the notation: `P` is used to represent all probabilities, `|` is used in conditional probabilities, when you are trying to estimate the probability of some event (that is specified before `|`) *given* that the condition (that is specified after `|`) applies.

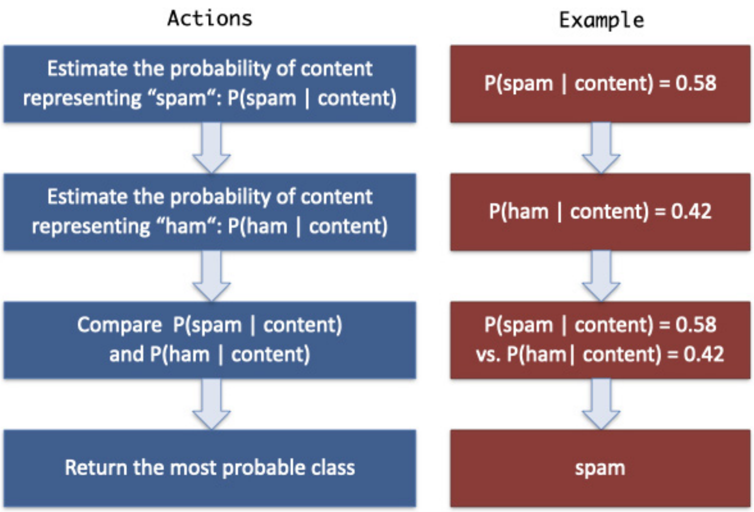

Actions
Example
Estimate the probability of content representing "spam": P(spam | content)
P(spam | content) = 0.58
Estimate the probability of content representing "ham": P(ham | content)
P(ham | content) = 0.42
Compare P(spam | content) and P(ham | content)
P(spam | content) = 0.58 vs. P(ham| content) = 0.42
Return the most probable class
spam

Figure 2.11 **Prediction is based on which conditional probability is higher.**

```
P(ham | "Participate in our lottery now!") = (number of emails "Participate
in our lottery now!" that are ham) / (total number of emails "Participate in
our lottery now!", either spam or ham)
```

In the general form, this can be expressed as:

```
P(outcome | condition) = number_of_times(condition led to outcome) /
number_of_times(condition applied)
```

You (and the machine) will need to make such estimations for all types of content in your collection, including for the email contents that are much longer than "Participate in our new lottery now!". Do you think you will come across enough examples to make such estimations? In other words, do you think you will see any particular combination of words (that you use as features), no matter how long, multiple times so that you can reliably estimate the probabilities from these examples? The answer is, you will probably see "Participate in our new lottery now!" only a few times, and you might see longer combinations of words only once, so such small numbers won't tell the algorithm much, and you won't be able to use them in the previous expression effectively. Additionally, you will constantly be getting new emails where the words will be used in different order and different combinations, so for some of these new combinations you will not have any counts at all, even though you might have counts for individual words in such new emails. The solution to this problem is to split the estimation into smaller bits. For instance, remember that you used tokenization to split long texts into separate words to let the algorithm access the smaller bits of information—words rather than whole sequences. The idea of estimating probabilities based on separate features rather than based on the whole sequence of features (whole text) is somewhat similar.

At the moment you are trying to predict a single outcome (class of spam or ham) given a single condition that is the whole text of the email, for example “Participate in our lottery now!”. In the previous step, you converted this single text into a set of features as [‘participate’: True, ‘in’: True, . . . , ‘now’: True, ‘!’: True]. Note that the conditional probabilities like `P(spam|` “Participate in our lottery now!”) and `P(spam|` [‘participate’: True, ‘in’: True, . . . , ‘now’: True, ‘!’: True]) are the same because this set of features encodes the text. Therefore, if the chances of seeing “Participate in our lottery now!” are low, the chances of seeing the set of features [‘participate’: True, ‘in’: True, . . . , ‘now’: True, ‘!’: True] encoding this text are equally low. Is there a way to split this set to get at more fine-grained, individual probabilities; for example, to establish a link between [‘lottery’: True] and the class of “spam”?

Unfortunately, there is no way to split the conditional probability estimation like `P(outcome | conditions)` when there are multiple conditions specified; however, it is possible to split the probability estimation like `P(outcomes | condition)` when there is a single condition and multiple outcomes. In spam detection, the class is a single value (it is “spam” or “ham”), while features are a set ([‘participate’: True, ‘in’: True, …, ‘now’: True, ‘!’: True]). If you can flip around the single value of class and the set of features in such a way that the class becomes the *new condition* and the features become the *new outcomes*, you can split the probability into smaller components and establish the link between individual features like [‘lottery’: True] and class values like “spam”. Figure 2.12 visualizes this idea.

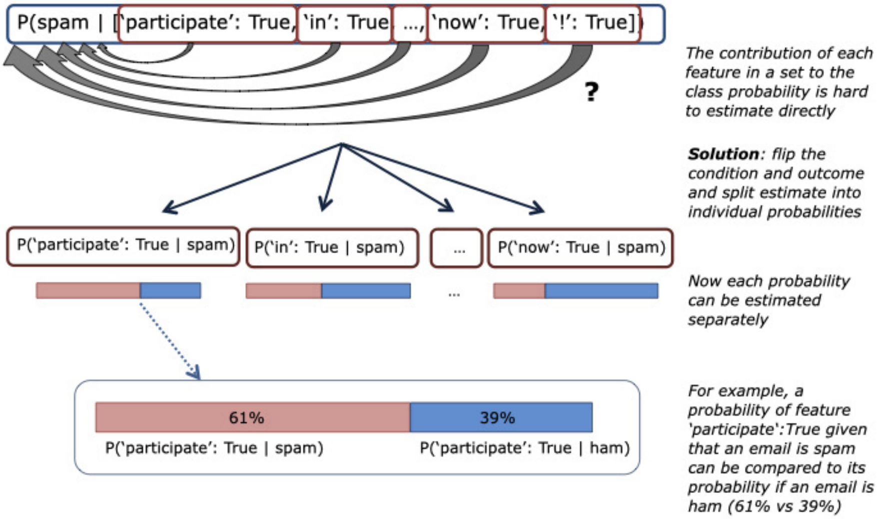

P(spam | [‘participate’: True, ‘in’: True, …, ‘now’: True, ‘!’: True])
?
The contribution of each feature in a set to the class probability is hard to estimate directly
Solution: flip the condition and outcome and split estimate into individual probabilities
P(‘participate’: True | spam)
P(‘in’: True | spam)
…
P(‘now’: True | spam)
…
Now each probability can be estimated separately
61%
39%
P(‘participate’: True | spam)
P(‘participate’: True | ham)
For example, a probability of feature ‘participate’:True given that an email is spam can be compared to its probability if an email is ham (61% vs 39%)

Figure 2.12 Since the conditional probability of class given a whole set of features is hard to estimate directly, flip the condition and outcome around and estimate the probabilities separately.

Luckily, there is a way to flip the outcomes (class) and conditions (features extracted from the content) around! Let's look into the estimation of conditional probabilities again: you estimate the probability that the email is spam *given* that its content is, "Participate in our new lottery now!" based on how often in the past an email with such content was spam. For that, you take the proportion of the times you have seen "Participate in our new lottery now!" in a spam email among the emails with this content. So you can express it as:

```
P(spam | “Participate in our new lottery now!”) =
P(“Participate in our new lottery now!” is used in a spam email) /
P(“Participate in our new lottery now!” is used in an email)
```

Let's call this Formula 1. What is the conditional probability of the content "Participate in our new lottery now!" *given* class spam then? Similarly, to how you estimated the probabilities above, you need the proportion of times you have seen "Participate in our new lottery now!" in a spam email among all spam emails. So you can express it as:

```
P(“Participate in our new lottery now!” | spam) =
P(“Participate in our new lottery now!” is used in a spam email) /
P(an email is spam)
```

Let's call this Formula 2. That is, every time you use conditional probabilities, you need to divide how likely it is that you see the condition and outcome together by how likely it is that you see the condition on its own—this is the bit after |. Now you can see that both Formulas 1 and 2 rely on how often you see particular content in an email of particular class. They share this bit, so you can use it to connect the two formulas. For instance, from Formula 2 you know that:

```
P(“Participate in our new lottery now!” is used in a spam email) =
P(“Participate in our new lottery now!” | spam) * P(an email is spam)
```

Now you can fit this into Formula 1:

```
P(spam | “Participate in our new lottery now!”) =
P(“Participate in our new lottery now!” is used in a spam email) /
P(“Participate in our new lottery now!” is used in an email) =
[P(“Participate in our new lottery now!” | spam) * P(an email is spam)] /
P(“Participate in our new lottery now!” is used in an email)
```

Figure 2.13 illustrates this process.
In the general form:

```
P (class | content) = P(content represents class) / P(content)
                    = [P(content | class) * P(class)] / P(content)
```

In other words, you can express the probability of a class given email content via the probability of the content given the class. Let's look into these two new probabilities, `P(content | class)` and `P(class)`, more closely as they have interesting properties:

- `P(class)` expresses the probability of each class. This is simply the distribution of the classes in your data. Imagine opening your inbox and seeing a new incoming email. What do you expect this email to be—spam or ham? If you mostly get normal emails and your spam filter is working well, you would most

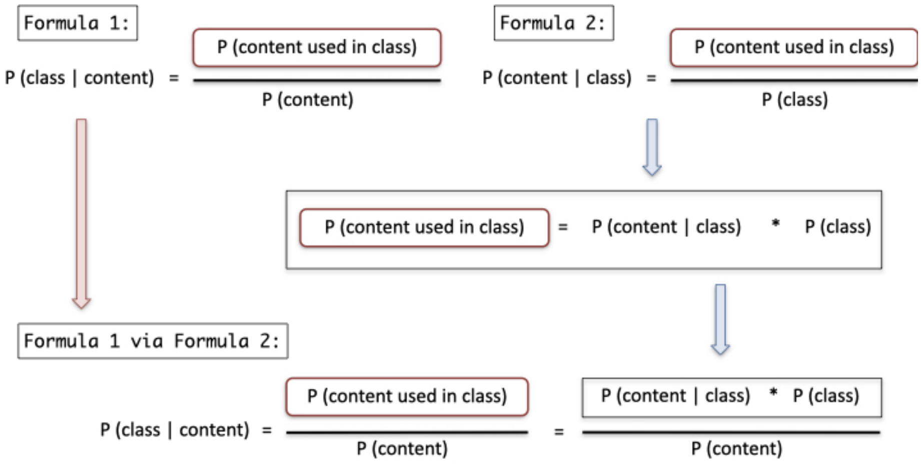

Formula 1:
P (class | content) =
P (content used in class)
P (content)
Formula 2:
P (content | class) =
P (content used in class)
P (class)
P (content used in class) = P (content | class) * P (class)
Formula 1 via Formula 2:
P (class | content) =
P (content used in class)
P (content)
=
P (content | class) * P (class)
P (content)

Figure 2.13 **The conditional probability for P(class | content) can be expressed via the conditional probability for P(content | class).**

probably expect a new email to also be ham rather than spam. For example, in enron1/ ham folder contains 3672 emails, and spam folder contains 1500 email, making the distribution approximately 71%:29%, or P(“ham”)=0.71 and P(“spam”)=0.29. This is often referred to as *prior* probability, because it reflects the beliefs of the classifier about where the data comes from *prior* to any particular evidence: for example, here the classifier will expect that it is much more likely (chances are 71 to 29) that a random incoming email is ham.

- `P(content | class)` is the *evidence*, as it expresses how likely it is that you (or the algorithm) will see this particular content given that the email is spam or ham. For example, imagine you have opened this new email and now you can assess how likely it is that these words are used in a spam email versus how likely they are to be used in a ham email. The combination of these factors may in the end change your, or classifier’s, original belief about the most likely class that you had before seeing the content (evidence).

Now you can replace the conditional probability of `P(class | content)` with `P(content | class)`; for example, whereas before you had to calculate `P("spam" |` “Participate in our new lottery now!”) or equally `P("spam" |` [‘participate’: True, ‘in’: True, …, ‘now’: True, ‘!’: True]), which is hard to do because you will often end up with too few examples of exactly the same email content or exactly the same combination of features, now you can estimate `P(`[‘participate’: True, ‘in’: True, . . . , ‘now’: True, ‘!’: True] | `"spam")` instead. But how does this solve the problem? Aren’t you still dealing with a long sequence of features?

Here is where the “naïve” assumption in Naïve Bayes helps: it assumes that the features are *independent of each other*, or that your chances of seeing a word “lottery” in an email are independent of seeing a word “new” or any other word in this email before.

So you can estimate the probability of the whole sequence of features given a class as a product of probabilities of *each* feature given this class. That is:

P(['participate': True, 'in': True, . . . , 'now': True, '!': True] | "spam") = P('participate': True | "spam") * P('in': True | "spam") … * P('!': True | "spam")

If you express ['participate': True] as the first feature in the feature list, or f_1, ['in': True] as f_2, and so on, until f_n = ['!': True], you can use the general formula:

$$P([f_1, f_2, \ldots, f_n] \mid class) = P(f_1 \mid class) * P(f_2 \mid class) \ldots * P(f_n \mid class)$$

Figure 2.14 illustrates the classification process.

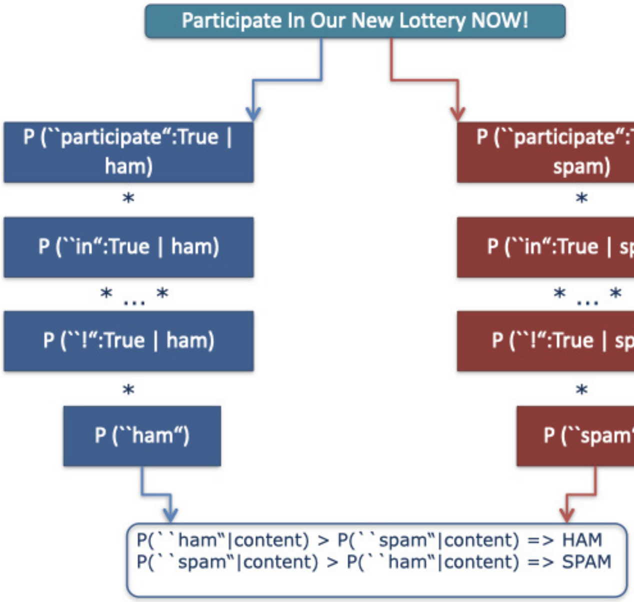

Participate In Our New Lottery NOW!
P (``participate":True | ham)
*
P (``in":True | ham)
* ... *
P (``!":True | ham)
*
P (``ham")
P (``participate":True | spam)
*
P (``in":True | spam)
* ... *
P (``!":True | spam)
*
P (``spam")
P(` `ham"|content) > P(` `spam"|content) => HAM
P(` `spam"|content) > P(` `ham"|content) => SPAM

Figure 2.14 Classification process: the conditional probabilities are multiplied with the class probabilities.

