8 Text mining and text analytics

This chapter covers

- Understanding the importance of text mining
- Introducing the most important concepts in text mining
- Working through a text mining project

Most of the human recorded information in the world is in the form of written text. We all learn to read and write from infancy so we can express ourselves through writing and learn what others know, think, and feel. We use this skill all the time when reading or writing an email, a blog, text messages, or this book, so it's no wonder written language comes naturally to most of us. Businesses are convinced that much value can be found in the texts that people produce, and rightly so because they contain information on what those people like, dislike, what they know or would like to know, crave and desire, their current health or mood, and so much more. Many of these things can be relevant for companies or researchers, but no single person can read and interpret this tsunami of written material by themself. Once again, we need to turn to computers to do the job for us.

Sadly, however, the natural language doesn't come as "natural" to computers as it does to humans. Deriving meaning and filtering out the unimportant from

the important is still something a human is better at than any machine. Luckily, data scientists can apply specific text mining and text analytics techniques to find the relevant information in heaps of text that would otherwise take them centuries to read themselves.

Text mining or *text analytics* is a discipline that combines language science and computer science with statistical and machine learning techniques. Text mining is used for analyzing texts and turning them into a more structured form. Then it takes this structured form and tries to derive insights from it. When analyzing crime from police reports, for example, text mining helps you recognize persons, places, and types of crimes from the reports. Then this new structure is used to gain insight into the evolution of crimes. See figure 8.1.

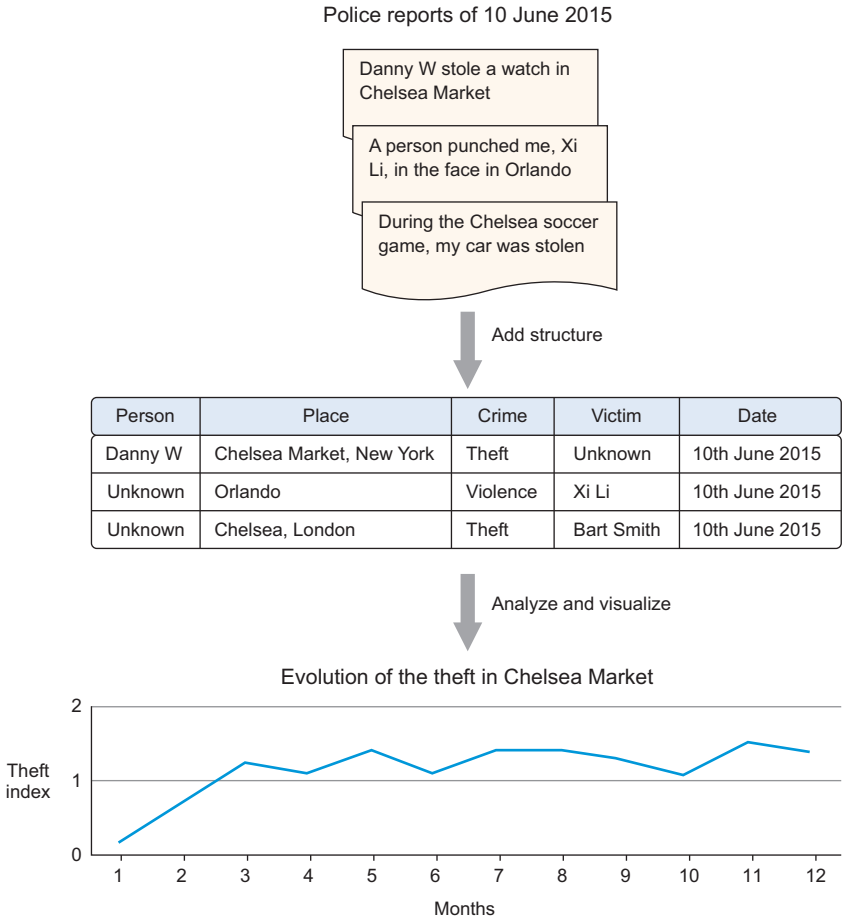

Figure 8.1 In text analytics, (usually) the first challenge is to structure the input text; then it can be thoroughly analyzed.

While language isn't limited to the natural language, the focus of this chapter will be on *Natural Language Processing (NLP)*. Examples of non-natural languages would be machine logs, mathematics, and Morse code. Technically even Esperanto, Klingon, and Dragon language aren't in the field of natural languages because they were invented deliberately instead of evolving over time; they didn't come "natural" to us. These last languages are nevertheless fit for natural communication (speech, writing); they have a grammar and a vocabulary as all natural languages do, and the same text mining techniques could apply to them.

8.1 Text mining in the real world

In your day-to-day life you've already come across text mining and natural language applications. Autocomplete and spelling correctors are constantly analyzing the text you type before sending an email or text message. When Facebook autocompletes your status with the name of a friend, it does this with the help of a technique called *named entity recognition*, although this would be only one component of their repertoire. The goal isn't only to detect that you're typing a noun, but also to guess you're referring to a person and recognize who it might be. Another example of named entity recognition is shown in figure 8.2. Google knows Chelsea is a football club but responds differently when asked for a person.

Google uses many types of text mining when presenting you with the results of a query. What pops up in your own mind when someone says "Chelsea"? Chelsea could be many things: a person; a soccer club; a neighborhood in Manhattan, New York or London; a food market; a flower show; and so on. Google knows this and returns different answers to the question "Who is Chelsea?" versus "What is Chelsea?" To provide the most relevant answer, Google must do (among other things) all of the following:

- Preprocess all the documents it collects for named entities
- Perform language identification
- Detect what type of entity you're referring to
- Match a query to a result
- Detect the type of content to return (PDF, adult-sensitive)

This example shows that text mining isn't only about the direct meaning of text itself but also involves meta-attributes such as language and document type.

Google uses text mining for much more than answering queries. Next to shielding its Gmail users from spam, it also divides the emails into different categories such as social, updates, and forums, as shown in figure 8.3.

It's possible to go much further than answering simple questions when you combine text with other logic and mathematics.

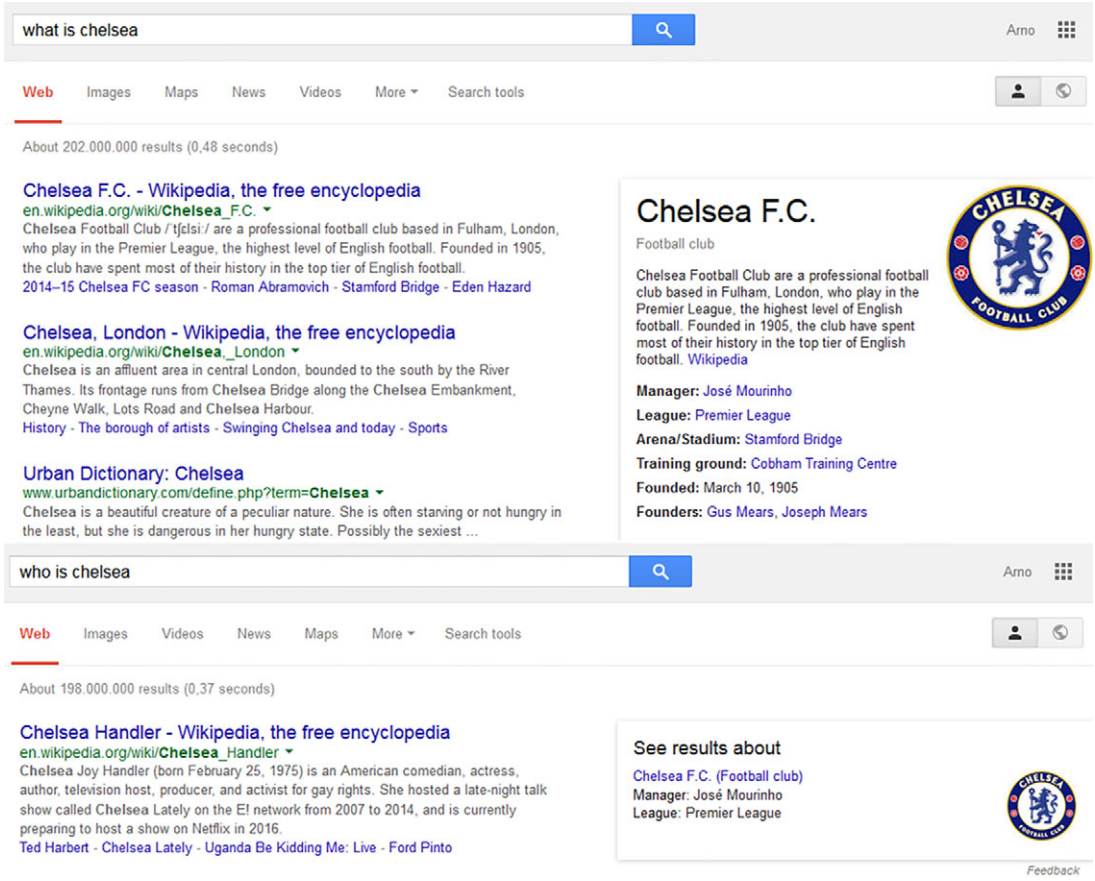

Figure 8.2 The different answers to the queries "Who is Chelsea?" and "What is Chelsea?" imply that Google uses text mining techniques to answer these queries.

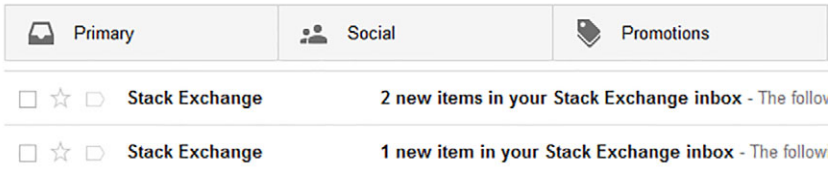

Figure 8.3 Emails can be automatically divided by category based on content and origin.

This allows for the creation of *automatic reasoning engines* driven by natural language queries. Figure 8.4 shows how "Wolfram Alpha," a computational knowledge engine, uses text mining and automatic reasoning to answer the question "Is the USA population bigger than China?"

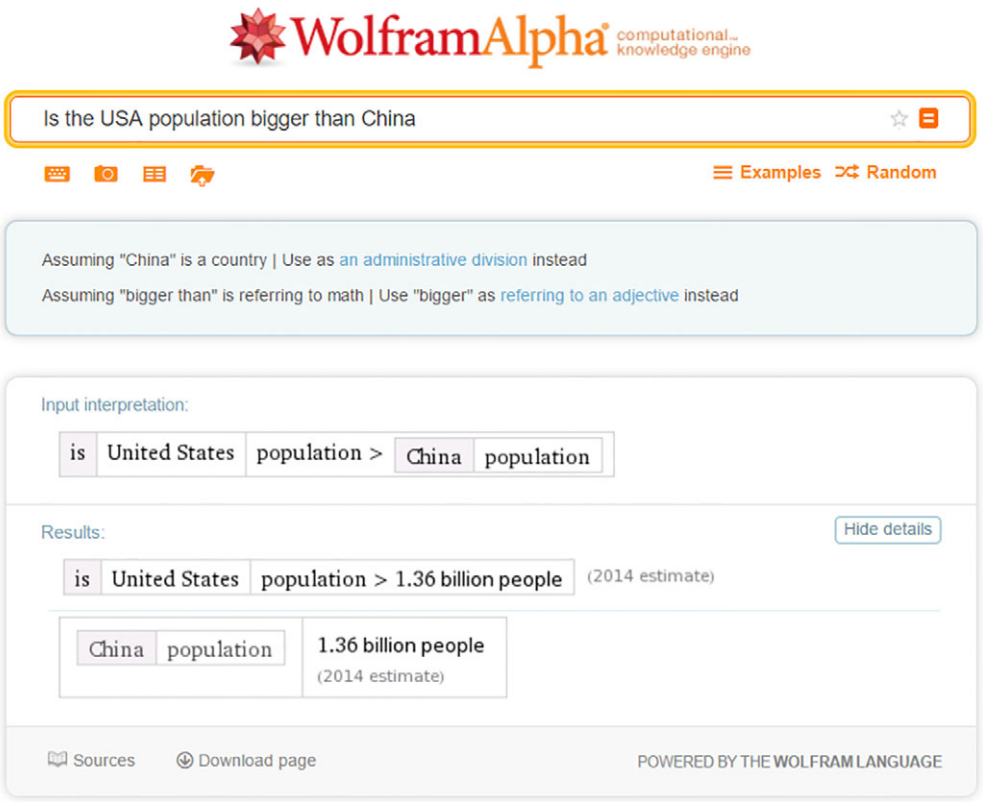

Figure 8.4 The Wolfram Alpha engine uses text mining and logical reasoning to answer a question.

If this isn't impressive enough, the IBM Watson astonished many in 2011 when the machine was set up against two human players in a game of *Jeopardy*. *Jeopardy* is an American quiz show where people receive the answer to a question and points are scored for guessing the correct question for that answer. See figure 8.5.

It's safe to say this round goes to artificial intelligence. IBM Watson is a cognitive engine that can interpret natural language and answer questions based on an extensive knowledge base.

Figure 8.5 IBM Watson wins *Jeopardy* against human players.

Text mining has many applications, including, but not limited to, the following:

- Entity identification
- Plagiarism detection
- Topic identification
- Text clustering
- Translation
- Automatic text summarization
- Fraud detection
- Spam filtering
- Sentiment analysis

Text mining is useful, but is it difficult? Sorry to disappoint: Yes, it is.

When looking at the examples of Wolfram Alpha and IBM Watson, you might have gotten the impression that text mining is easy. Sadly, no. In reality text mining is a complicated task and even many seemingly simple things can't be done satisfactorily. For instance, take the task of guessing the correct address. Figure 8.6 shows how difficult it is to return the exact result with certitude and how Google Maps prompts you for more information when looking for "Springfield." In this case a human wouldn't have done any better without additional context, but this ambiguity is one of the many problems you face in a text mining application.

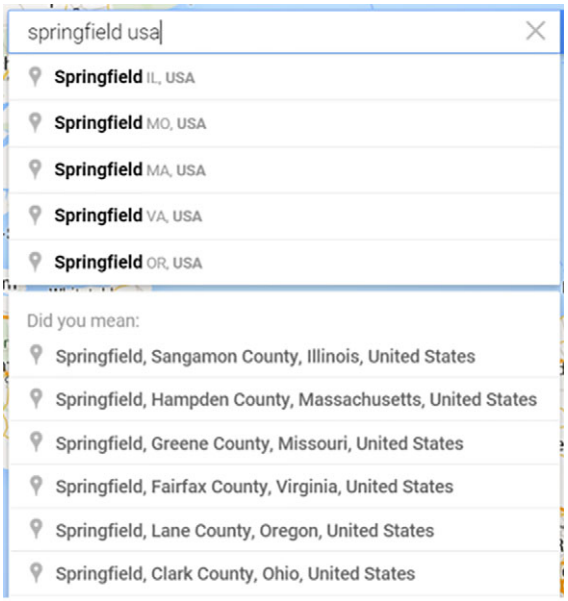

Figure 8.6 Google Maps asks you for more context due to the ambiguity of the query "Springfield."

Another problem is *spelling mistakes* and *different (correct) spelling* forms of a word. Take the following three references to New York: "NY," "Neww York," and "New York." For a human, it's easy to see they all refer to the city of New York. Because of the way our brain interprets text, understanding text with spelling mistakes comes naturally to us; people may not even notice them. But for a computer these are unrelated strings unless we use algorithms to tell it that they're referring to the same entity. Related problems are synonyms and the use of pronouns. Try assigning the right person to the pronoun "she" in the next sentences: "John gave flowers to Marleen's parents when he met her parents for the first time. She was so happy with this gesture." Easy enough, right? Not for a computer.