Now that you have broken down the probability of the whole feature list given class into the probabilities for each word given that class, how do you actually estimate them? Since for each email you note which words occur in it, the *total number* of times you can switch on the flag ['feature': True] equals the total number of emails in that class, while the *actual number* of times you switch on this flag is the number of emails where this feature is actually present. The conditional probability P(feature | class) is simply the proportion of the two:

```
P(feature | class) = number(emails in class with feature present) /
                     total_number(emails in class)
```

These numbers are easy to estimate from the training data—let's try to do that with an example.

Exercise 6

Suppose you have 5 spam emails and 10 ham emails. What are the conditional probabilities for P('prescription':True | spam), P('meeting':True | ham), P('stock':True | spam) and P('stock':True | ham), if:

- 2 spam emails contain word *prescription*
- 1 spam email contains word *stock*
- 3 ham emails contain word *stock*
- 5 ham emails contain word *meeting*

Solution:

The probabilities are simply:

- P('prescription': True | spam) = number(spam emails with 'prescription')/number(spam emails) = 2/5 = 0.40
- P('meeting': True | ham) = 5/10 = 0.50
- P('stock': True | spam) = 1/5 = 0.20
- P('stock':True | ham) = 3/10 = 0.30

Now you have all the components in place. Let's iterate through the classification steps again: during the training phase, the algorithm learns prior class probabilities (this is simply class distribution. For example, P(ham)=0.71 and P(spam)=0.29) and probabilities for each feature given each of the classes (this is simply the proportion of emails with each feature in each class, e.g. P('meeting':True | ham) = 0.50). During the test phase, or when the algorithm is applied to a new email and is asked to predict its class, the following comparison from the beginning of this section is applied:

```
Predict "spam" if P(spam | content) > P(ham | content)
Predict "ham" otherwise
```

This is what we started with originally, but we said that the conditions are flipped, so it becomes:

```
Predict "spam" if P(content | spam) * P(spam) / P(content) > P(content | ham)
     * P(ham) / P(content)
Predict "ham" otherwise
```

Note that we end up with P(content) in denominator on both sides of the expression, so the absolute value of this probability doesn't matter and it can be removed from the expression altogether.[13] So we can simplify the expression as:

```
Predict "spam" if P(content | spam) * P(spam) > P(content | ham) * P(ham)
Predict "ham" otherwise
```

[13] Since the probability always has a positive value, it won't change the comparative values on the two sides. For example, if you compare 10 to 4, you would get 10>4 whether you divide the two sides by the same positive number like (10/2)>(4/2) or not.

P(spam) and P(ham) are class probabilities estimated during training, and P(content | class), using naïve independence assumption, are products of probabilities, so:

Predict "spam" if P([f_1, f_2, …, f_n]| spam) * P(spam) > P([f_1, f_2, …, f_n]| ham) * P(ham)
Predict "ham" otherwise

is split into the individual feature probabilities as:

Predict "spam" if P(f_1 | spam) * P(f_2| spam) … * P(f_n| spam) * P(spam) >
P(f_1 | ham) * P(f_2| ham) … * P(f_n| ham) * P(ham)
Predict "ham" otherwise

This is the final expression the classifier relies on. The following listing implements this idea. Many toolkits come with an implementation of a range of widely used machine-learning algorithms. Since Naive Bayes is frequently used for NLP tasks, NLTK comes with its own implementation, too, and here you are going to use it.

Listing 2.9 Code to train a Naïve Bayes classifier

```
from nltk import NaiveBayesClassifier, classify

def train(features, proportion):
    train_size = int(len(features) * proportion)
    train_set, test_set = features[:train_size], features[train_size:]
    print (f"Training set size = {str(len(train_set))} emails")
    print (f"Test set size = {str(len(test_set))} emails")
    classifier = NaiveBayesClassifier.train(train_set)
    return train_set, test_set, classifier

train_set, test_set, classifier = train(all_features, 0.8)
```

Import the classifier implementation from NLTK.

Remember that you need to set aside part of the data for testing.

Use the first n% (according to the specified proportion) of emails with their features for training, and the rest for testing.

Print out simple statistics to make sure the data is split correctly.

Initialize the classifier.

Apply the train function using 80% (or a similar proportion) of emails for training. Note the use of the all_features structure created using code from listing 2.8.

2.3.5 Step 5: Evaluate your classifier

Finally, let's evaluate how well the classifier performs in detecting whether an email is spam or ham. For that, let's use the accuracy score returned by the NLTK's classifier in the following listing.

Listing 2.10 Code to evaluate classifier's performance

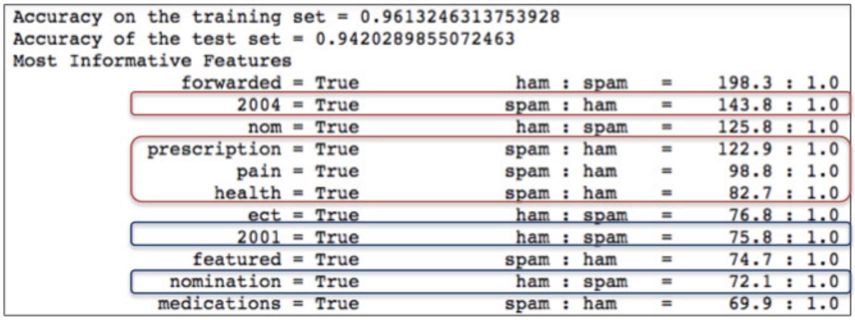
```
def evaluate(train_set, test_set, classifier):
    print (f"Accuracy on the training set =
     {str(classify.accuracy(classifier, train_set))}")
    print (f"Accuracy of the test set = {str(classify.accuracy(classifier,
     test_set))}")
    classifier.show_most_informative_features(50)

evaluate(train_set, test_set, classifier)
```

Let's define a function that will estimate the accuracy of the classifier on each set.

In addition, the NLTK's classifier allows you to inspect the most informative features (words). You need to specify the number of the top most informative features to look into. For example, 50 here.

Figure 2.15 presents an example of an output returned by the previous code.



```
Accuracy on the training set = 0.9613246313753928
Accuracy of the test set = 0.9420289855072463
Most Informative Features
               forwarded = True           ham : spam   =    198.3 : 1.0
                    2004 = True          spam : ham    =    143.8 : 1.0
                     nom = True           ham : spam   =    125.8 : 1.0
            prescription = True          spam : ham    =    122.9 : 1.0
                    pain = True          spam : ham    =     98.8 : 1.0
                  health = True          spam : ham    =     82.7 : 1.0
                     ect = True           ham : spam   =     76.8 : 1.0
                    2001 = True           ham : spam   =     75.8 : 1.0
                featured = True          spam : ham    =     74.7 : 1.0
              nomination = True           ham : spam   =     72.1 : 1.0
             medications = True          spam : ham    =     69.9 : 1.0
```

Figure 2.15 Output of the code in Listing 2.10. Features indicative of spam are highlighted in red, and features indicative of ham are highlighted in blue.

One piece of information that this code provides you with is the *most informative features*, that is, the list of words that are most strongly connected to a particular class. This is functionality of the classifier that is implemented in NLTK, so all you need to do is call on this function as `classifier.show_most_informative_features` and specify the number of words n that you want to see as an argument. This function then returns the top `n` words ordered by their "informativeness" or predictive power. Behind the scenes, the function measures "informativeness" as the highest value of the difference in probabilities between `P(feature | spam)` and `P(feature | ham)`. That is, `max[P(word: True | ham) / P(word: True | spam)]` for most predictive ham features, and `max[P(word: True | spam) / P(word: True | ham)]` for most predictive spam features.[14] The output shows that such words (features) as "prescription", "pain", "health" and so on are much more strongly associated with spam emails—the ratios on the right show the comparative probabilities for the two classes. For instance, P("prescription" | spam) is 122.9 times higher than `P("prescription" | ham)`. On the other hand, "nomination" is more strongly associated with ham emails. As you can see, many spam emails in this dataset are related to medications, which shows a particular bias—the most typical spam that you personally get might be on a different topic altogether! What effect might this mismatch between the training data from the publicly available dataset like Enron and your personal data have? You will see another example of this issue in Exercise 7.

One other piece of information presented in this output is accuracy. Test accuracy shows the proportion of test emails that are correctly classified by Naive Bayes among all test emails. The code above measures the accuracy on both the training data and test data. Note that since the classifier is trained on the training data, it actually gets to "see" all the correct labels for the training examples. Shouldn't it then know the correct

[14] Check out NLTK's documentation for more information at https://www.nltk.org/api/nltk.classify.html#nltk.classify.naivebayes.NaiveBayesClassifier.most_informative_features.

answers and perform at 100% accuracy on the training data? Well, the point here is that the classifier doesn't just retrieve the correct answers: during training it has built some probabilistic model (that is, learned about the distribution of classes and the probability of different features), and then it applies this model to the data. So, it is actually very likely that the probabilistic model doesn't capture all the things in the data 100% correctly. For example, there might be noise and inconsistencies in the real emails: note that "2004" gets strongly associated with the spam emails and "2001" with the ham emails, although it does not mean that there is anything peculiar about the spam originating from 2004. This might simply show a bias in the particular dataset, and such phenomena are hard to filter out in any real data. That means, however, that if some ham email in training data contains a word "2004" as well as some other words that are otherwise related to spam, this email will get classified as spam by the algorithm. Similarly, as many medication-related words are strongly associated with spam, a rare ham email that is actually talking about some medication the user ordered might get misclassified as spam.

Therefore, when you run the previous code, you will get accuracy on the training data of 96.13%. This is not perfect (that is, not 100%) but very close to it! When you apply the same classifier to new data—the test set that the classifier hasn't seen during training—the accuracy reflects its generalizing ability. That is, it shows whether the probabilistic assumptions it made based on the training data can be successfully applied to any other data. The accuracy on the test set is 94.20%, which is slightly lower than that on the training set, but it is also very high.

Finally, if you'd like to gain any further insight into how the words are used in the emails from different classes, you can also check the occurrences of any particular word in all available contexts. For example, word "stocks" features as a very strong predictor of spam messages. Why is that? You might be thinking, "OK, some emails containing "stocks" will be spam, but surely there must be contexts where "stocks" is used in a completely harmless way?" Let's check this using the following code.

Listing 2.11 Code to check the contexts of specific words

```
from nltk.text import Text

def concordance(data_list, search_word):
    for email in data_list:
        word_list = [word for word in word_tokenize(email.lower())]
        text_list = Text(word_list)
        if search_word in word_list:
            text_list.concordance(search_word)


print ("STOCKS in HAM:")
concordance(ham_list, "stocks")
print ("\n\nSTOCKS in SPAM:")
concordance(spam_list, "stocks")
```

Import NLTK's Text data structure.

"Concordancer" is a tool that checks for the occurrences of the specified word and prints out the word in its context. By default, NLTK's concordancer prints out the search_word surrounded by the previous 36 and the following 36 characters—so note, that it doesn't always result in full words.

Apply this function to two lists—ham_list and spam_list—to find out about the different contexts of use for the word "stocks".

If you run this code and print out the contexts for “stocks”, you will find out that “stocks” feature in only four ham contexts (for example, an email reminder “Follow your *stocks* and news headlines”) as compared to hundreds of spam contexts including “*Stocks* to play”, “Big money was made in these *stocks*”, “Select gold mining *stocks*”, “Little *stocks* can mean big gains for you”, and so on.

Congratulations—you have built your own spam-filtering algorithm and learned how to evaluate it and explore the results!

2.4 Deploying your spam filter in practice

Why are the evaluation steps important? We’ve said before that the machine learns from experience—data that it is provided with—so obviously, the more data the better. You started with about five thousand emails, but you had to set 20% aside for testing, and you were not allowed to use them while training. Doesn’t it mean practically “losing” valuable data that the classifier could have used more effectively?

Well, if you build an application that you plan to use in real life you want it to perform its task well. However, you cannot predict in advance what data the classifier will be exposed to in the future, so the best way to predict how well it will perform is to test it on the available labeled data. This is the main purpose of setting aside 20% or so of the original labeled data and of running evaluation on this test set. Once you are happy with the results of your evaluation, you can deploy your classifier in practice!

For instance, the classifier that you’ve built in this chapter performs at 94% accuracy, so you can expect it to classify real emails into spam and ham quite accurately. It’s time to deploy it in practice then. When you run it on some new emails (perhaps, some from your own inbox) you need to perform the same steps on these emails as before, that is:

- You need to read them in, then
- You need to extract the features from these emails, and finally
- You need to apply the classifier that you trained before on these emails.

The following code shows how you can do that. Feel free to type in your own emails as input.

Listing 2.12 Code to apply spam filtering to new emails

Read the emails extracting their textual content and keeping the labels for further evaluation.

```
test_spam_list = ["Participate in our new lottery!", "Try out this new
     medicine"]
test_ham_list = ["See the minutes from the last meeting attached",
                 "Investors are coming to our office on Monday"]

test_emails = [(email_content, "spam") for email_content in test_spam_list]
test_emails += [(email_content, "ham") for email_content in test_ham_list]
```

Feel free to provide your own examples.

```
new_test_set = [(get_features(email), label) for (email, label) in
      test_emails]

evaluate(train_set, new_test_set, classifier)
```

Extract the features.

Apply the trained classifier and evaluate its performance.

The classifier that you've trained in this chapter performs with 100% accuracy on these examples. Good! How can you print out the predicted label for each particular email though? For that, you simply extract the features from the email content and print out the label. That is, you don't need to run the full evaluation with the accuracy calculation. The following code suggests how you can do that.

Listing 2.13 Code to print out the predicted label

```
for email in test_spam_list:
    print (email)
    print (classifier.classify(get_features(email)))
for email in test_ham_list:
    print (email)
    print (classifier.classify(get_features(email)))
```

For each email in each list, this code prints out the content of the email and the predicted label.

Finally, let's make the code more interactive and see if the classifier can predict the class label on any input text in real time. For example, how about reading the emails of your choice straight from the keyboard and predicting their label on the spot? This is not very different from what you've just done; the only difference is that instead of reading the emails from the predefined list, you should allow your code to read them from the keyboard input. Python's `input` functionality allows you to do that. Let's read the emails typed in from the keyboard and stop when no email is typed in. For that, use the `while-break loop` as the code below shows. The code will keep asking for the next email until the user presses Enter.

Listing 2.14 Code to classify the emails read in from the keyboard

```
while True:
    email = input("Type in your email here (or press ‘Enter’): ")
    if len(email)==0:
        break
    else:
        prediction = classifier.classify(get_features(email))
        print (f"This email is likely {prediction}\n")
```

Ask the user to type in the text of the email.

Stop when the user provides no text and presses Enter instead.

Print out the predicted label for the email.

Summary

Let's summarize what you have covered in this chapter:

- You have learned about a powerful family of NLP and machine learning tasks that deal with classification. *Classification* is concerned with assigning objects to a predefined set of categories, groups, or classes based on their characteristic properties.
- Humans perform classification on a regular basis, and machine-learning algorithms can be taught to do that provided with a sufficient number of examples

and some guidance from humans. When the labeled examples and the general outline of the task are provided for the machine, this is called *supervised learning*.