We can solve many similar problems with ease, but they often prove hard for a machine. We can train algorithms that work well on a specific problem in a well-defined scope, but more general algorithms that work in all cases are another beast altogether. For instance, we can teach a computer to recognize and retrieve US account numbers from text, but this doesn't generalize well to account numbers from other countries.

Language algorithms are also sensitive to the context the language is used in, even if the language itself remains the same. English models won't work for Arabic and vice versa, but even if we keep to English—an algorithm trained for Twitter data isn't likely to perform well on legal texts. Let's keep this in mind when we move on to the chapter case study: there's no perfect, one-size-fits-all solution in text mining.

8.2 Text mining techniques

During our upcoming case study we'll tackle the problem of *text classification*: automatically classifying uncategorized texts into specific categories. To get from raw textual data to our final destination we'll need a few data mining techniques that require background information for us to use them effectively. The first important concept in text mining is the "bag of words."

8.2.1 Bag of words

To build our classification model we'll go with the bag of words approach. *Bag of words* is the simplest way of structuring textual data: every document is turned into a word vector. If a certain word is present in the vector it's labeled "True"; the others are labeled "False". Figure 8.7 shows a simplified example of this, in case there are only two documents: one about the television show *Game of Thrones* and one about data science. The two word vectors together form the *document-term matrix*. The document-term matrix holds a column for every term and a row for every document. The values are yours to decide upon. In this chapter we'll use binary: term is present? True or False.

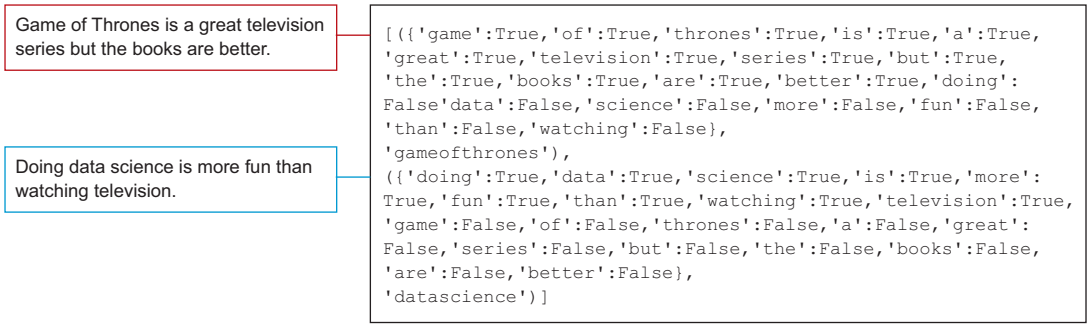

Figure 8.7 A text is transformed into a bag of words by labeling each word (term) with "True" if it is present in the document and "False" if not.

The example from figure 8.7 does give you an idea of the structured data we'll need to start text analysis, but it's severely simplified: not a single word was filtered out and no stemming (we'll go into this later) was applied. A big corpus can have thousands of unique words. If all have to be labeled like this without any filtering, it's easy to see we might end up with a large volume of data. *Binary coded bag of words* as shown in figure 8.7 is but one way to structure the data; other techniques exist.

Term Frequency—Inverse Document Frequency (TF-IDF)

A well-known formula to fill up the document-term matrix is *TF-IDF* or Term Frequency multiplied by Inverse Document Frequency. *Binary bag of words* assigns True or False (term is there or not), while *simple frequencies* count the number of times the term occurred. TF-IDF is a bit more complicated and takes into account how many times a term occurred in the document (TF). TF can be a simple term count, a binary count (True or False), or a logarithmically scaled term count. It depends on what works best for you. In case TF is a term frequency, the formula of TF is the following:

$$TF = f_{t,d}$$

TF is the frequency (f) of the term (t) in the document (d).

But TF-IDF also takes into account all the other documents because of the Inverse Document Frequency. IDF gives an idea of how common the word is in the entire corpus: the higher the document frequency the more common, and more common words are less informative. For example the words "a" or "the" aren't likely to provide specific information on a text. The formula of IDF with logarithmic scaling is the most commonly used form of IDF:

$$IDF = \log(N/|\{d \in D{:}t \in d\}|)$$

with N being the total number of documents in the corpus, and the $|\{d \in D{:}t \in d\}|$ being the number of documents (d) in which the term (t) appears.

The TF-IDF score says this about a term: how important is this word to distinguish this document from the others in the corpus? The formula of TF-IDF is thus

$$\frac{1F}{IDF} = f_{t,d}/\log(N/|\{d \in D{:}t \in d\}|)$$

We won't use TF-IDF, but when setting your next steps in text mining, this should be one of the first things you'll encounter. TF-IDF is also what was used by Elasticsearch behind the scenes in chapter 6. It's a good way to go if you want to use TF-IDF for text analytics; leave the text mining to specialized software such as SOLR or Elasticsearch and take the document/term matrix for text analytics from there.

Before getting to the actual bag of words, many other data manipulation steps take place:

- *Tokenization*—The text is cut into pieces called "tokens" or "terms." These tokens are the most basic unit of information you'll use for your model. The terms are often words but this isn't a necessity. Entire sentences can be used for analysis. We'll use *unigrams*: terms consisting of one word. Often, however, it's useful to include *bigrams* (two words per token) or *trigrams* (three words per token) to capture extra meaning and increase the performance of your models.

This does come at a cost, though, because you're building bigger term-vectors by including bigrams and/or trigrams in the equation.

- *Stop word filtering*—Every language comes with words that have little value in text analytics because they're used so often. NLTK comes with a short list of English stop words we can filter. If the text is tokenized into words, it often makes sense to rid the word vector of these low-information stop words.
- *Lowercasing*—Words with capital letters appear at the beginning of a sentence, others because they're proper nouns or adjectives. We gain no added value making that distinction in our term matrix, so all terms will be set to lowercase.

Another data preparation technique is *stemming*. This one requires more elaboration.

8.2.2 Stemming and lemmatization

Stemming is the process of bringing words back to their root form; this way you end up with less variance in the data. This makes sense if words have similar meanings but are written differently because, for example, one is in its plural form. Stemming attempts to unify by cutting off parts of the word. For example "planes" and "plane" both become "plane."

Another technique, called *lemmatization*, has this same goal but does so in a more grammatically sensitive way. For example, while both stemming and lemmatization would reduce "cars" to "car," lemmatization can also bring back conjugated verbs to their unconjugated forms such as "are" to "be." Which one you use depends on your case, and lemmatization profits heavily from POS Tagging (Part of Speech Tagging). *POS Tagging* is the process of attributing a grammatical label to every part of a sentence. You probably did this manually in school as a language exercise. Take the sentence "*Game of Thrones* is a television series." If we apply POS Tagging on it we get

({"game":"NN"},{"of":"IN},{"thrones":"NNS},{"is":"VBZ},{"a":"DT},{"television":"NN},
{"series":"NN})

NN is a noun, IN is a preposition, NNS is a noun in its plural form, VBZ is a third-person singular verb, and DT is a determiner. Table 8.1 has the full list.