- Spam filtering is an example of a *binary classification task*: the machine has to learn to distinguish between exactly two classes—spam and normal email (often called ham).
- Classification relies on specific properties of the classified objects. In machine learning terms, such properties are called *features*. For spam filtering, some of the most informative features are words used in the emails.
- To build a spam-filtering algorithm, you can use one of the publicly available spam datasets. In this chapter, you've used *Enron* dataset.
- You have learned how build a classifier in five steps:
  - First, the emails should be read, and the two classes should be clearly defined for the machine to learn from.
  - Next, the text content should be extracted.
  - Then the content should be converted into features.
  - The classifier should be trained on the training set of the data.
  - Finally, the classifier should be evaluated on the test set.
- The data comes in as a single string of symbols. To extract the words from it, you may rely on the NLP tools called *tokenizers*.
- NLP libraries come with such tools, as well as implementations of a range of frequently used classifiers. In this chapter, you used Natural Language Processing Toolkit (NLTK).
- There are a number of machine learning classifiers, and in this chapter, you've applied one of the most interpretable of them—*Naive Bayes*. Naive Bayes is a probabilistic classifier: it assumes that the data in two classes is generated by different probability distributions, which are learned from the training data. Despite its simplicity and "naive" feature independence assumption, Naive Bayes often performs well in practice, and sets competitive baseline for other more sophisticated algorithms.
- It is important that you split your data into training (for example, 80% of the original dataset) and test (the rest of the data) sets, and train the classifier on the training data *only*, so that you can fairly assess it on the test set. The test set serves as new unseen data for the algorithm, so you can come to a realistic conclusion about how your classifier may perform in practice. In this chapter, you learned how to evaluate the classifier and interpret the results.
- Once satisfied with the performance of your classifier on the test data, you can deploy it in practice.

Finally, apply what you have learned in this chapter and test your new skills by attempting the following practical exercise. You can check your solutions against the notebook, but first try to write the code yourself!

Exercise 7

Apply the trained classifier to a different dataset, for example to enron2/ spam and ham emails that originate with a different owner (check Summary.txt for more information). For that you need to:

- Read the data from the spam/ and ham/ subfolders in enron2/.
- Extract the textual content and convert it into features.
- Evaluate the classifier.

What do the results suggest? *Hint: one man's spam may be another man's ham*. If you are not satisfied with the results, try combining the data from the two owners in one dataset.

Chapter 3 from *Getting Started with Natural Language Processing*
by Ekaterina Kochmar

In this chapter, you'll explore information search, or information *retrieval*, a task widely used in many applications, from searching in an Internet browser to searching for the relevant files on your personal computer. You'll learn to evaluate the importance and relevance of different information as well as other useful NLP concepts as you implement your own information search algorithm, step by step.

3 Introduction to Information Search

This chapter covers

- Implementing your own information retrieval algorithm
- Understanding useful NLP concepts, including stemming and stopwords removal
- Assessing importance of different bits of information in search
- Evaluating the relevance of the documents to the information need

This chapter will focus on algorithms for information search, which also has a more technical name—*information retrieval*. It will explain the steps in the search algorithm from beginning to end, and by the end of this chapter you will be able to implement your own search algorithm.

You might have come across the term information retrieval in the context of search engines: for example, Google famously started its business by providing a powerful search algorithm that kept improving over time. The search for information, however, is a basic need that you may face not only in the context of searching online. For instance, every time you search for the files on your computer, you also perform a kind of information retrieval. In fact, the task predates digital era. Before computers and Internet became a commodity, one had to manually wade through paper copies of encyclopedias, books, documents, files, and so on. Thanks to the technology, the algorithms these days help you do many of these tasks automatically.

The field of information retrieval has a long history and has seen a lot of development over the past decades. As you can imagine, Google and other search engines are dealing with large amounts of data, which makes their task exceptionally challenging—they have to process billions of pages in a matter of seconds and be able to return most relevant of those to satisfy the information needs of the users. Truly amazing, if you think about the complexity of the task!

In this chapter, we will break this process into steps. We will look into how the information need is expressed as a query and processed for the computer to understand it, how the documents should be processed and matched to the queries, and how the relevance of the documents to the queries can be assessed. Search engines, fundamentally, go through all the same steps, albeit they do it on a much larger scale, and employ a number of additional techniques, for example, learning from user clicks, linking web content via hyperlinks, using optimization to speed the processing up, storing intermediate results, and so on. So this chapter should perhaps start with a disclaimer: we are not going to build a new Google competitor algorithm here (although you might consider building one in the future), but we will build a core information search application that you can use in your real life projects.

3.1 Understanding the task

Let's look into the scenario from chapter 1 again.

Scenario 1 (reminder from chapter 1)

Imagine that you have to perform the search in a collection of documents yourself, without the help of the machine. For example, you have a thousand printed out notes and minutes related to the meetings at work, and you only need those that discuss the management meetings. How will you find *all* such documents? How will you identify the *most relevant* of these?

We said that if you were tasked with this in actual life, you would go through the documents one by one, identifying those that contain the key words (like "management" and "meetings") and split all the documents into two piles: those documents that you should keep and look into further and those that you can discard because they do not answer your information need in learning more about the management meetings. This task is akin to filtering, and figure 3.1 should remind you how we set it up in chapter 1.

Now, there are a couple of points that we did not get to discuss before. Imagine there are a hundred of documents in total and you can quickly skim through them to filter out the most irrelevant ones—those that do not even mention either "meetings"

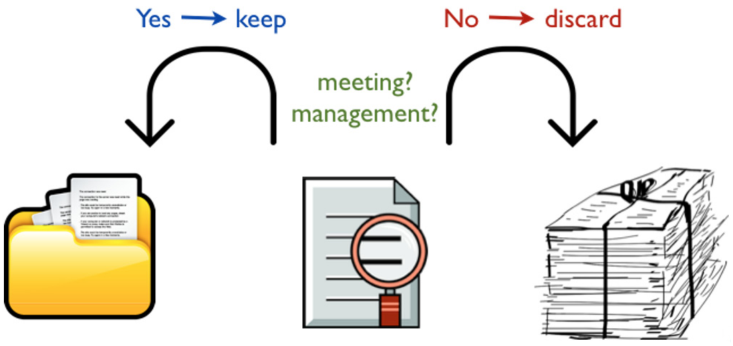


Figure 3.1 Simple filtering of documents into "keep" and "discard" piles based on occurrence of words.

or "management". But what if a high number of documents actually do contain one, another, or both words? Say, after this initial filtering, you end up with 70 such documents. This is not the original thousand, but still too much to read through carefully. At the very least, you'd like to be able to sort them in the order of relevance, so that you can start reading with the most relevant ones and then stop as soon as you found the information you were looking for. How can you judge whether one of the documents is more relevant than the others, and how can you sort all of them in the order of decreasing relevance?

Luckily, these days we have computers, and most documents are stored electronically. Computers can really help us speed the things up here. If we can formulate our information needs for them more or less precisely, they can be much quicker in spotting the key words, estimating the relevance and sorting the documents for us—in fact, in a matter of seconds (think Google). So let's formulate a new, more technical scenario for this chapter.

Scenario 2 (based on Scenario 1, but more technical!)

Imagine that you have to perform the search in a collection of documents, this time *with the help of the machine*. For example, you have a thousand notes and minutes related to the meetings at work stored in electronic format, and you only need those that discuss the management meetings.

First, how will you find *all* such documents? In other words, how can you code the *search* algorithm and what characteristics of the documents should the search be based on?

Second, how will you identify the *most relevant* of these documents? In other words, how can you implement a *sorting* algorithm to sort the documents in order of decreasing relevance?

This scenario is only different from the previous one in that it allows you to leverage the computational power of the machine, but the drill is the same as before: get the machine to identify the texts that have the keywords in them, and then sort the "keep" pile according to the relevance of the texts, starting with the most relevant for the user or yourself to look at.

Despite us saying just now that the procedure is similar to how the humans perform the task (as in Scenario 1), there are actually some steps involved in getting the machine to identify the documents with the keywords in them and sorting by relevance that we are not explicitly mentioning here. For instance, we humans have the following abilities that we naturally possess but machines naturally lack:

- We know what represents a word, while a machine gets in a sequence of symbols and does not, by itself, have a notion of what a "word" is.
- We know which words are keywords. For example, if we are interested in finding the documents on management meetings, we will consider those containing "meeting" and "management", but also those containing "meetings" and potentially even "manager" and "managerial". The machine, on the other hand, does not know that these words are related, similar, or basically different forms of the same word.
- We have an ability to focus on what matters. In fact, when reading texts we usually skim over certain words rather than pay equal attention to each word. For instance, when reading a sentence "Last Friday the management committee had a meeting", which words do you pay more attention to? Which ones express the key idea of this message? Think about it—and we will return to this question later. The machines, on the other hand, should be specifically "told" which words matter more.
- Finally, we also intuitively know how to judge what is more relevant. The machines can make relevance judgments, too, but unlike us humans they need to be "told" how to measure relevance in precise numbers.

That, in a nutshell, represents the basic steps in the search algorithm. Let's visualize these steps as in figure 3.2.

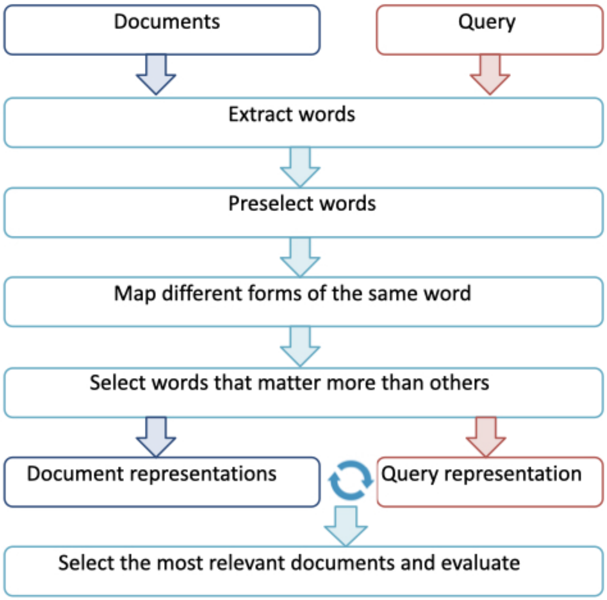

Documents
Query
Extract words
Preselect words
Map different forms of the same word
Select words that matter more than others
Document representations
Query representation
Select the most relevant documents and evaluate

Figure 3.2 Information search algorithm in a nutshell.

You are already familiar with the first step in this algorithm. When building a spam filtering application in chapter 2, you learned how to use a tokenizer to extract words from raw text. In this chapter, you will learn about other NLP techniques to preselect words, map the different forms of the same word to each other, and weigh the words according to how much information they contribute to the task. Then you will build an information search algorithm that for any query (for example, "management meetings") will find the most relevant documents in the collection of documents (for example, all minutes of the past managerial meetings sorted by their relevance).

Suppose you have built such an application following all the steps. You type in a query and the algorithm returns a document or several documents that are supposedly relevant to this query. How can you tell whether the algorithm has picked out the right documents? When you were building the spam filtering classifier, you faced the same problem, and we said that before you deploy your spam filter in practice it is a good idea to get an initial estimate of how well the classifier performs. You can do that if for some data you know the true labels—which emails are spam and which ones are ham. These true labels are commonly referred to as *ground truth* or *gold standard*, and to make sure your algorithm performs well you first evaluate it against gold standard labels. For that, you used a spam dataset where such gold standard labels were provided. You are going to do the same here. Let's use a dataset of documents and queries, where the documents are labeled with respect to their relevance to the queries. You will use this dataset as your gold standard, and before using the information search algorithm in practice, evaluate its performance against the ground truth labels in the labeled dataset.

3.1.1 Data and data structures

As in chapter 2, you are going to use a publicly available dataset labeled for the task. That means a dataset with a number of documents and various queries, and a labeled list specifying which queries correspond to which documents. Once you implement and evaluate a search algorithm on such data labeled with ground truth, you can apply it to your own documents in your own projects.

There are a number of datasets that can be used for this purpose.[15] In this chapter, you will use the dataset collected by the Centre for Inventions and Scientific Information (CISI)[16], which contains abstracts and some additional metadata from the journal articles on information systems and information retrieval. Despite the availability of other datasets, there are several reasons to choose the CISI dataset for this chapter, the primary of which are:

[15] See a list of publicly available datasets here: http://ir.dcs.gla.ac.uk/resources/test_collections/.

[16] You can download the dataset from http://ir.dcs.gla.ac.uk/resources/test_collections/cisi/.

- It is a relatively small dataset of 1460 documents and 112 queries, which is easy to process and work with. In addition, each document is relatively short, consisting of an article abstract and some additional information, which helps faster processing further.
- It contains gold standard annotations for the relevance of the documents to 76 queries.
- The results are easy to interpret, because the dataset does not include highly technical terms. In contrast, some other widely used benchmark datasets include medical articles or articles on technical subjects such as aerodynamics, which are harder to interpret for non-experts.

Let's first read in the data and initialize the data structures to keep the content. Note, that this dataset contains many documents and various queries. For instance, one query in this dataset asks what information science is, while another asks about the methods of information retrieval, and so on—there is a diverse set of 76 questions. Although you might just extract one particular query and focus on, say, searching for the documents in this dataset answering "What is information science?", since you have access to so many diverse queries, why not read all of them and store them in some data structure? This way, you will be able to search for the matching documents for *any* of the queries rather than for *only one specific query*, and check how your algorithm deals with a variety of information needs. For that, it would be good to keep track of different queries. For example, if you assign an unique identifier to each query (id1 = "What is information science?", id2 = "What methods do information retrieval systems use?", and so on), you can then easily select any of the queries by its id. You can apply the same approach to storing the documents, too. If each document is assigned a unique id, it will be easy to identify a particular document by its id. Finally, the matching between a particular query and documents answering the information need in this query can be encoded as a correspondence between query id and documents ids.

This suggests that you can use three data structures for this application:

1. A data structure for the documents that will keep document ids and contents.
2. A data structure for the queries that will keep query ids and contents.
3. A data structure matching the queries to the documents.

Exercise 1

What would be the best format(s) for representing these three data structures? What type of information do you need to keep in each case?