Table 8.1 A list of all POS tags

Tag	Meaning	Tag	Meaning
CC	Coordinating conjunction	CD	Cardinal number
DT	Determiner	EX	Existential
FW	Foreign word	IN	Preposition or subordinating conjunction
JJ	Adjective	JJR	Adjective, comparative
JJS	Adjective, superlative	LS	List item marker
MD	Modal	NN	Noun, singular or mass

Table 8.1 A list of all POS tags *(continued)*

Tag	Meaning	Tag	Meaning
NNS	Noun, plural	NNP	Proper noun, singular
NNPS	Proper noun, plural	PDT	Predeterminer
POS	Possessive ending	PRP	Personal pronoun
PRP$	Possessive pronoun	RB	Adverb
RBR	Adverb, comparative	RBS	Adverb, superlative
RP	Particle	SYM	Symbol
UH	Interjection	VB	Verb, base form
VBD	Verb, past tense	VBG	Verb, gerund or present participle
VBN	Verb, past participle	VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present	WDT	Wh-determiner
WP	Wh-pronoun	WP$	Possessive wh-pronoun
WRB	Wh-adverb		

POS Tagging is a use case of sentence-tokenization rather than word-tokenization. After the POS Tagging is complete you can still proceed to word tokenization, but a POS Tagger requires whole sentences. Combining POS Tagging and lemmatization is likely to give cleaner data than using only a stemmer. For the sake of simplicity we'll stick to stemming in the case study, but consider this an opportunity to elaborate on the exercise.

We now know the most important things we'll use to do the data cleansing and manipulation (text mining). For our text analytics, let's add the decision tree classifier to our repertoire.

8.2.3 Decision tree classifier

The data analysis part of our case study will be kept simple as well. We'll test a Naïve Bayes classifier and a decision tree classifier. As seen in chapter 3 the Naïve Bayes classifier is called that because it considers each input variable to be independent of all the others, which is naïve, especially in text mining. Take the simple examples of "data science," "data analysis," or "game of thrones." If we cut our data in unigrams we get the following separate variables (if we ignore stemming and such): "data," "science," "analysis," "game," "of," and "thrones." Obviously links will be lost. This can, in turn, be overcome by creating bigrams (data science, data analysis) and trigrams (game of thrones).

The decision tree classifier, however, doesn't consider the variables to be independent of one another and actively creates *interaction variables* and *buckets*. An *interaction*

variable is a variable that combines other variables. For instance "data" and "science" might be good predictors in their own right but probably the two of them co-occurring in the same text might have its own value. A bucket is somewhat the opposite. Instead of combining two variables, a variable is split into multiple new ones. This makes sense for numerical variables. Figure 8.8 shows what a decision tree might look like and where you can find interaction and bucketing.

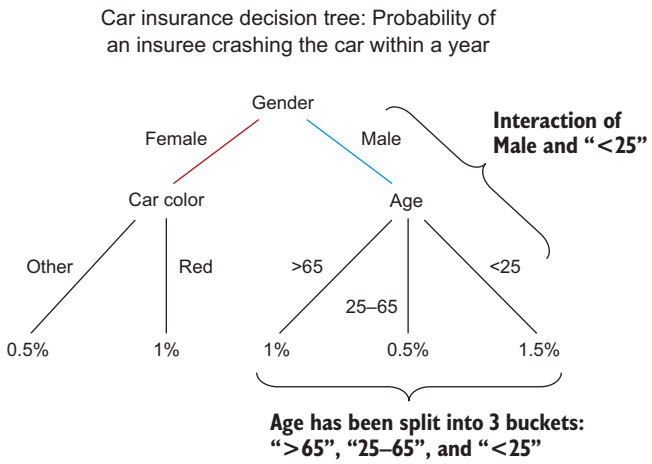

Figure 8.8 Fictitious decision tree model. A decision tree automatically creates buckets and supposes interactions between input variables.

Whereas Naïve Bayes supposes independence of all the input variables, a decision tree is built upon the assumption of interdependence. But how does it build this structure? A decision tree has a few possible criteria it can use to split into branches and decide which variables are more important (are closer to the root of the tree) than others. The one we'll use in the NLTK decision tree classifier is "information gain." To understand information gain, we first need to look at entropy. *Entropy* is a measure of unpredictability or chaos. A simple example would be the gender of a baby. When a woman is pregnant, the gender of the fetus can be male or female, but we don't know which one it is. If you were to guess, you have a 50% chance to guess correctly (give or take, because gender distribution isn't 100% uniform). However, during the pregnancy you have the opportunity to do an ultrasound to determine the gender of the fetus. An ultrasound is never 100% conclusive, but the farther along in fetal development, the more accurate it becomes. This accuracy gain, or *information gain*, is there because uncertainty or entropy drops. Let's say an ultrasound at 12 weeks pregnancy has a 90% accuracy in determining the gender of the baby. A 10% uncertainty still exists, but the ultrasound did reduce the uncertainty

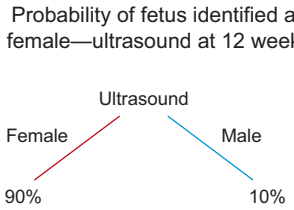

Figure 8.9 Decision tree with one variable: the doctor's conclusion from watching an ultrasound during a pregnancy. What is the probability of the fetus being female?

from 50% to 10%. That's a pretty good discriminator. A decision tree follows this same principle, as shown in figure 8.9.

If another gender test has more predictive power, it could become the root of the tree with the ultrasound test being in the branches, and this can go on until we run out of variables or observations. We can run out of observations, because at every branch split we also split the input data. This is a big weakness of the decision tree, because at the leaf level of the tree robustness breaks down if too few observations are left; the decision trees starts to overfit the data. *Overfitting* allows the model to mistake randomness for real correlations. To counteract this, a decision tree is *pruned*: its meaningless branches are left out of the final model.

Now that we've looked at the most important new techniques, let's dive into the case study.

8.3 Case study: Classifying Reddit posts

While text mining has many applications, in this chapter's case study we focus on *document classification*. As pointed out earlier in this chapter, this is exactly what Google does when it arranges your emails in categories or attempts to distinguish spam from regular emails. It's also extensively used by contact centers that process incoming customer questions or complaints: written complaints first pass through a topic detection filter so they can be assigned to the correct people for handling. Document classification is also one of the mandatory features of social media monitoring systems. The monitored tweets, forum or Facebook posts, newspaper articles, and many other internet resources are assigned topic labels. This way they can be reused in reports. *Sentiment analysis* is a specific type of text classification: is the author of a post negative, positive, or neutral on something? That "something" can be recognized with entity recognition.

In this case study we'll draw on posts from Reddit, a website also known as the self-proclaimed "front page of the internet," and attempt to train a model capable of distinguishing whether someone is talking about "data science" or "game of thrones."

The end result can be a presentation of our model or a full-blown interactive application. In chapter 9 we'll focus on application building for the end user, so for now we'll stick to presenting our classification model.

To achieve our goal we'll need all the help and tools we can get, and it happens Python is once again ready to provide them.

8.3.1 Meet the Natural Language Toolkit

Python might not be the most execution efficient language on earth, but it has a mature package for text mining and language processing: the *Natural Language Toolkit (NLTK)*. NLTK is a collection of algorithms, functions, and annotated works that will guide you in taking your first steps in text mining and natural language processing. NLTK is also excellently documented on nltk.org. NLTK is, however, not often used for production-grade work, like other libraries such as scikit-learn.

Installing NLTK and its corpora

Install NLTK with your favorite package installer. In case you're using Anaconda, it comes installed with the default Anaconda setup. Otherwise you can go for "pip" or "easy_install". When this is done you still need to install the models and corpora included to have it be fully functional. For this, run the following Python code:

- import nltk
- nltk.download()

Depending on your installation this will give you a pop-up or more command-line options.

Figure 8.10 shows the pop-up box you get when issuing the nltk.download() command.

You can download all the corpora if you like, but for this chapter we'll only make use of "punkt" and "stopwords". This download will be explicitly mentioned in the code that comes with this book.