Solution:

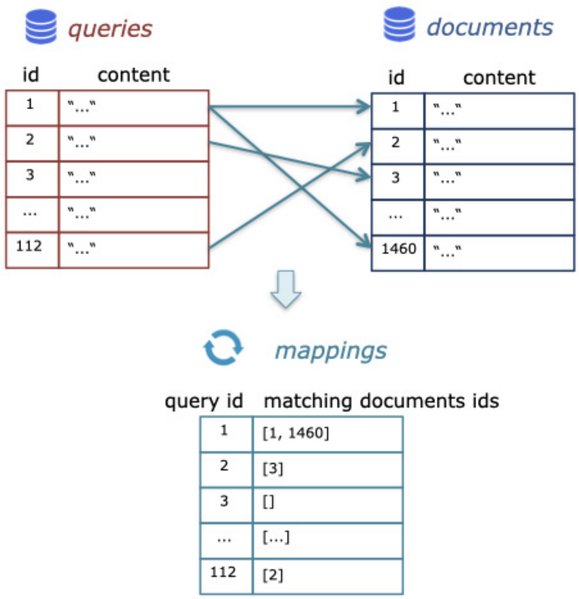
Information search is based on the idea that the content of a document or set of documents is relevant given the content of a particular query, so *documents* data structure should keep the contents of all available documents for the algorithm to select from.

If you have only one query to search for, you can store it as a string of text. However, if you want to use your algorithm to search for multiple queries, you may use a similar *queries* data structure to keep contents of all queries available in the dataset.

What would be the best way to keep track of which content represents which document? The most informative and useful way would be to assign a unique identifier—an index—to each document and each query. You can imagine, for example, storing content of the documents and queries in two separate tables, with each row representing a single document or query, and row numbers corresponding to the documents and queries ids. In Python, tables can be represented with dictionaries.[17]

Now, if you keep two Python dictionaries (tables) matching each unique document identifier (called *key*) to the document's content (called *value*) in *documents* dictionary and matching each unique query identifier to the query's content in *queries* dictionary, how should you represent the relevance mappings? You can use a dictionary structure again: this time, the keys will contain the queries ids, while the values should keep the matching documents ids. Since each query may correspond to multiple documents, it would be best to keep the ids of the matching documents as lists.

Figure 3.3 visualizes these data structures.



id	content
1	"..."
2	"..."
3	"..."
...	"..."
112	"..."

id	content
1	"..."
2	"..."
3	"..."
...	"..."
1460	"..."

query id	matching documents ids
1	[1, 1460]
2	[3]
3	[]
...	[...]
112	[2]

Figure 3.3 **Three data structures keeping the documents, queries, and their relevance mappings.**

[17] See https://docs.python.org/3/tutorial/datastructures.html.

As this figure shows, query with id 1 matches documents with ids 1 and 1460, therefore the *mappings* data structure keeps a list of [1, 1460] for query 1; similarly it keeps [3] for query 2, [2] for query 112, and an empty list for query 3, because in this example there are no documents relevant for this query.

Now let's look into the CISI dataset and code the data reading and initialization step. All documents are stored in a single text file CISI.ALL. It has a peculiar format: it keeps the abstract of each article[18] and some additional information, such as the index in the set, the title, authors' list and cross-references—a list of indexes for the articles that cite each other. Table 3.1 explains the notation.

Table 3.1 Notation Used in the CISI Dataset for the Articles

Notation	Meaning
.I	Document index in the set
.T	Article's title
.A	Authors' list
.W	Text of the abstract
.X	Cross-references list

For the information search application, arguably the most useful information is the content of the abstract: abstracts in the articles typically serve as a concise summary of what the article presents, something akin to a snippet. Are the other types of information included in the dataset useful? Well, you might be interested in the articles published by particular authors, so in some situations you might be interested in searching on the .A field specifically; similarly, if you are interested in articles with particular titles, you might benefit from using .T field only. For the sake of simplicity, in the application that you will develop in this chapter we won't distinguish between the .T, .A, and .W list and we'll merge them into one "content of the article" field, assuming that the information from each of them is equally valuable. .X field shows how many other articles refer to the particular article, so it may be used as a credibility rating of an article. This may be quite useful in practice, if you want to rate the articles by how reliable or respected they are (this is what the cross-references show), however in this application we won't focus on that and will remove the .X field.

Table 3.2 shows the format of information presentation in the CISI.ALL file using an example of the very first article in the set.

[18] Note that the full texts of the articles are not included in this dataset. However, this is not a problem for your search algorithm application. First of all, abstracts typically summarize the main content of the article in a concise manner, so the abstract content is a more condensed version of the article content. Second, the mappings are established between the queries and the documents containing the information summarized in table 3.1.

Table 3.2 Format Used for the Articles Representation in the CISI Dataset

```
.I 1
.T
18 Editions of the Dewey Decimal Classifications
.A
Comaromi, J.P.
.W
   The present study is a history of the DEWEY Decimal
Classification.  The first edition of the DDC was published
in 1876, the eighteenth edition in 1971, and future editions
will continue to appear as needed.  In spite of the DDC's
long and healthy life, however, its full story has never
been told.  There have been biographies of Dewey
that briefly describe his system, but this is the first
attempt to provide a detailed history of the work that
more than any other has spurred the growth of
librarianship in this country and abroad.
.X
...
```

As you can see, the field identifiers such as .A or .W are separated from the actual text by new line. In addition, the text within each field, for example, the abstract may be spread across multiple lines. Ideally, we would like to convert this format into something like text in Table 3.3. Note that for the text that falls within the same field. For example, with .W, the line breaks ("\n") are replaced with whitespaces, so each line now starts with a field identifier followed by the field content.

Table 3.3 Modified Format for the Articles Representation in the Dataset

.I *1*

.T *18 Editions of the Dewey Decimal Classifications*

.A *Comaromi, J.P.*

.W *The present study is a history of the DEWEY Decimal Classification. The first edition of the DDC was published in 1876, the eighteenth edition in 1971, and future editions will continue to appear as needed. In spite of the DDC's long and healthy life, however, its full story has never been told. There have been biographies of Dewey that briefly describe his system, but this is the first attempt to provide a detailed history of the work that more than any other has spurred the growth of librarianship in this country and abroad.*

The format in table 3.3 is much easier to work with: you can now read the text line by line, extract the unique identifier for the article from the field .I, merge the content of the fields .T, .A, and .W, and store the result in the *documents* dictionary as {1: "18

Editions of the … this country and abroad."}. Code in the following listing implements all these steps.

Listing 3.1 Code to populate the *documents* dictionary

```
def read_documents():
    f = open("cisi/CISI.ALL")
    merged = ""

    for a_line in f.readlines():
        if a_line.startswith("."):
            merged += "\n" + a_line.strip()
        else:
            merged += " " + a_line.strip()

    documents = {}

    content = ""
    doc_id = ""

    for a_line in merged.split("\n"):
        if a_line.startswith(".I"):
            doc_id = a_line.split(" ")[1].strip()
        elif a_line.startswith(".X"):
            documents[doc_id] = content
            content = ""
            doc_id = ""
        else:
            content += a_line.strip()[3:] + " "
    f.close()
    return documents

documents = read_documents()
print(len(documents))
print(documents.get("1"))
```

String variable merged keeps the result of merging the field identifier (e.g., .W) with its content.

Unless a string starts with a new field identifier, add the content to the current field separating the content from the previous line with a whitespace; otherwise, start a new line with the next identifier and field.

Initialize documents dictionary.

Each entry in the dictionary contains key=doc_id, which specifies the document's unique identifier, and value=content, which specifies the content of the article.

doc_id can be extracted from the line with the .I field identifier.

As .X field is always the last in each article, the start of the .X field signifies that you are done reading in the content of the article and can put the entry doc_id:content into the documents dictionary.

Otherwise, keep extracting the content from other fields (.T, .A and .W) removing the field identifiers themselves.

As a sanity check, print out the size of the dictionary (make sure it contains all 1460 articles) and print out the content of the very first article—it should correspond to the text in table 3.3.

The queries are stored in CISI.QRY file and follow a very similar format: half the time, you see only two fields, .I for the unique identifier and .W for the content of the query. Other queries, though, are formulated not as questions but rather as abstracts from other articles. In such cases, the query also has an .A field for the authors' list, .T for the title and .B field, which keeps the reference to the original journal in which the abstract was published. Table 3.4 presents an example of one of such original queries:

Table 3.4 Format Used for the Queries Representation in the CISI Dataset

```
.I 88
.T
Natural Language Access to Information Systems. An Evaluation Study
of Its Acceptance by End Users
.A
Krause, J.
.W
   The question is asked whether it is feasible to use subsets of
natural languages as query languages for data bases in actual applications
using the question answering system "USER SPECIALTY LANGUAGES" (USL).
Methods of evaluating a natural language based information system will
be discussed.  The results (error and language structure evaluation)
suggest how to form the general architecture of application systems which
use a subset of German as query language.
.B
(Inform. Systems, Vol. 5, No. 4, May 1980, pp. 297-318)
```

We are going to only focus on the unique identifiers and the content of the query itself (fields .W and .T, where available), so the code in listing 3.2 is quite similar to Listing 1 as it allows you to populate the *queries* dictionary with data.

Listing 3.2 Code to populate the *queries* dictionary

```
def read_queries():
    f = open("cisi/CISI.QRY")
    merged = ""

    for a_line in f.readlines():
        if a_line.startswith("."):
            merged += "\n" + a_line.strip()
        else:
            merged += " " + a_line.strip()

    queries = {}

    content = ""
    qry_id = ""

    for a_line in merged.split("\n"):
        if a_line.startswith(".I"):
            if not content=="":
                queries[qry_id] = content
                content = ""
                qry_id = ""
            qry_id = a_line.split(" ")[1].strip()
        elif a_line.startswith(".W") or a_line.startswith(".T"):
```

As before, merge the content of each field with its identifier, and separate different fields with line breaks \n.

Initialize queries dictionary and store key=qry_id and value=content for each query in the dataset as a separate entry in the dictionary

The start of a new entry is signified with the next .I field. At this point, add an entry to the dictionary.

```
            content += a_line.strip()[3:] + " "
    queries[qry_id] = content
    f.close()
    return queries

queries = read_queries()
print(len(queries))
print(queries.get("1"))
```

Otherwise, keep adding content to the content variable.

The very last query is not followed by any next .I field, so the strategy from above won't work—you need to add the entry for the last query to the dictionary using this extra step.

Print out the length of the dictionary (it should contain 112 entries), and the content of the very first query (this you can check against the text of the first query in CISI.QRY).

For the query example in table 3.4, this code will put the unique identifier linked to the query content from the fields .T and .W into the data structure. The particular entry will be represented as {88: "Natural Language Access to Information Systems . . . use a subset of German as query language."}.

Finally, you need to know which queries correspond to which documents. This information is contained in the CISI.REL file. This file uses a simple column-based format, where the first column keeps the reference to the query id, and the second column contains an id of one of the articles (documents) that matches this query. So all you need to do is read this file, split it into columns, and associate to the query id the list of ids for the documents that match the query. Listing 3.3 shows how to do this in Python.

Listing 3.3 Code to populate the *mappings* dictionary

```
def read_mappings():
    f = open("cisi/CISI.REL")

    mappings = {}

    for a_line in f.readlines():
        voc = a_line.strip().split()
        key = voc[0].strip()
        current_value = voc[1].strip()
        value = []
        if key in mappings.keys():
            value = mappings.get(key)
        value.append(current_value)
        mappings[key] = value

    f.close()
    return mappings

mappings = read_mappings()
print(len(mappings))
print(mappings.keys())
print(mappings.get("1"))
```

Split each line into columns. Python's split() performs splitting by whitespaces, while strip() helps with removing any trailing whitespaces.

If the mappings dictionary already contains some document ids for the documents matching the given query, you need to update the existing list with the current value; otherwise just add current value to the new list.

The key (query id) is stored in the first column, while the document id is stored in the second column.

Print out some information about the mappings data structure. For example, its length (it should tell you that 76 queries have documents associated with them), list of keys (so you can check which queries don't have any matching documents), and the list of ids for the documents matching the very first query (this should print out a list of 46 document ids, which you can check against CISI.REL).

For example, for the very first query, the mappings data structure should keep the following list: {1: [28, 35, 38, 1196, 1281]}.

That's it—you have successfully initialized one dictionary for *documents* with the ids linked to the articles content, another dictionary for *queries* linking queries ids to their correspondent texts, and the *mappings* dictionary, which matches the queries ids to the lists of relevant document ids.

Now, you are all set to start implementing the search algorithm for this data.

3.1.2 Boolean search algorithm

Let's start with the simplest approach: the information need is formulated as a query. If you extract the words from the query, you can then search for the documents that contain these words and return these documents, as they should be relevant to the query.

Here is the algorithm in a nutshell:

- Extract the words from the query.
- For each document, compare the words in the document to the words in the query.
- Return the document as relevant if any of the query words occurs in the document.

Figure 3.4 visualizes this algorithm.

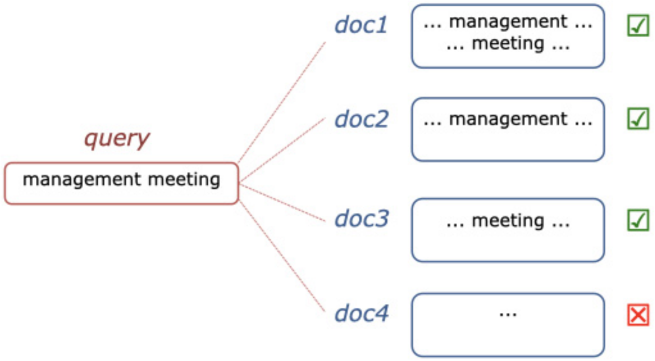


Figure 3.4 Simple search algorithm selects all documents that contain any of the words from the query.

The very first step in this algorithm is extraction of the words from both queries and documents. You may recall from chapter 2 that text comes in as a sequence of symbols or characters, and the machine needs to be told what a word is—you used a special NLP tool called tokenizer to extract words. Let's apply this text-preprocessing step here, too.

Listing 3.4 Preprocess the data in documents and queries

```
import nltk
from nltk import word_tokenize

def get_words(text):
    word_list = [word for word in word_tokenize(text.lower())]
    return word_list

doc_words = {}
qry_words = {}
for doc_id in documents.keys():
```

Use NLTK's word_tokenize as in chapter 2.

Text is converted to lower case and split into words.

```
    doc_words[doc_id] = get_words(documents.get(doc_id))
for qry_id in queries.keys():
    qry_words[qry_id] = get_words(queries.get(qry_id))

print(len(doc_words))
print(doc_words.get("1"))
print(len(doc_words.get("1")))
print(len(qry_words))
print(qry_words.get("1"))
print(len(qry_words.get("1")))
```

Entries in both documents and queries are represented as word lists.

Print out the length of the dictionaries (these should be the same as before—1460 and 112), and check what words are extracted from the first document and the first query.

Now let's code the simple search algorithm described previously. We will refer to it as the Boolean search algorithm since it relies on presence (1) or absence (0) of the query words in the documents.

Listing 3.5 Simple Boolean search algorithm

```
def retrieve_documents(doc_words, query):
    docs = []
    for doc_id in doc_words.keys():
        found = False
        i = 0
        while i<len(query) and not found:
            word = query[i]
            if word in doc_words.get(doc_id):
                docs.append(doc_id)
                found=True
            else:
                i+=1
    return docs

docs = retrieve_documents(doc_words, qry_words.get("3"))
print(docs[:100])
print(len(docs))
```

Iterate through the documents.

found flag will be turned on as soon as you find any of the query words in the document.

i is the index of the query word in the query word list.