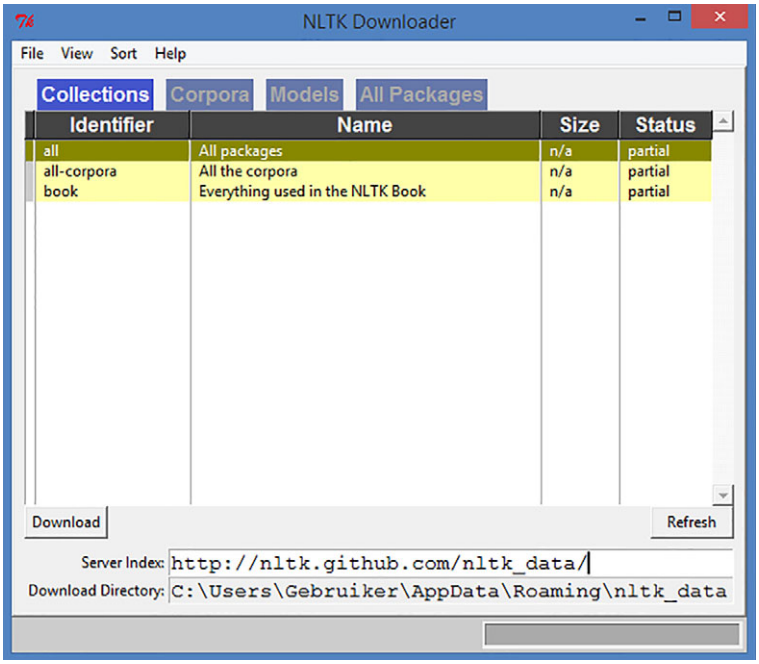

Figure 8.10 Choose All Packages to fully complete the NLTK installation.

Two IPython notebook files are available for this chapter:

- *Data collection*—Will contain the data collection part of this chapter's case study.
- *Data preparation and analysis*—The stored data is put through data preparation and then subjected to analysis.

All code in the upcoming case study can be found in these two files in the same sequence and can also be run as such. In addition, two interactive graphs are available for download:

- *forceGraph.html*—Represents the top 20 features of our Naïve Bayes model
- *Sunburst.html*—Represents the top four branches of our decision tree model

To open these two HTML pages, an HTTP server is necessary, which you can get using Python and a command window:

- Open a command window (Linux, Windows, whatever you fancy).
- Move to the folder containing the HTML files and their JSON data files: decisionTreeData.json for the sunburst diagram and NaiveBayesData.json for the force graph. It's important the HTML files remain in the same location as their data files or you'll have to change the JavaScript in the HTML file.
- Create a Python HTTP server with the following command: `python -m SimpleHTTPServer 8000`
- Open a browser and go to localhost:8000; here you can select the HTML files, as shown in figure 8.11.

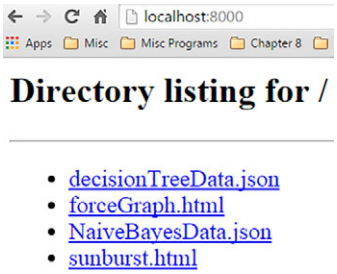

Figure 8.11 Python HTTP server serving this chapter's output

The Python packages we'll use in this chapter:

- *NLTK*—For text mining
- *PRAW*—Allows downloading posts from Reddit
- *SQLite3*—Enables us to store data in the SQLite format
- *Matplotlib*—A plotting library for visualizing data

Make sure to install all the necessary libraries and corpora before moving on. Before we dive into the action, however, let's look at the steps we'll take to get to our goal of creating a topic classification model.

8.3.2 Data science process overview and step 1: The research goal

To solve this text mining exercise, we'll once again make use of the data science process. Figure 8.12 shows the data science process applied to our Reddit classification case.

Not all the elements depicted in figure 8.12 might make sense at this point, and the rest of the chapter is dedicated to working this out in practice as we work toward our research goal: creating a classification model capable of distinguishing posts about "data science" from posts about "Game of Thrones." Without further ado, let's go get our data.

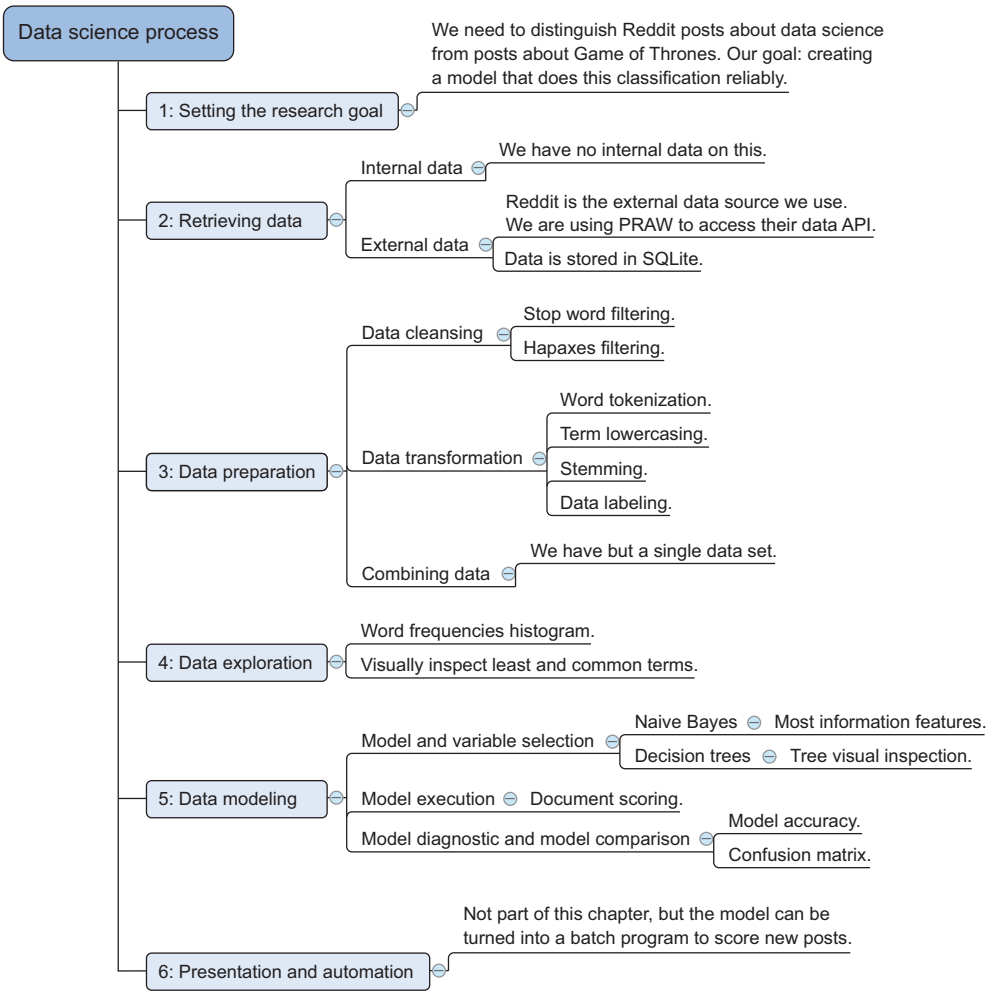

Figure 8.12 Data science process overview applied to Reddit topic classification case study

8.3.3 Step 2: Data retrieval

We'll use Reddit data for this case, and for those unfamiliar with Reddit, take the time to familiarize yourself with its concepts at www.reddit.com.

Reddit calls itself "the front page of the internet" because users can post things they find interesting and/or found somewhere on the internet, and only those things deemed interesting by many people are featured as "popular" on its homepage. You could say Reddit gives an overview of the trending things on the internet. Any user can post within a predefined category called a "subreddit." When a post is made, other users get to comment on it and can up-vote it if they like the content or down-vote it if they dislike it. Because a post is always part of a subreddit, we have this meta-data at our disposal when we hook up to the Reddit API to get our data. We're effectively fetching labeled data because we'll assume that a post in the subreddit "gameofthrones" has something to do with "gameofthrones."

To get to our data we make use of the official Reddit Python API library called PRAW. Once we get the data we need, we'll store it in a lightweight database-like file called SQLite. SQLite is ideal for storing small amounts of data because it doesn't require any setup to use and will respond to SQL queries like any regular relational database does. Any other data storage medium will do; if you prefer Oracle or Post-gres databases, Python has an excellent library to interact with these without the need to write SQL. SQLAlchemy will work for SQLite files as well. Figure 8.13 shows the data retrieval step within the data science process.

Figure 8.13 The data science process data retrieval step for a Reddit topic classification case

Open your favorite Python interpreter; it's time for action, as shown in listing 8.1. First we need to collect our data from the Reddit website. If you haven't already, use `pip install praw` or `conda install praw` (Anaconda) before running the following script.

NOTE The code for step 2 can also be found in the IPython file "Chapter 8 data collection." It's available in this book's download section.

Listing 8.1 Setting up SQLLite database and Reddit API client

```
import praw
import sqlite3
```

Import PRAW and SQLite3 libraries.

```
conn = sqlite3.connect('reddit.db')
c = conn.cursor()
```

Set up connection to SQLite database.

```
c.execute('''DROP TABLE IF EXISTS topics''')
c.execute('''DROP TABLE IF EXISTS comments''')
c.execute('''CREATE TABLE topics
             (topicTitle text, topicText text, topicID text,
topicCategory text)''')
c.execute('''CREATE TABLE comments
             (commentText text, commentID text ,
topicTitle text, topicText text, topicID text ,
 topicCategory text)''')
```

Execute SQL statements to create topics and comments table.

```
user_agent = "Introducing Data Science Book"
r = praw.Reddit(user_agent=user_agent)
```

Create PRAW user agent so we can use Reddit API.

```
subreddits = ['datascience','gameofthrones']
```

Our list of subreddits we'll draw into our SQLite database.

```
limit = 1000
```

Maximum number of posts we'll fetch from Reddit per category. Maximum Reddit allows at any single time is also 1,000.

Let's first import the necessary libraries.

Now that we have access to the SQLite3 and PRAW capabilities, we need to prepare our little local database for the data it's about to receive. By defining a connection to a SQLite file we automatically create it if it doesn't already exist. We then define a data cursor that's capable of executing any SQL statement, so we use it to predefine the structure of our database. The database will contain two tables: the topics table contains Reddit topics, which is similar to someone starting a new post on a forum, and the second table contains the comments and is linked to the topic table via the "topicID" column. The two tables have a one (topic table) to many (comment table) relationship. For the case study, we'll limit ourselves to using the topics table, but the data collection will incorporate both because this allows you to experiment with this extra data if you feel like it. To hone your text-mining skills you could perform sentiment analysis on the topic comments and find out what topics receive negative or positive comments. You could then correlate this to the model features we'll produce by the end of this chapter.

We need to create a PRAW client to get access to the data. Every subreddit can be identified by its name, and we're interested in "datascience" and "gameofthrones." The limit represents the maximum number of topics (posts, not comments) we'll draw in from Reddit. A thousand is also the maximum number the API allows us to fetch at any given request, though we could request more later on when people have

posted new things. In fact we can run the API request periodically and gather data over time. While at any given time you're limited to a thousand posts, nothing stops you from growing your own database over the course of months. It's worth noting the following script might take about an hour to complete. If you don't feel like waiting, feel free to proceed and use the downloadable SQLite file. Also, if you run it now you are not likely to get the exact same output as when it was first run to create the output shown in this chapter.

Let's look at our data retrieval function, as shown in the following listing.

Listing 8.2 Reddit data retrieval and storage in SQLite

```
def prawGetData(limit,subredditName):
    topics = r.get_subreddit(subredditName).get_hot(limit=limit)
    commentInsert = []
    topicInsert = []
    topicNBR = 1
    for topic in topics:
        if (float(topicNBR)/limit)*100 in xrange(1,100):
            print '*********** TOPIC:' + str(topic.id)
+ ' *********COMPLETE: ' + str((float(topicNBR)/limit)*100)
+ ' % ****'
        topicNBR += 1
        try:
            topicInsert.append((topic.title,topic.selftext,topic.id,
            subredditName))
        except:
            pass
        try:
            for comment in topic.comments:
                commentInsert.append((comment.body,comment.id,
topic.title,topic.selftext,topic.id,subredditName))
        except:
            pass
    print  '********************************'
    print  'INSERTING DATA INTO SQLITE'
    c.executemany('INSERT INTO topics VALUES (?,?,?,?)', topicInsert)
    print  'INSERTED TOPICS'
    c.executemany('INSERT INTO comments VALUES (?,?,?,?,?,?)', commentInsert)
    print  'INSERTED COMMENTS'
    conn.commit()

for subject in subreddits:
    prawGetData(limit=limit,subredditName=subject)
```

Specific fields of the topic are appended to the list. We only use the title and text throughout the exercise but the topic ID would be useful for building your own (bigger) database of topics.

From subreddits, get hottest 1,000 (in our case) topics.

This part is an informative print and not necessary for code to work. It only informs you about the download progress.

Append comments to a list. These are not used in the exercise but now you have them for experimentation.

Insert all topics into SQLite database.

Insert all comments into SQLite database.

Commit changes (data insertions) to database. Without the commit, no data will be inserted.

The function is executed for all subreddits we specified earlier.

The `prawGetData()` function retrieves the "hottest" topics in its subreddit, appends this to an array, and then gets all its related comments. This goes on until a thousand topics are reached or no more topics exist to fetch and everything is stored in the SQLite database. The print statements are there to inform you on its progress toward gathering a thousand topics. All that's left for us to do is execute the function for each subreddit.

If you'd like this analysis to incorporate more than two subreddits, this is a matter of adding an extra category to the subreddits array.

With the data collected, we're ready to move on to data preparation.

8.3.4 Step 3: Data preparation

As always, data preparation is the most crucial step to get correct results. For text mining this is even truer since we don't even start off with structured data.

The upcoming code is available online as IPython file "Chapter 8 data preparation and analysis." Let's start by importing the required libraries and preparing the SQLite database, as shown in the following listing.

Listing 8.3 Text mining, libraries, corpora dependencies, and SQLite database connection

```
import sqlite3
import nltk
import matplotlib.pyplot as plt
from collections import OrderedDict
import random
```

Import all required libraries

```
nltk.download('punkt')
nltk.download('stopwords')
```

Download corpora we make use of

```
conn = sqlite3.connect('reddit.db')
c = conn.cursor()
```

Make a connection to SQLite database that contains our Reddit data

In case you haven't already downloaded the full NLTK corpus, we'll now download the part of it we'll use. Don't worry if you already downloaded it, the script will detect if your corpora is up to date.

Our data is still stored in the Reddit SQLite file so let's create a connection to it.

Even before exploring our data we know of at least two things we have to do to clean the data: stop word filtering and lowercasing.

A general word filter function will help us filter out the unclean parts. Let's create one in the following listing.