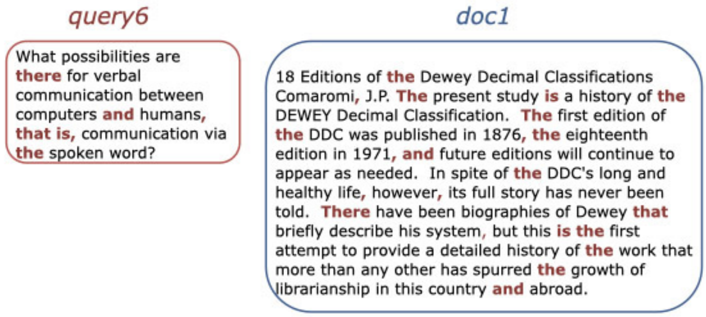
Keep iterating through the words in the query word list until either of the two conditions is satisfied. You have reached the end of the word list, or one of the words from the query word list is found in the document (found flag is on).

As soon as you find a query word in the document, turn the found flag on. This will help you optimize the search, since as soon as you find one word, you don't need to look any further in this document.

Check the results: select a query by its id (for example, query with id 3 here), print out the ids of the documents that the algorithm found (for example, the first 100, as there may be many), check how many there are in total.

If you run this code with a query with id 3 from the `queries` data structure (the text of this query is "What is information science? Give definitions where possible."), you will get 1410 documents returned as relevant—this means that almost each document in the collection of 1460 documents is considered "relevant" by this algorithm! There is nothing special about query with id 3; in fact, almost any query will return comparably huge number of "relevant" documents with this approach. That probably means that no truly relevant document escapes such thorough search, but in practice it is not helpful. In addition to returning a huge number of documents, the algorithm does not provide any relevance sorting for them, and without such sorting looking through 1410 is not significantly better than looking through 1460. So what exactly went wrong here?

Let's look into how the algorithm decided on the documents relevant for query with id 6 ("What possibilities are there for verbal communication between computers and humans, that is, communication via the spoken word?"). According to the gold standard in mappings data structure, only one document matches this query, but the simple algorithm you applied above returns all 1460 documents as relevant. Figure 3.5 highlights the words by which the match was identified between query 6 and document 1. As it shows, the query is matched to the document based on occurrence of such words as "there", "this", "the", "and", "is", and even a comma since punctuation marks are part of the word list returned by the tokenizer.


query6
doc1

What possibilities are **there** for verbal communication between computers **and** humans**,** **that is,** communication via **the** spoken word?

18 Editions of **the** Dewey Decimal Classifications Comaromi**,** J.P. **The** present study **is** a history of **the** DEWEY Decimal Classification. **The** first edition of **the** DDC was published in 1876**,** **the** eighteenth edition in 1971**,** **and** future editions will continue to appear as needed. In spite of **the** DDC's long and healthy life**,** however**,** its full story has never been told. **There** have been biographies of Dewey **that** briefly describe his system, but this **is** **the** first attempt to provide a detailed history of **the** work that more than any other has spurred **the** growth of librarianship in this country **and** abroad.

Figure 3.5 The match between the query and the documents is established based on highlighted words.

On the face of it, there is a considerable word overlap between the query and the document, yet if you read the text of the query and the text of the document, they don't seem to have any ideas in common, so in fact this document is not relevant for the given query at all! It seems like the words on the basis of which the query and the document are matched here are simply the wrong ones. They are somewhat irrelevant to the actual information need expressed in the query. How can you make sure that the query and the documents are matched on the basis of more meaningful words?

Exercise 2

Another way to match the documents to the queries would be to make it a requirement that the document should contain *all* the words from the query rather than *any*.

Is this a better approach? Modify the code of the simple Boolean search algorithm to match documents to the queries on the basis of all words, and compare the results.

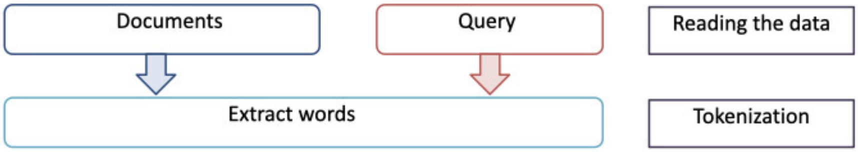
Solution:
First try to solve this task yourself, and then check your solution against the sample solution in the Jupyter notebook.[19]

If you consider an example of any of the queries, you may notice that it is rarely the case that a document, even if it is generally relevant, contains *all* words from the query. (At the very least, it does not have to contain question words like "what" and "which" from the query to be relevant). Therefore, if you run this code, which applies the more conservative approach of returning *only* the documents with *all* query words

[19] All the code for this book is currently available at https://app.box.com/folder/70006068203.

in them, it will work even worse at this stage—it simply will not find any relevant documents for any of the queries.

Before we move on, let's summarize which steps of the algorithm you have implemented so far: you have read the data, initialized the data structures, and tokenized the texts.



3.2 Processing the data further

In the previous section, we have identified several weaknesses of the current algorithm. Let's look into further preprocessing steps that will help you represent the content of both the documents and the queries in a more informative way.

3.2.1 Preselecting the words that matter: Stopwords removal

The main problem with the search algorithm identified so far is that it considers all words in the queries and documents as equally important. This leads to poor search results, but on top of that it is also intuitively incorrect. Let's consider an example of query 6, "What possibilities are there for verbal communication between computers and humans, that is, communication via the spoken word?", and identify the words that matter.

Exercise 3

Look at the following three queries. Which of the words express the information need most precisely?

(1) What possibilities are there for verbal communication between computers and humans?

(2) How much do information retrieval and dissemination systems cost?

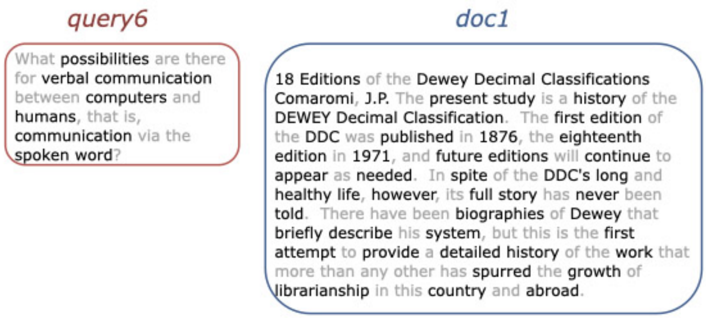
(3) Testing automated information systems.

Solution:

You may notice that not all words are equally meaningful in the sentences above. A good test for that would be to ask yourself whether you can define in one phrase what a particular word means: for example, what does "the" mean? You can say that "the" does not have a precise meaning of its own, rather it serves a particular function—it signifies that the word following it is defined in the specific context. For example, when you see "the" in "Look at the following queries", you know *precisely which queries* I am talking about.

You may find out that many of the frequent and short words like “for”, “at”, “a”, “the”, and a number of others are less charged with meaning than rarer and longer words like “communication” or “retrieval”. Such short words are very frequent in language—almost any text you look at would contain multiple “the”s, “a”s, and so on. You have seen an example of that when you ran the simple search algorithm and it was misled by the presence of such words in all texts. Most of such words don’t have a particular meaning of their own, rather they express a function: similarly to “the” denoting that the next word or phrase is identifiable in the context, “at” and “in” help specifying location or time, and “which” or “what” in the beginning of a sentence suggest that the sentence may be a question. In linguistic terms, such words are called *function words*. You might even notice that when you read a text, for example an article or an email, you tend to skim over such words without paying much attention to them.

What happens to the search algorithm when these words are present? You have seen in the example before that they don’t help identify the relevant texts, so in fact the algorithm’s effort is wasted on them. What would happen if the less meaningful words were not taken into consideration? Figure 3.6 shows an example with the more meaningful words highlighted and the less meaningful ones grayed out.



query6

What **possibilities** are there for **verbal communication** between **computers** and **humans**, that is, **communication** via the **spoken word**?

doc1

18 Editions of the **Dewey Decimal Classifications Comaromi**, **J.P.** The **present study** is a **history** of the **DEWEY Decimal Classification**. The **first edition** of the **DDC** was **published** in **1876**, the **eighteenth edition** in **1971**, and **future editions** will **continue** to **appear** as **needed**. In **spite** of the **DDC's long** and **healthy life**, **however**, its **full story** has **never** been **told**. There have been **biographies** of **Dewey** that **briefly describe** his **system**, but this is the **first attempt** to **provide** a **detailed history** of the **work** that more than any other has **spurred** the **growth** of **librarianship** in this **country** and **abroad**.

Figure 3.6 The more meaningful words in the query and document are highlighted.

You can see that were the less meaningful words not removed before matching documents to queries, document 1 would not stand a chance—there is simply not a single word overlapping between the query and this document. You can also see that the words that are not grayed out concisely summarize the main idea of the text.

This suggests the first improvement to the developed algorithm. Let’s remove the less meaningful words. In NLP applications, the less meaningful words are called *stopwords*, and luckily you don’t have to bother with enumerating them, Since stopwords are highly repetitive in English, most NLP toolkits have a specially defined stopwords list, so you can rely on this list when processing the data, unless you want to customize it. For example if you believe that it should be extended with more words or that some words that are included in the standard stopwords list should not be there, you could use your own list of stopwords.

In addition to removing stopwords note that Figure 3.6 doesn’t have punctuation marks, that is, full stops, commas, and question marks, highlighted. Punctuation marks may prove useful in some applications, but will unlikely help here: many queries will contain question marks and documents won't necessarily have any, while all documents will have commas and full stops, so punctuation marks are not going to be informative in the matching process. Let's filter them out, too. Code in listing 3.6 shows how to do that.

Listing 3.6 Preprocessing: Stopwords and punctuation marks removal

NLTK includes stopwords for multiple languages, so you need to specify that you want to use the one for English. You can check which words are included by print(stoplist).

Import Python's string module to help remove punctuation marks.

Import NLTK's stopwords list.

Tokenize text, convert it to lower case and only add the words if they are not included in the stoplist and are not punctuation marks.

```
import nltk
import string
from nltk import word_tokenize
from nltk.corpus import stopwords

def process(text):
    stoplist = set(stopwords.words('english'))
    word_list = [word for word in word_tokenize(text.lower())
                 if not word in stoplist and not word in string.punctuation]
    return word_list

word_list = process(documents.get("1"))
print(word_list)
```

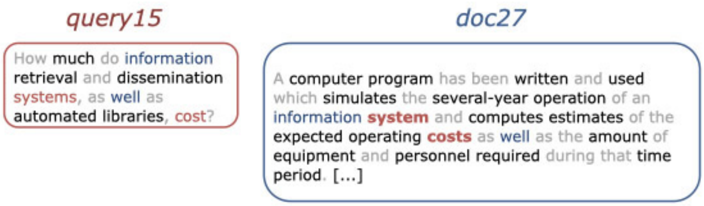
Check the result of these preprocessing steps on some documents or queries, for example, document I.

If you run the code from Listing 3.6 to preprocess document 1, it will return the list of words including ["18", "editions"', "dewey", "decimal", "classifications", . . .] for the original text of document 1 from Table 3.3 that goes as "18 Editions of the Dewey Decimal Classifications . . . " That is, the preprocessing step helps removing the stopwords like "of" and "the" from the word list.

3.2.2 Matching forms of same word: Morphological processing

One effect that stopwords and punctuation marks removal has is optimization of search algorithm. The words that do not matter much are removed, so the computational resources are not wasted on them. In general, the more concise and the more informative the data representation is, the better.

This brings us to the next issue. Take a look at figure 3.7 illustrating the query with id 15 and document with id 27, which are a match according to the ground truth mappings.


query15
doc27

How much do information retrieval and dissemination systems, as well as automated libraries, cost?

A computer program has been written and used which simulates the several-year operation of an information system and computes estimates of the expected operating costs as well as the amount of equipment and personnel required during that time period. [...]

Figure 3.7 The words highlighted in blue will be matched between the query and the document; the ones in red will be missed.

As Figure 3.7 shows, after removing the stopwords and punctuation marks, the algorithm will be able to match the query to the document on some words, but will miss others. For instance, it won't be able to tell that "system" and "systems" as well as "cost" and "costs" essentially represent the same words in different forms. In this particular case, the query and document will still be matched on such words as "information" or "well", but the *degree* to which their contents overlap will be lower. As you will see shortly, such degree matters, because it allows you to reason about the relevance ranking of the document. In addition, in other cases, the query-document correspondence might not be established at all, if the only relevant words are used in different forms in the query and the document.

The reason for this mismatch is that words may take different forms in different contexts: some contexts may require a mention of a single object or concept like "system", while others may need multiple "system**s**" to be mentioned. Such different forms of a word that depend on the context and express different aspects of meaning, for instance, multiplicity of "systems", are technically called *morphological forms,* and when you see a word like "systems" and try to match it to its other variant "system" you are dealing with *morphology*. English is a relatively "lucky" case—it is not very rich in morphology. That is, it has a limited variety of word forms. Other languages distinguish between many more morphological forms, whereas English forms may be concisely described as in table 3.5.

Table 3.5 Concise Description of English Morphological System

Type of word	Example	Type of form
Nouns (words that denote objects, people, animals, concepts)	*system*, *man*, *mouse*, *phenomenon*	Base form: singular form
	*system**s***, *m**e**n*, *m**ic**e*, *phenomen**a***	Plural form
Verbs (words that denote actions, states)	*be*, *have*, *retrieve*, *sing*	Base form: infinitive
	is, *ha**s***, *retriev**es***, *sing**s***	Third-person form (used with "he/she")
	was / were, *ha**d***, *retriev**ed***, *s**a**ng*	Past tense form
	been, *ha**d***, *retriev**ed***, *s**u**ng*	Past participle form (used in phrases like "have been")
	*be**ing***, *hav**ing***, *retriev**ing***, *sing**ing***	Progressive form (as in "I am having a nice time"

The *base form* in table 3.5 is always the most basic form of the word—it is the starting point for any further changes and aspects of meaning, and it is also the word form that you would find in a dictionary if you were looking up for a word. So the process of mapping different forms of the word to its most basic one is similar to that of looking up for words in a dictionary. Imagine you wanted to know what "sung" meant. Your best strategy would be to look up straight away for "sing". Similarly, the search algorithm would benefit from mapping “sing”, “sang”, and “sung” to the same word, by default the most basic one—“sing”. Now, check your understanding of this processing step with the following exercise.

Exercise 4

What base word forms will you end up with after processing this text:

“A computer program has been written and used which simulates the several-year operation of an information system and computes estimates of the costs as well as the amount of equipment and personnel required during that time period.”

Solution:

Conversion of this piece of text to the base forms should result in “A computer program *have* *be* *write* and *use* which *simulate* the several-year operation of an information system and *compute* *estimate* of the *cost* as well as the amount of equipment and personnel *require* during that time period.”