Listing 8.4 Word filtering and lowercasing functions

```
def wordFilter(excluded,wordrow):
    filtered = [word for word in wordrow if word not in excluded]
    return filtered
stopwords = nltk.corpus.stopwords.words('english')
def lowerCaseArray(wordrow):
    lowercased = [word.lower() for word in wordrow]
    return lowercased
```

wordFilter() function will remove a term from an array of terms

Stop word variable contains English stop words per default present in NLTK

lowerCaseArray() function transforms any term to its lowercased version

The English stop words will be the first to leave our data. The following code will provide us these stop words:

```
stopwords = nltk.corpus.stopwords.words('english')
print stopwords
```

Figure 8.14 shows the list of English stop words in NLTK.

```
stopwords = nltk.corpus.stopwords.words('english')
print stopwords
```

```
[u'i', u'me', u'my', u'myself', u'we', u'our', u'ours', u'ourselves', u'you',
u'your', u'yours', u'yourself', u'yourselves', u'he', u'him', u'his', u'himsel
f', u'she', u'her', u'hers', u'herself', u'it', u'its', u'itself', u'they', u'th
em', u'their', u'theirs', u'themselves', u'what', u'which', u'who', u'whom', u't
his', u'that', u'these', u'those', u'am', u'is', u'are', u'was', u'were', u'be',
u'been', u'being', u'have', u'has', u'had', u'having', u'do', u'does', u'did',
u'doing', u'a', u'an', u'the', u'and', u'but', u'if', u'or', u'because', u'as',
u'until', u'while', u'of', u'at', u'by', u'for', u'with', u'about', u'against',
u'between', u'into', u'through', u'during', u'before', u'after', u'above', u'bel
ow', u'to', u'from', u'up', u'down', u'in', u'out', u'on', u'off', u'over', u'un
der', u'again', u'further', u'then', u'once', u'here', u'there', u'when', u'wher
e', u'why', u'how', u'all', u'any', u'both', u'each', u'few', u'more', u'most',
u'other', u'some', u'such', u'no', u'nor', u'not', u'only', u'own', u'same', u's
o', u'than', u'too', u'very', u's', u't', u'can', u'will', u'just', u'don', u'sh
ould', u'now']
```

Figure 8.14 English stop words list in NLTK

With all the necessary components in place, let's have a look at our first data processing function in the following listing.

Listing 8.5 First data preparation function and execution

```
def data_processing(sql):
    c.execute(sql)
    data = {'wordMatrix':[],'all_words':[]}
    row = c.fetchone()
    while row is not None:
        wordrow = nltk.tokenize.word_tokenize(row[0]+" "+row[1])
        wordrow_lowercased = lowerCaseArray(wordrow)
        wordrow_nostopwords = wordFilter(stopwords,wordrow_lowercased)
        data['all_words'].extend(wordrow_nostopwords)
        data['wordMatrix'].append(wordrow_nostopwords)
        row = c.fetchone()
    return data

subreddits = ['datascience','gameofthrones']
data = {}
for subject in subreddits:
    data[subject] = data_processing(sql='''SELECT
     topicTitle,topicText,topicCategory FROM topics
WHERE topicCategory = '''+"'"+subject+"'")
```

We'll use data['all_words'] for data exploration.

Create pointer to AWLite data.

Fetch data row by row.

row[0] is title, row[1] is topic text; we turn them into a single text blob.

Get new document from SQLite database.

Our subreddits as defined earlier.

Call data processing function for every subreddit.

data['wordMatrix'] is a matrix comprised of word vectors; 1 vector per document.

Our `data_processing()` function takes in a SQL statement and returns the document-term matrix. It does this by looping through the data one entry (Reddit topic) at a time and combines the topic title and topic body text into a single word vector with the use of word tokenization. A *tokenizer* is a text handling script that cuts the text into pieces. You have many different ways to tokenize a text: you can divide it into sentences or words, you can split by space and punctuations, or you can take other characters into account, and so on. Here we opted for the standard NLTK word tokenizer. This word tokenizer is simple; all it does is split the text into terms if there's a space between the words. We then lowercase the vector and filter out the stop words. Note how the order is important here; a stop word in the beginning of a sentence wouldn't be filtered if we first filter the stop words before lowercasing. For instance in "I like Game of Thrones," the "I" would not be lowercased and thus would not be filtered out. We then create a word matrix (term-document matrix) and a list containing all the words. Notice how we extend the list without filtering for doubles; this way we can create a histogram on word occurrences during data exploration. Let's execute the function for our two topic categories.

Figure 8.15 shows the first word vector of the "datascience" category.

```
print data['datascience']['wordMatrix'][0]
```

```
print data['datascience']['wordMatrix'][0]
```

```
[u'data', u'science', u'freelancing', u"'m", u'currently', u'master
s', u'program', u'studying', u'business', u'analytics', u"'m", u'try
ing', u'get', u'data', u'freelancing', u'.', u"'m", u'still', u'lear
ning', u'skill', u'set', u'typically', u'see', u'right', u"'m", u'fa
irly', u'proficient', u'sql', u'know', u'bit', u'r.', u'freelancer
s', u'find', u'jobs', u'?']
```

Figure 8.15 The first word vector of the "datascience" category after first data processing attempt

This sure looks polluted: punctuations are kept as separate terms and several words haven't even been split. Further data exploration should clarify a few things for us.

8.3.5 Step 4: Data exploration

We now have all our terms separated, but the sheer size of the data hinders us from getting a good grip on whether it's clean enough for actual use. By looking at a single vector, we already spot a few problems though: several words haven't been split correctly and the vector contains many single-character terms. Single character terms might be good topic differentiators in certain cases. For example, an economic text will contain more $, £, and € signs than a medical text. But in most cases these one-character terms are useless. First, let's have a look at the frequency distribution of our terms.

```
wordfreqs_cat1 = nltk.FreqDist(data['datascience']['all_words'])
plt.hist(wordfreqs_cat1.values(), bins = range(10))
plt.show()
wordfreqs_cat2 = nltk.FreqDist(data['gameofthrones']['all_words'])
plt.hist(wordfreqs_cat2.values(), bins = range(20))
plt.show()
```

By drawing a histogram of the frequency distribution (figure 8.16) we quickly notice that the bulk of our terms only occur in a single document.

Single-occurrence terms such as these are called *hapaxes*, and model-wise they're useless because a single occurrence of a feature is never enough to build a reliable model. This is good news for us; cutting these hapaxes out will significantly shrink our data without harming our eventual model. Let's look at a few of these single-occurrence terms.

```
print wordfreqs_cat1.hapaxes()
print wordfreqs_cat2.hapaxes()
```

Terms we see in figure 8.17 make sense, and if we had more data they'd likely occur more often.

```
print wordfreqs_cat1.hapaxes()
print wordfreqs_cat2.hapaxes()
```

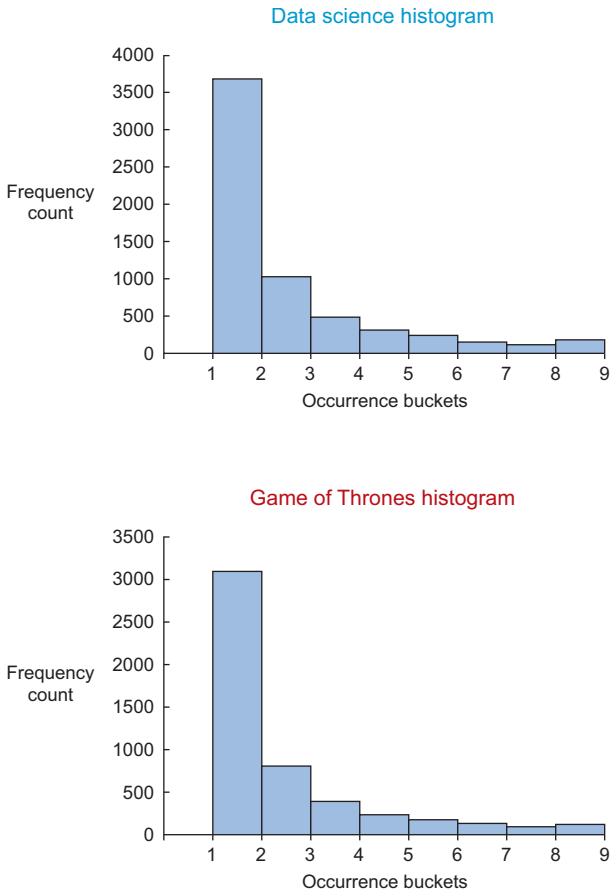

Figure 8.16 This histogram of term frequencies shows both the "data science" and "game of thrones" term matrices have more than 3,000 terms that occur once.