Such a preprocessing step is quite useful—it results in a more compact search space than the original where different forms of the same word are mapped together to a single dictionary form. How can a machine perform such a conversion? The solution would be to keep a large dictionary of all known words in a language and try to map the different forms to the known base forms in this dictionary. You might see straight away that there are potential problems with such an approach. To begin with, it is resource-intensive, because it has to keep a dictionary for the look-up. Moreover, it would not scale, because it is hard to make sure that a dictionary indeed contains *all* the words in a language. Human languages are creative and new words tend to crop up on a regular basis, so no dictionary can cover all words in a language, past, present and future. Can you do better than relying on a dictionary then?

In fact, there is another option for word form preprocessing that is called *stemming*. Stemming takes word matching one step further and tries to map related words across the board, and this means not just the forms of the very same word. For that, the stemmers rely on a set of rules that try to reduce the related words to the same basic core. Such rules rely on the idea that even though languages may add new words and borrow words from other languages, they will still apply the very same set of rules to build morphological forms for such new additions. For example, the word “selfie” has been relatively recently invented in English, but if you take multiple photo**s** you will still use the same rules of language and say that you took multiple “selfie**s**”. Similarly, if you use Twitter you might “tweet” once in a while, or you might be “tweet**ing**” pretty regularly, just like you might write an odd blog post once a year or you might be an active blogger, who is constantly writ**ing** new blog posts.

How does stemming work then and what resulting forms does it produce? Take the verb *retrieve* as an example. You can make a whole range of forms out of it, including *retrieving, retrieves,* and *retrieved,* just as table 3.5 shows. However, if you want to describe the process of retrieving something, you use the word *retrieval*. *Retrieval* is derived from *retrieve*, and the stemmer helps identifying this connection by reducing all related words to their common core that is called *stem*, thus the name for the tool. The rule in that particular case will define that the words ending in –al can be mapped to the words without –al: in fact, forming new words with an addition of –al is a productive pattern in English (*remove* + *-al* = *removal*, *approve* + *-al* = *approval*, *deny* + *-al* = *denial*, and so on). The stem in {*retrieve, retrieves, retrieved, retrieving, retrieval*} is *retriev*. So here is the difference with the technique that you used before—stemming might result in non-words, as for example, you won't find a word like *retriev* in a dictionary. To provide you with a couple of other examples, the stem for {*expect, expects, expected, expecting, expectation, expectations*} is *expect* and the stem for {*continue, continuation, continuing*} is *continu*—see figure 3.8.

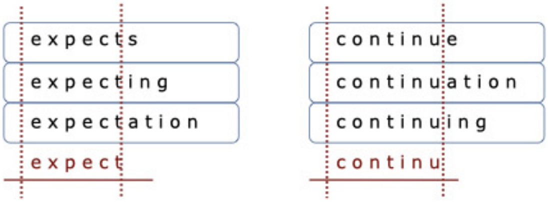


Figure 3.8 Stemming applied to different groups of related words.

Note that the stemmer tries to identify which part of the word is shared between the different forms and related words and returns this part as a stem by cutting off the differing word endings.

Now let's implement the stemming preprocessing step using NLTK's stemming functionality. NLTK provides a suite of different stemming tools,[20] and in this chapter you will use one of the most accurate of them—the Lancaster Stemmer, as shown in the following listing.[21]

Listing 3.7 Preprocessing: Stemming

```
import nltk
import string
from nltk import word_tokenize
from nltk.corpus import stopwords
from nltk.stem.lancaster import LancasterStemmer

def process(text):
    stoplist = set(stopwords.words('english'))
    st = LancasterStemmer()
    word_list = [st.stem(word) for word in word_tokenize(text.lower())
                 if not word in stoplist and not word in string.punctuation]
    return word_list

word_list = process(documents.get("27"))
```

Import the tools, including the stemmer.

Initialize the LancasterStemmer.

Apply stemming to the preprocessed text.

[20] Check the documentation here: https://www.nltk.org/api/nltk.stem.html.

[21] The source code can be found here: https://www.nltk.org/_modules/nltk/stem/lancaster.html.

```
print(word_list)
word_list = process("organize, organizing, organizational, organ, organic,
     organizer")
print(word_list)
```

As before, check the results on some document or query; you can also pass in a list of words. directly. Do the results correspond to your expectations?

When you run the previous code above on a particular document, for example, document 27, the function `process` receives the following text as input:

Input = “Cost Analysis and Simulation Procedures for the Evaluation of Large Information Systems …”[22]

As an output, it returns the following list of stems:

Output = ['cost', 'analys', 'sim', 'proc', 'evalu', 'larg', 'inform', 'system', …]

Stem ‘analys’ for “analysis” will help the algorithm to map “analysis” to such words as “analyse” (in British spelling), “analysing”, “analyst” and so on; stem ‘proc’ will help the algorithm group words like “procedure”, “process” and “processing” and so on. So this step results in even more compact search space and helps establish useful correspondences between similar words that should help the search algorithm find content related to the information need more effectively.

Now, what happens when you run this function on the input=[‘organize’, ‘organizing’, ‘organizational’, ‘organ’, ‘organic’, ‘organizer’]? Intuitively, the words {*organize, organizing, organizational, organizer*} belong to one group and you might expect them to be processed as *organiz*, while {*organ, organic*} belong to another group which should result in something like *organ*. However, the actual output returned by the function `process` is a list of identical stems for all the words in the input list: ['org', 'org', 'org', 'org', 'org', 'org']. This example is used here rather as a warning about the way stemmers work. While they are useful in mapping related words to each other, sometimes they might produce an unexpected output and map unrelated words together. This happens because stemmers sometimes go too far in their attempt to establish the correspondences. As the stemmer blindly applies a rather general set of rules to all examples, some of these rules overgeneralize.

The following list explains how the output for this list of words is produced, step-by-step:[23]

- {*organize, organizing, organizational, organizer*} may all be reduced to organiz by application of the following rules: -ing (as in make –> making), -ational (operate -> operational), and -er (produce –> producer).
- The mapping between *organ* and *organic* is explained by the addition of –ic as in acid –> acidic.
- The less straightforward mapping between *organ* and *organize* is established through the application of ending –ize, as in *modern* –> *modernize.*

[22] As before, we use “…” to indicate that there are more words in the input and more stems returned in the output

[23] Note, that this list of rules explaining how the output is arrived at is advanced content. You can consult with this list in case you are interested in what happens “behind the scenes” when you apply the stemming algorithm. However, understanding or knowing these rules is not critical for the application of the stemming algorithm itself.

- Finally, *organ* gets mapped to *org* by the application of –an. It is, in fact, applicable in cases like *Italy* –> *Italian* and *history* –> *historian*, that is, to form words describing properties and qualities (such words are called adjectives) from words (nouns) that describe people in cases when these words are related in meaning.

So, technically, the last two rules should not be applied to map cases like *organ* –> *organize*, because the two words do not mean similar things, and it would be better for the applications like search algorithm to make the distinction between the two groups of words {*organize, organizing, organizational, organizer*} and {*organ, organic*}. However, unfortunately, the stemmer algorithm does not take into account what words mean, so once in a while it may make mistakes and connect unrelated words. Figure 3.9 visualizes all the rules that are applied to this set of words, showing the resulting stems and the endings of the words that are cut off by the application of different rules.

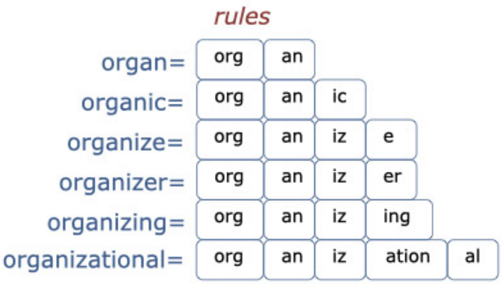
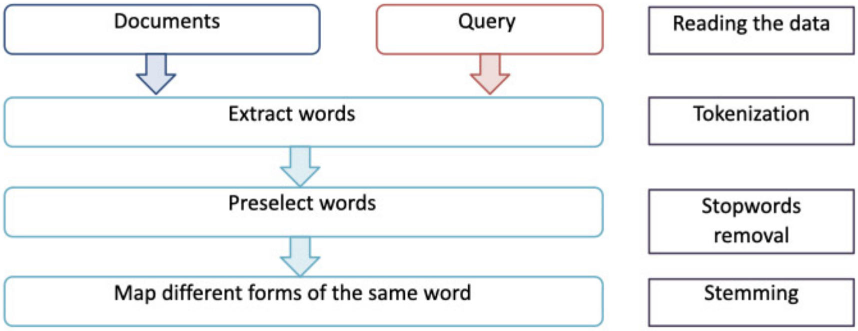


Figure 3.9 **The full analysis of rules applied to the example including *organ* and *organize*.**

Now, why should you be aware of this peculiarity of the stemmer algorithms? Since in some cases the stemmer would map together words that are not closely related to each other, your search algorithm might consider documents talking about *organic products* somewhat relevant for the query that asks about *organizational skills*. This is something to keep in mind. In general, because the queries are mapped to the relevant documents on the basis of more than one word from the query, such incorrect mappings are usually outweighed by the relevance of other words.

Before we move on, let's summarize which steps of the algorithm you have implemented so far. You have read the data, initialized the data structures and tokenized the texts, removed stopwords, and applied the stemming preprocessing.



3.3 Information weighing

Another problem with the simple Boolean search algorithm implemented in this chapter is that it can only return a list of documents that contain some or all of the words from the query, but it cannot tell which of the documents are more relevant. You've seen before that when you run the algorithm from listing 3.5, for most queries it returns a huge number of documents. Stopwords removal helps filter out the less relevant words, while stemming helps find the correspondences between the related words, which alleviates some of these issues. However, your algorithm still returns the relevant documents as an unsorted list. Without some measure of relevance and relevance, ordering it would still be time-consuming to look through all the documents returned by the algorithm. What could serve as such a measure of relevance? Let's look into an example in figure 3.10.

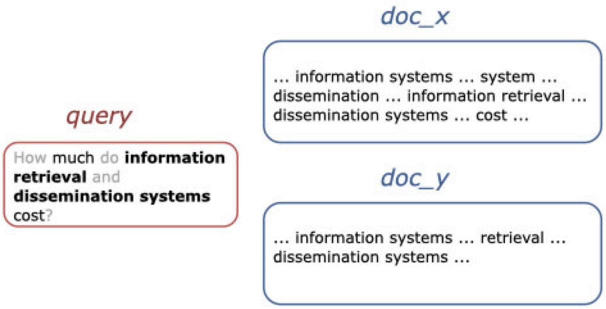

doc_x
... information systems ... system ... dissemination ... information retrieval ... dissemination systems ... cost ...
query
How much do **information retrieval** and **dissemination systems** cost?
doc_y
... information systems ... retrieval ... dissemination systems ...

Figure 3.10 An example of the different distribution of query keywords in documents.

Suppose you try to find documents most relevant to the given query. After stopwords and punctuation marks removal you end up with the query words—let's call them *keywords*—consisting of {*much, information, retrieval, dissemination, systems, cost*}. Which of the two documents appears to be more relevant? Document doc_x does not only contain more keywords than doc_y, each keyword also occurs more times, so it would be reasonable to assume that doc_x is more relevant—given a choice between these two documents, you should start with doc_x if you want to find the answer to the query. How can we take the factors like more keywords and higher number of occurrences into account?

3.3.1 Weighing words with term frequency

The first requirement, that you should take into account all keywords, suggests that you need to keep track of the words used in the queries and documents. The second requirement, that the number of occurrences of each of the keywords matters, suggests that you need to count the number of occurrences rather than simply register presence or absence of a keyword. You can achieve this by keeping the number of occurrences for the keywords in a table, or translating this into a Python data structure you can use a dictionary that will allow you to keep track of which counts correspond to which keywords. For instance, the example from figure 3.10 will result in table 3.6.

Table 3.6 The Keyword Occurrences Merged into a Shared Representation

	much	information	retrieval	dissemination	system(s)	cost
query	1	1	1	1	1	1
doc_x	0	2	1	2	3	1
doc_y	0	1	1	1	2	0

The correspondent Python dictionaries will be as follows:

> Query={much:1, information:1, retrieval:1, dissemination:1, systems:1, cost:1}
> Doc_x={much:0, information:2, retrieval:1, dissemination:2, systems:3, cost:1}
> Doc_y={much:0, information:1, retrieval:1, dissemination:1, systems:2, cost:0}

This approach, based on calculating frequency of occurrence, corresponds to the well-known technique in Information Retrieval called *term frequency (tf)*. It relies on the idea that the more frequently the word (term) is used in a document, the more relevant this document becomes to the query. We will use the word *term* instead of "word" from now on following this widely accepted convention: after all, since you apply stemming, not all keywords keep to be proper "words" anymore (think of the case of *retriev*). Listing 3.8 shows how to implement this step.

Listing 3.8 Code to estimate term frequency in documents and queries

```
def get_terms(text):
    stoplist = set(stopwords.words('english'))
    terms = {}
    st = LancasterStemmer()
    word_list = [st.stem(word) for word in word_tokenize(text.lower())
        if not word in stoplist and not word in string.punctuation]
        for word in word_list:
      terms[word] = terms.get(word, 0) + 1
    return terms
```

Apply all the preprocessing steps as before.

Estimate the counts for each term and populate the dictionary.

```
doc_terms = {}
qry_terms = {}

for doc_id in documents.keys():
    doc_terms[doc_id] = get_terms(documents.get(doc_id))
for qry_id in queries.keys():
    qry_terms[qry_id] = get_terms(queries.get(qry_id))
```

Populate the term frequency dictionaries for all documents and all queries.

```
print(len(doc_terms))
print(doc_terms.get("1"))
print(len(doc_terms.get("1")))
print(len(qry_terms))
print(qry_terms.get("1"))
print(len(qry_terms.get("1")))
```

Check out the results: you can print out the length of the resulting data structures (this shouldn't change from before—1460 for the documents, 112 for the queries), the term frequency dictionaries for a specific document or query (e.g., the first ones in the set), and the length of these dictionaries (it should be 43 terms for the document 1 and 14 terms for the query 1).

Now, let's represent all queries and all documents in the same shared space. For example, table 3.6 represents one query and two documents in a space where the columns of the table are shared among all three. In the Python data structure, each of these columns represents a separate dimension. For instance, column 1 keeps the counts of the term "much" across the query and both documents, column 2 keeps the counts for "information", and so on; similarly, the Python data structures keep these counts in the first two dimensions as:

Query={much:1, information:1, …}
Doc_x={much:0, information:2, …}
Doc_y={much:0, information:1, …}

Now let's add all terms from the data set as columns, and keep the counts for each of them in each query and each document as rows. In terms of Python data structures, this means that each document and each query will keep the whole dictionary of terms in the collection with the associated term frequencies. Listing 3.9 presents this step.

Listing 3.9 Code to represent the data in a shared space

```
def collect_vocabulary():
    all_terms = []
    for doc_id in doc_terms.keys():
        for term in doc_terms.get(doc_id).keys():
            all_terms.append(term)
    for qry_id in qry_terms.keys():
        for term in qry_terms.get(qry_id).keys():
            all_terms.append(term)
    return sorted(set(all_terms))

all_terms = collect_vocabulary()
print(len(all_terms))
print(all_terms[:10])

def vectorize(input_features, vocabulary):
    output = {}
    for item_id in input_features.keys():
        features = input_features.get(item_id)
        output_vector = []
        for word in vocabulary:
            if word in features.keys():
                output_vector.append(int(features.get(word)))
            else:
                output_vector.append(0)
        output[item_id] = output_vector
    return output

doc_vectors = vectorize(doc_terms, all_terms)
qry_vectors = vectorize(qry_terms, all_terms)

print(len(doc_vectors))
print(len(doc_vectors.get("1460")))
print(len(qry_vectors))
print(len(qry_vectors.get("112")))
```

First, collect the shared vocabulary of terms used in documents and queries; return it as a sorted list for convenience.

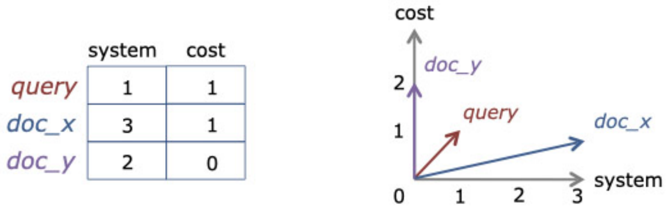
Print out the length of the shared vocabulary (you should end up with 8881 terms in total) and check the first several terms in the vocabulary.

Now each query and each document can be represented with a dictionary with the same set of keys—the terms from the shared vocabulary. The values will either be equal to the term frequency in the particular query and document or will be 0 if the term is not in the query or document.

Using the vectorize method you can represent all queries and documents in this shared space.

Print out some statistics on these data structures: you should still have 1460 doc_vectors and 112 qry_vectors, with 8881 terms each.

Now, another way to think about each of these term dictionaries associated with each document and each query is as vectors: that is, each document and each query is represented as a vector in a shared space, with the number of dimensions equal to the length of the shared vocabulary (8881) and the term frequencies in each dimension representing the coordinates. This may remind you of the discussion on vectors in chapter 1, and figure 3.11 reinterprets the query, and two documents from table 3.6 as vectors in two dimensions associated with terms "system" and "cost" (but you can imagine how these vectors are extended to other dimensions, too).[24]



	system	cost
query	1	1
doc_x	3	1
doc_y	2	0

Figure 3.11 Vector representation of the query and two documents along two dimensions.

Now you can estimate the relevance, or similarity, of the query and documents using the distance between them in the vector space. But before you do that, there is one more observation due.

3.3.2 Weighing words with inverse document frequency

In a collection of documents, you are working with, some terms are much *more frequently used across all documents* than others. For instance, since this is a collection of articles on information science and information retrieval systems, such terms as "information" or "system" may occur in many documents while other terms like "cost" may occur in fewer documents. Which ones are more helpful in search then? Imagine that you were to find the relevant documents for the query 15, "How much do information retrieval and dissemination systems cost?" If "information" and "system" occur in lots of documents, then you better focus your attention on those documents that contain other terms from the query, such as "dissemination" and "cost", because it is those documents that contain these words that are more relevant. In other words, you would like to give these rarer terms like "dissemination" and "cost" higher weight so that the search algorithm knows it should trust their vote for relevance more. The most straightforward way to assign such weights to the terms is to make it proportionate to the number of documents where the term occurs. The higher the number of documents that contain the term, the lower its discriminative power, and therefore the lower the weight that the term should get.

Take the term "inform" as an example (this is a stem for such words as "inform" and "information"). It occurs in 651 out of 1460, so its *document frequency (df)* equals 651/1460 ? 0.45. On the other hand, the term "dissemin" (stem of "dissemination")

[24] You may recall that figure 1.5 uses a similar representation for a different example.

only occurs in 68 documents, so its df = 68/1460 ? 0.05. “Dissemin” is a more valuable term for the search algorithm because it is rare: if a query contains it, the documents that also contain, it should be given preference. To assign a higher weight to “dissemin” than to “inform”, let’s take the inverse document frequency (idf): idf(“inform”) = 1/0.45 ? 2.22, idf(“dissemin”) = 1/0.05 ? 20. These weights show that the rare term “dissemin” is almost 10 times more important than the much more frequent term “inform”. There are two more things to take into account here:

- First, some terms from the shared vocabulary may not occur in any of the documents, so their df will be 0. To avoid division by 0, it is common to *smooth* the counts. To calculate the idf, take (df+1) rather than df, that is, idf = 1/(df+1), so you will never have to divide by zero, and the absolute values of idf won’t change much.
- Second, it is common to “tone down” the differences in absolute counts, because the difference between very rare and very frequent terms might be huge, especially in large collections. It is assumed that the weight given to the terms should increase not linearly (that is, by one with each document) but rather sub-linearly (that is, more slowly). Logarithmic function achieves this effect: the relative order of the term’s importance doesn’t change, while the absolute number does.

To put all the components together, here are the idf values for the terms “inform” and “dissemin” in this collection:

idf(“inform”) = $\log_{10}$(1460/(651+1)) ≈ 0.35
idf(“dissemin”) = $\log_{10}$(1460/68+1)) ≈ 1.33

As you can see, the difference is still significant, but the counts are more comparable. The general formula then is:

idf(term) = $\log_{10}$(N / (df(term) + 1)

where N is the total number of documents in the collection.

Exercise 5

What are the inverse document frequency (idf) values for the following terms based on the number of documents (df for document frequency) they occur in:

df(“system”) = 531; df(“us” = stem of “use”) = 800; df(“retriev”) = 287; df(“cost”) = 137

Solution:

idf(“system”) = $\log_{10}$(1460/(531+1)) ≈ 0.44
idf(“us”) = $\log_{10}$(1460/(800+1)) ≈ 0.26
idf(“retriev”) = $\log_{10}$(1460/(287+1)) ≈ 0.71
idf(“cost”) = $\log_{10}$(1460/(137+1)) ≈ 1.02

Now, if a particular document contains two occurrences of the term “cost”, its idf-weighed value will be 2*1.02=2.04, while if it contains two occurrences of the term “system”, its idf-weighed value will be 2*0.44=0.88, so despite the same term frequencies the more informative term “cost” will get higher overall weight. For instance, here is how idf weighing will change the weights of the terms in the documents from table 3.6.

Table 3.7 Idf Weighing Applied to the Term Frequencies in the Two Documents

	system(s)	cost
query	1	1
doc_x	3*0.44=1.32	1*1.02=1.02
doc_y	2*0.44=0.88	0

Listing 3.10 shows how to implement this in Python.

Listing 3.10 Code to calculate and apply inverse document frequency weighing

```
import math

def calculate_idfs(vocabulary, doc_features):
    doc_idfs = {}
    for term in vocabulary:
        doc_count = 0
        for doc_id in doc_features.keys():
            terms = doc_features.get(doc_id)
            if term in terms.keys():
                doc_count += 1
        doc_idfs[term] = math.log(float(len(doc_features.keys()))/float(1 +
     doc_count), 10)
    return doc_idfs
```

Estimate idf values for each term in the vocabulary by counting how many documents contain it.

Apply the idf formula from above.

```
doc_idfs = calculate_idfs(all_terms, doc_terms)
print(len(doc_idfs))
print(doc_idfs.get("system"))
```

Check out the results: you should have idf values for all 8881 terms from the vocabulary; the idf for any particular term should coincide with your estimates as with Exercise 5.

```
def vectorize_idf(input_terms, input_idfs, vocabulary):
    output = {}
    for item_id in input_terms.keys():
        terms = input_terms.get(item_id)
        output_vector = []
        for term in vocabulary:
            if term in terms.keys():

     output_vector.append(input_idfs.get(term)*float(terms.get(term)))
            else:
                output_vector.append(float(0))
        output[item_id] = output_vector
    return output

doc_vectors = vectorize_idf(doc_terms, doc_idfs, all_terms)
```

Define a method to apply idf weighing to the input_terms (in particular, to doc_terms) data structure.

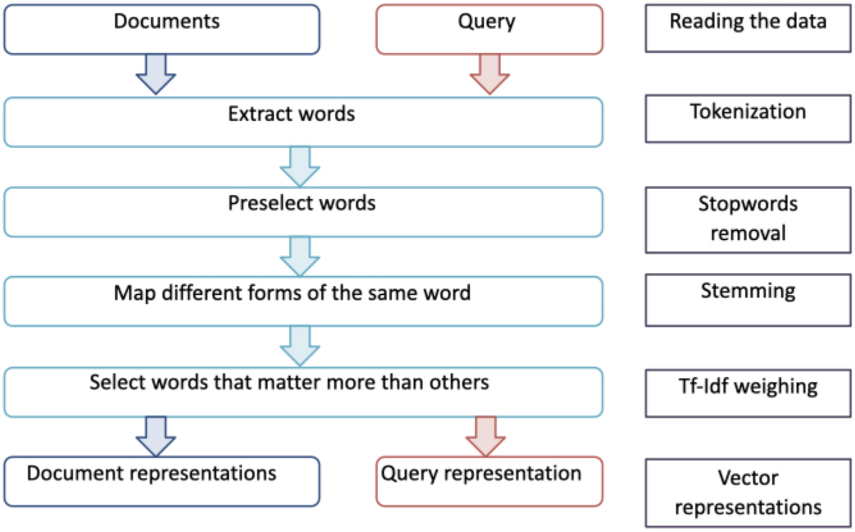
For that, multiply the term frequencies with the idf weights if the term is present in the document; otherwise, its term frequency stays 0.

Apply idf weighing to doc_terms.

```
print(len(doc_vectors))
print(len(doc_vectors.get("1460")))
```

Print out some statistics: the dimensionality of the data structure should still be 1460 documents by 8881 terms.

Let's now summarize what you have implemented so far.



3.4 Practical use of the search algorithm

Now that the documents and queries are represented in the shared search space, it's time to run the search algorithm, find the most relevant documents for each query and evaluate the results.

3.4.1 Retrieval of the most similar documents

How can you estimate query to document similarity based on the vector representations? In chapter 1 we discussed that the similarity can be interpreted as distance in space defined by the query and document vectors. Here is a refresher:

- Each document and each query are represented as vectors in a shared space, with the dimensions representing terms and coordinates representing weighted term counts
- Similarity is estimated using distances in this shared space. To eliminate the effect of different lengths (as queries are traditionally much shorter than documents), it is more reliable to use the cosine of the angle between the vectors, because it normalizes the distance with respect to the different lengths of the vectors. Because of this normalization step, estimating the angle between the vectors of different lengths is equivalent to estimating the distance between the vectors of same length.[25]

[25] Figure 1.8 visualizes this idea.

- The higher the cosine, the more similar the query and the document are.
- The cosine can be estimated using the formula:

```
cosine(vec1, vec2) = dot_product(vec1,vec2)/(length(vec1)*length(vec2))
```

Let's calculate the cosine between the query and documents doc_x and doc_y from table 3.6 (using only tf and ignoring the idf weighing for the sake of simplicity here):

```
cosine(query, doc_x) = (0+2+1+2+3+1)/(sqrt(6)*sqrt(19)) ≈ 0.84
cosine(query, doc_y) = (0+1+1+1+2+0)/(sqrt(6)*sqrt(7)) ≈ 0.77
```

Based on these results, doc_x is more similar to the query than doc_y, so if you apply the cosine similarity estimation for the given query to the set of two documents, you should return them ordered as (doc_x, doc_y). As it is doc_x that is more similar and thus more relevant to the query, if you want more relevant information you should start with doc_x.

Let's apply cosine similarity to the input queries and documents in the dataset and return the resulting lists of relevant documents ordered by their relevance scores, that is cosine similarity values, as shown in the following listing.

Listing 3.11 Code to run search algorithm for a given query on the set of documents

operator's itemgetter functionality is helpful when you want to sort Python dictionaries by keys or values.

Initialize the query by selecting an example with a particular qry_id, for example, query 3.

For each document in the set of documents, calculate cosine similarity between the input query and the document, and store the document id as the key and cosine as the value in results dictionary.

```
from operator import itemgetter

query = qry_vectors.get("3")
results = {}

for doc_id in doc_vectors.keys():
    document = doc_vectors.get(doc_id)
    cosine = calculate_cosine(query, document)
    results[doc_id] = cosine

for items in sorted(results.items(), key=itemgetter(1), reverse=True)[:44]:
    print(items[0])
```

Sort the results dictionary by cosine values (key=itemgetter(1)) in descending order starting with the highest value (reverse=True). Return the top n ones—here, we use 44 because that is the number of relevant documents for query 3 according to the gold standard. Note that sorted function returns tuples of (document_id, similarity score), so if you want to print out the document's ids only, use items[0].

This piece of code returns a list of 44 documents identified by the search algorithm as relevant to query 3, ordered by cosine similarity starting with the most relevant one. A quick glance over the first 10 returned documents (that is how many you would see on the first page in the Internet browser) shows that 8 out of 10 documents are also included in the gold standard. Perhaps even more importantly the top two documents in the returned list are relevant according to the gold standard—and you might not even need to look any further than the first couple of documents! This looks like a good result, but how can you get a more comprehensive overview of the results across the board, i.e. over multiple queries?

3.4.2 Evaluation of the results

If you are building a search algorithm as part of some application for the users, it is key to the success of your application that the users are satisfied with the results. If you are building an application for your own needs, it is important to be able to measure whether it is doing a good job. How can you measure if the users or yourself are satisfied with the results?

In the previous step, you added similarity estimation to your search algorithm that allows it to return the results as an ordered list. Suppose you are looking for the documents related to query 3, "What is information science? Give definitions where possible." According to the gold standard, there are 44 documents in this set that match this query. In some situations, you might be interested in exhaustive search, that is, you will measure the success of your algorithm by its ability to find *all* 44 documents. However, in most situations what you would like is for the algorithm to return the relevant documents at the top of the list: it is more important that the first document returned by the algorithm is relevant than whether the 44th document is relevant. Often, if the very first document is relevant to your query, you will read no further. For example, how often do you check the second page of results on Google?

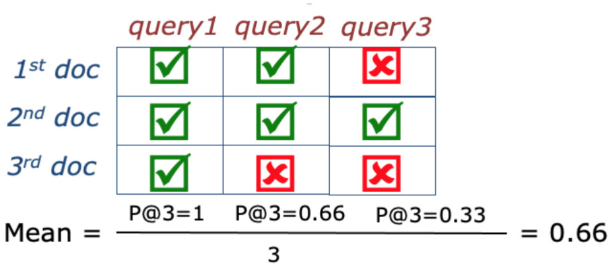
Since the number of relevant documents in the gold standard varies for different queries—for example, it is 44 for query 3 but there is only 1 relevant document for query 6—you may prefer to set the number of top documents to be returned by your algorithm in advance. In addition, it is rarely the case that users are interested in documents after the first several relevant ones, so returning something between top 3 to top 10 documents would be reasonable. The number of documents that are returned by the algorithm among those top-3 (top-10) that are also included as relevant in the gold standard is called *true positives*—they are truly relevant documents actually identified by your algorithm. The proportion of true positives to the total number of documents returned by the algorithm is called *precision*, and if you predefine the number of returned documents to be *k* this measure is called *precision@k* (for example, precision@3 or precision@10). Another example is that the code in listing 3.11 returns 8 relevant documents in the top 10 ones—its precision@10 equals 0.8. That is, precision@10 is defined as:

```
precision@10 = (true positives among the top 10 documents) / 10 =
(number of documents that are actually relevant among the top 10) / 10
```

And in the general case, precision@k is:

```
precision@k = (true positives among the top k documents) / k =
(number of documents that are actually relevant among the top k) / k
```

The higher the precision, the better the algorithm you have built; however, the results may also depend on the quality of the dataset and the queries themselves. For example, since there are 44 matching documents for query 3 in the dataset and only 1 matching document for query 6 it would be much easier for the algorithm to find relevant documents for query 3. If you want the results to be more objective, it is useful to evaluate precision across all queries. This is called *mean precision*, because it takes the mean across all queries. For example, if the top 3 results for the first query are all relevant, precision@3=1; if only 2 are relevant, precision@3=0.66; for only one relevant result, precision@3=0.33. If you estimate the mean precision across 3 queries with such results, it would be equal to 0.66, as figure 3.12 shows.



	query1	query2	query3
1st doc	☑	☑	☒
2nd doc	☑	☑	☑
3rd doc	☑	☒	☒

$$\text{Mean} = \frac{\text{P@3=1} \quad \text{P@3=0.66} \quad \text{P@3=0.33}}{3} = 0.66$$

Figure 3.12 **Mean precision@3 per 3 queries.**

Thus, the mean precision@k can be estimated as:

```
Mean_p@k = sum_over_queries(p@k)/number_of_queries =
sum_over_queries(true_positives/k)/number_of_queries
```

You might also be interested in knowing how often the top results contain at least one relevant document: in the case exemplified in figure 3.12 the user will be able to find at least one relevant document in the top 3 results, which is quite useful, therefore this ratio will be equal to 1. Listing 3.12 shows how these measures can be implemented in Python.

Listing 3.12 Code to estimate precision@k and ratio of cases with at least one relevant document

```
def calculate_precision(model_output, gold_standard):
    true_pos = 0
    for item in model_output:
        if item in gold_standard:
            true_pos += 1
    return float(true_pos)/float(len(model_output))

def calculate_found(model_output, gold_standard):
    found = 0
    for item in model_output:
        if item in gold_standard:
            found = 1
    return float(found)
```

Define a method to estimate precision.

Precision equals to the number of relevant documents from the gold standard that are also returned in the top-k results by the algorithm.

Alternatively, give the algorithm some credit if at least one document in the top k is relevant.

```
precision_all = 0.0
found_all = 0.0
for query_id in mappings.keys():
    gold_standard = mappings.get(str(query_id))
    query = qry_vectors.get(str(query_id))
    results = {}
    model_output = []
    for doc_id in doc_vectors.keys():
        document = doc_vectors.get(doc_id)
        cosine = calculate_cosine(query, document)
        results[doc_id] = cosine
    for items in sorted(results.items(), key=itemgetter(1),
     reverse=True)[:3]:
        model_output.append(items[0])
    precision = calculate_precision(model_output, gold_standard)
    found = calculate_found(model_output, gold_standard)
    print(f"{str(query_id)}: {str(precision)}")
    precision_all += precision
    found_all += found

print(precision_all/float(len(mappings.keys())))
print(found_all/float(len(mappings.keys())))
```

Gold standard is the list of relevant document ids that can be extracted from the mappings data structure.

Calculate mean values across all queries.

For each document, estimate its relevance to the query with cosine similarity as before.

Sort the results and only consider top-k (e.g., top-3) most relevant documents.

Accumulate evaluation values across all queries; track the results by a printout message.

In the end, estimate the mean values for all queries.

According to the results, on some queries the algorithm performs very well. A printout message "`1: 1.0`" shows that all 3 documents returned for query 1 are relevant, making precision@3 for this query equal to 1. However, on other queries the algorithm does not perform that well: for example, "`6: 0.0`", because there is only one document relevant for query 6 according to the gold standard, the algorithm fails to put it within the first 3 and gets a score of 0 for this query. The mean value of precision@3 for this algorithm is 0.39, and in 66% of the cases the algorithm finds at least one relevant document among the top 3.

If you are only interested in the proportion of cases when the top most relevant document identified by the algorithm is actually relevant you can calculate that modifying the code in listing 3.12 only slightly: instead of sorting all the results and then taking the top 3 it simply needs to identify and store a single best result. You can use this task as an exercise.

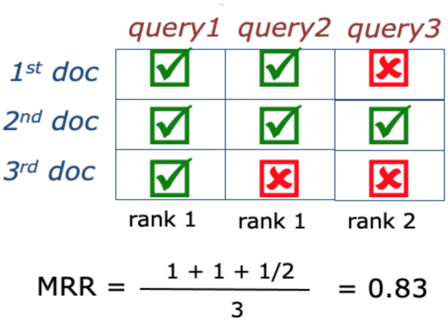
Exercise 6

Modify the code from listing 3.12 to calculate precision@1. That is, the mean value across the queries when the top 1 document returned by the algorithm is indeed relevant.

Solution:

First try to code this yourself, and then check the solution in the notebook provided with the book.

Finally, you may wish to know how highly, on the average, the algorithm places the relevant document in its ranking. This shows how far into the list of the returned results you should typically look to find the first relevant document. The measure that allows you to evaluate that relies on the use of the highest ranking of a relevant document identified by the algorithm. Since you already sort the returned documents by their relevance scores starting with the most relevant one, position one in this list is called first *rank*, position two—second rank, and so on. Take a look at the search results from figure 3.12 again.



	query1	query2	query3
1st doc	☑	☑	☒
2nd doc	☑	☑	☑
3rd doc	☑	☒	☒
	rank 1	rank 1	rank 2

$$\text{MRR} = \frac{1 + 1 + 1/2}{3} = 0.83$$

Figure 3.13 Ranks for the first relevant document for each of the three queries and mean reciprocal rank (MRR) across all three results.

- The first relevant documents for both query 1 and query 2 in this example are at position 1 in the ordered lists of returned documents, so their ranks are 1. For query 3, the first relevant document is found in the second position, which gives this result rank 2.
- However, returning the first relevant document at rank 1 is better than returning the first relevant document at any further position, so your measure should reflect this by assigning a higher score to the results with the rank 1. Just like with the inverse document frequency, if you take the inverse of the ranks, you will end up with exactly such measure: for both queries 1 and 2 the algorithm returns the best possible results by placing the first relevant document at position 1, so it gets a score of 1/1=1 for that. For query 3 it returns an irrelevant document in position 1 and the first relevant document in position 2. For that it gets only half the full score, 1/2. To summarize, to assign a score for the results for each query take the inverse of the rank of the first relevant document in the ordered list of results—this is called *reciprocal rank*:

```
reciprocal rank = 1 / rank of the first relevant document in the
ordered list of results
```

- Finally, as before, you want to have a comprehensive overview of the results across all queries, so you need to take a *mean reciprocal rank (MRR)* for the reciprocal ranks across all queries. For the example from figure 3.13, this will equal to (1 + 1 + 1/2) / 3 = 0.83.

```
MRR = sum_of_reciprocal_ranks_across_queries / number_of_queries
```

The best-case scenario is when the algorithm always puts a relevant document at the top of the list, so it assigns rank 1 in all cases. If the first relevant document is always found at rank 2, the mean will equal to 1/2; for the results at rank 3, the mean will be 1/3, and so on. The result that you get for the example from figure 3.13, MRR = (1+1+1/2)/3 = 0.83, lies between 1/2 and 1 and is closer to 1. This value shows, that on the average, the ranking of the first relevant document returned by the algorithm is between 1st and 2nd rank, and is in fact more often 1st than 2nd.

Listing 3.13 shows how to implement this measure in Python.

Listing 3.13 Code to estimate mean reciprocal rank

Sort the documents returned by the algorithm in descending order starting with the most similar. The position of each document in this sorted list is called rank.

As before, extract the list of gold standard mappings for each query.

```
rank_all = 0.0
for query_id in mappings.keys():
    gold_standard = mappings.get(str(query_id))
    query = qry_vectors.get(str(query_id))
    results = {}
    for doc_id in doc_vectors.keys():
        document = doc_vectors.get(doc_id)
        cosine = calculate_cosine(query, document)
        results[doc_id] = cosine
    sorted_results = sorted(results.items(), key=itemgetter(1), reverse=True)
    index = 0
    found = False
    while found==False:
        item = sorted_results[index]
        index += 1
        if index==len(sorted_results):
            found = True
        if item[0] in gold_standard:
            found = True
            print(f"{str(query_id)}: {str(float(1) / float(index))}")
            rank_all += float(1) / float(index)
print(rank_all/float(len(mappings.keys())))
```

You only need to find the first relevant document in this list, so set the flag found to False, and switch it to True as soon as you encounter the first relevant document, or reach the end of the list.

Increment index (rank) with each document in the results.

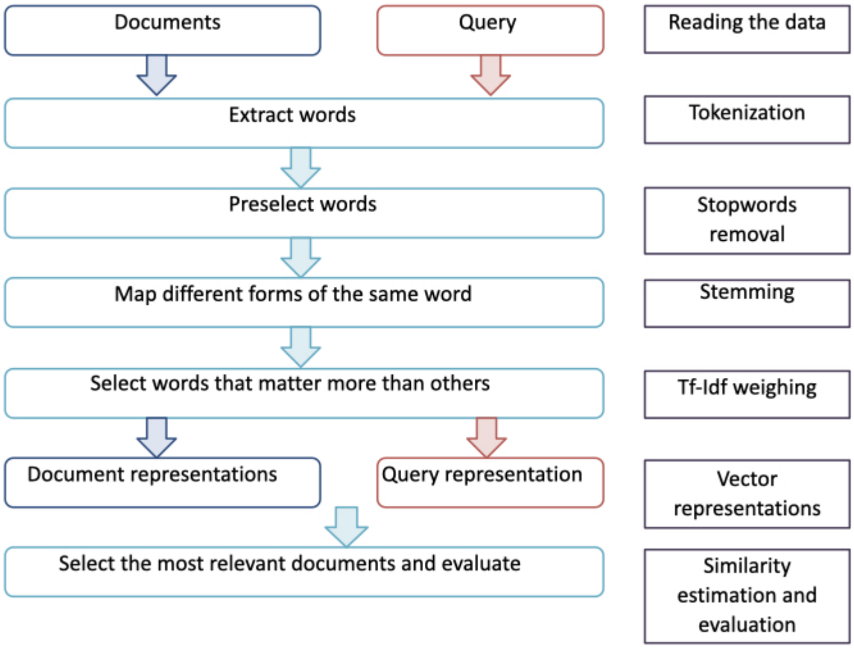
As before, the document id is the first element in the sorted tuples of (document_id, similarity score).

Estimate inverse of the rank.

Calculate and print out the mean value across all queries.

The result—mean reciprocal rank of 0.58—printed by this piece of code suggests that, on the average, the highest rank of a relevant document identified by this search algorithm is between 1st and 2nd. That is, you will often find the relevant results within the first pair of returned documents.

This concludes the implementation of the search algorithm, so let's summarize what steps you have implemented.



3.4.3 Deploying search algorithm in practice

Finally, once you have implemented the algorithm and decided on its components—the use of stemming, the type of term weighing, and so on—you can deploy it in practice. For instance, you may have your own data within your own project where searching for relevant information is useful. If you don't have such a project in mind, try applying the algorithm to another dataset anyway to practice the new skills: you can download one of the datasets from http://ir.dcs.gla.ac.uk/resources/test_collections/.

Exercise 7

Apply the search algorithm to your own data. For that, you would need to read in the files one by one as you did for spam filtering application in chapter 2.

Alternatively, apply the search algorithm to a different dataset from http://ir.dcs.gla.ac.uk/resources/test_collections/. Among these, the Cranfield dataset uses a similar data format to the CISI dataset, and is also relatively small and easy to work with.

Summary

Let's summarize what you have covered in this chapter:

- You have learned about search, or *information retrieval*, algorithms. Search algorithms are widely used in many applications, from search in an Internet browser to search for the relevant files on your personal computer. In addition, any application where there is a need to efficiently find relevant information in an arbitrarily large collection of documents would benefit from information retrieval algorithms. The valuable property of these algorithms is that they can sort the results in order of their relevance and ability to answer the information need (typically formulated as a query).

- Before you deploy the search algorithm in practice, it is a good idea to evaluate its performance on some annotated dataset. Such annotation is called *ground truth* or *gold standard*, and there are a number of publicly available datasets that you can use. In this chapter, you have used CISI, a collection of queries and abstracts from articles on information science and information retrieval.
- You have learned how to implement a simple *Boolean search algorithm*. This algorithm relies on the idea that any document that contains at least one word from the query is relevant for this query. However, it is unable to assess relative relevance of the documents and the results cannot be sorted.
- There is a particular set of words, including “a”, “the”, “in”, “at”, and the like, that are highly frequent in English—they occur in all or virtually all documents, so they are not informative for the search algorithms. In addition, they don’t capture the meaning, as they mainly link other words together and fulfill particular functions. Such words are commonly called *stopwords*, and they should be removed so that they do not mislead the algorithm. Many toolkits, including NLTK, contain standard stopwords list that you can use.
- Words in language may occur in several different forms. In English, this is relevant for nouns (words denoting objects, people, animals and abstract concepts) and verbs (words denoting actions and states). Mapping the different forms of a word to its base (dictionary) form allows the algorithm to establish useful correspondences and optimize the search space; one step further is to apply a set of rules to identify the correspondences across all related words. To link the related words to each other, use an NLP tool called *stemmer* that relies on a set of predefined rules.
- Documents that contain more occurrences of the query terms should be given preference as compared to the documents with lower number of occurrences. The number of occurrences represents *term frequencies (tf)*.
- Not all terms are equally important. Even after the stopwords are removed, there are still terms that are frequently used across all documents. Such terms are less discriminative, and their relative weights should reflect this. Use *inverse document frequency (idf)* to weigh the terms according to their distribution across the documents.
- To estimate the relevance of the documents to the queries in the collection, represent them in a shared search space, where each term stands for an individual dimension and term frequencies or tf-idf weighted counts are used as the coordinates.
- The relevance in the shared space can be estimated using *cosine similarity*, estimation of relevance in shared space and.

- The search algorithm can be evaluated with the use of one or more popular measures. For example, you can estimate the proportion of the relevant documents returned in the top-k results—this measure is called *precision@k*. Alternatively, you can measure the average highest rank for the relevant documents returned by the algorithm—this measure is called *mean reciprocal rank*.

index

manning