Least frequent terms within data science posts

```
print wordfreqs_cat1.hapaxes()
```

```
[u'post-grad', u'marching', u'cytoscape', u'wizardry', u"'pure", u'i
mmature', u'socrata', u'filenotfoundexception', u'side-by-side', u'b
ringing', u'non-experienced', u'zestimate', u'formatting*', u'sustai
```

Least frequent terms within Game of Thrones posts

```
print wordfreqs_cat2.hapaxes()
```

```
[u'hordes', u'woods', u'comically', u'pack', u'seventy-seven', u"'co
ntext", u'shaving', u'kennels', u'differently', u'screaming', u'her-
', u'complainers', u'sailed', u'contributed', u'payoff', u'hallucina
```

Figure 8.17 "Data science" and "game of thrones" single occurrence terms (hapaxes)

Many of these terms are incorrect spellings of otherwise useful ones, such as: Jaimie is Jaime (Lannister), Milisandre would be Melisandre, and so on. A decent *Game of Thrones*-specific thesaurus could help us find and replace these misspellings with a fuzzy search algorithm. This proves data cleaning in text mining can go on indefinitely if you so desire; keeping effort and payoff in balance is crucial here.

Let's now have a look at the most frequent words.

```
print wordfreqs_cat1.most_common(20)
print wordfreqs_cat2.most_common(20)
```

Figure 8.18 shows the output of asking for the top 20 most common words for each category.

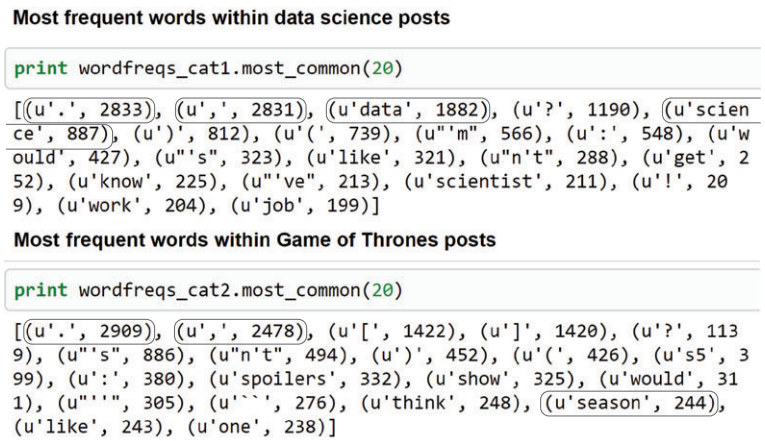

Figure 8.18 Top 20 most frequent words for the "data science" and "game of thrones" posts

Now this looks encouraging: several common words do seem specific to their topics. Words such as "data," "science," and "season" are likely to become good differentiators. Another important thing to notice is the abundance of the single character terms such as "." and ","; we'll get rid of these.

With this extra knowledge, let's revise our data preparation script.

8.3.6 Step 3 revisited: Data preparation adapted

This short data exploration has already drawn our attention to a few obvious tweaks we can make to improve our text. Another important one is stemming the terms.

The following listing shows a simple stemming algorithm called "snowball stemming." These snowball stemmers can be language-specific, so we'll use the English one; however, it does support many languages.

Listing 8.6 The Reddit data processing revised after data exploration

```
stemmer = nltk.SnowballStemmer("english")
def wordStemmer(wordrow):
    stemmed = [stemmer.stem(word) for word in wordrow]
    return stemmed

manual_stopwords = [',','.',')',',','(','m',"'m","n't",'e.g',"'ve",'s','#','/
     ','``',"'s","'",'!','r',']','=','[','s','&','%','*','...','1','2','3','
     4','5','6','7','8','9','10','--',"'",';','-',':']

def data_processing(sql,manual_stopwords):
    #create pointer to the sqlite data
    c.execute(sql)
    data = {'wordMatrix':[],'all_words':[]}
    interWordMatrix = []
    interWordList = []

    row = c.fetchone()
    while row is not None:
        tokenizer = nltk.tokenize.RegexpTokenizer(r'\w+|[^\w\s]+')

        wordrow = tokenizer.tokenize(row[0]+" "+row[1])
        wordrow_lowercased = lowerCaseArray(wordrow)
        wordrow_nostopwords = wordFilter(stopwords,wordrow_lowercased)

        wordrow_nostopwords =
           wordFilter(manual_stopwords,wordrow_nostopwords)
        wordrow_stemmed = wordStemmer(wordrow_nostopwords)

        interWordList.extend(wordrow_stemmed)
        interWordMatrix.append(wordrow_stemmed)

        row = c.fetchone()

    wordfreqs = nltk.FreqDist(interWordList) s
    hapaxes = wordfreqs.hapaxes()
    for wordvector in interWordMatrix:
        wordvector_nohapexes = wordFilter(hapaxes,wordvector)
        data['wordMatrix'].append(wordvector_nohapexes)
        data['all_words'].extend(wordvector_nohapexes)

    return data

for subject in subreddits:
    data[subject] = data_processing(sql='''SELECT
     topicTitle,topicText,topicCategory FROM topics
WHERE topicCategory = '''+"'"+subject+"'",
manual_stopwords=manual_stopwords)
```

Initializes stemmer from NLTK library.

Stop words array defines terms to remove/ignore.

Now we define our revised data preparation.

Fetch data (reddit posts) one by one from SQLite database.

row[0] and row[1] contain the title and text of the post, respectively. We combine them into a single text blob.

Remove manually added stop words from text blob.

Temporary word list used to remove hapaxes later on.

Temporary word matrix; will become final word matrix after hapaxes removal.

Get new topic.

Make frequency distribution of all terms.

Get list of hapaxes.

Loop through temporary word matrix.

Remove hapaxes in each word vector.

Append correct word vector to final word matrix.

Extend list of all terms with corrected word vector.

Run new data processing function for both subreddits.

Notice the changes since the last `data_processing()` function. Our tokenizer is now a regular expression tokenizer. Regular expressions are not part of this book and are often considered challenging to master, but all this simple one does is cut the text into words. For words, any alphanumeric combination is allowed (\w), so there are no more special characters or punctuations. We also applied the word stemmer and removed a list of extra stop words. And, all the hapaxes are removed at the end because everything needs to be stemmed first. Let's run our data preparation again.

If we did the same exploratory analysis as before, we'd see it makes more sense, and we have no more hapaxes.

```
print wordfreqs_cat1.hapaxes()
print wordfreqs_cat2.hapaxes()
```

Let's take the top 20 words of each category again (see figure 8.19).

Top 20 most common "Data Science" terms after more intense data cleansing

```
wordfreqs_cat1 = nltk.FreqDist(data['datascience']['all_words'])
print wordfreqs_cat1.most_common(20)
```

```
[(u'data', 1971), (u'scienc', 955), (u'would', 418), (u'work', 368), (u'use', 34
7), (u'program', 343), (u'learn', 342), (u'like', 341), (u'get', 325), (u'scient
ist', 310), (u'job', 268), (u'cours', 265), (u'look', 257), (u'know', 239), (u's
tatist', 228), (u'want', 225), (u've', 223), (u'python', 205), (u'year', 204),
(u'time', 196)]
```

Top 20 most common "Game of Thrones" terms after more intense data cleansing

```
wordfreqs_cat2 = nltk.FreqDist(data['gameofthrones']['all_words'])
print wordfreqs_cat2.most_common(20)
```

```
[(u's5', 426), (u'spoiler', 374), (u'show', 362), (u'episod', 300), (u'think', 2
89), (u'would', 287), (u'season', 286), (u'like', 282), (u'book', 271), (u'one',
249), (u'get', 236), (u'sansa', 232), (u'scene', 216), (u'cersei', 213), (u'kno
w', 192), (u'go', 188), (u'king', 183), (u'throne', 181), (u'see', 177), (u'char
act', 177)]
```

Figure 8.19 Top 20 most frequent words in "data science" and "game of thrones" Reddit posts after data preparation

We can see in figure 8.19 how the data quality has improved remarkably. Also, notice how certain words are shortened because of the stemming we applied. For instance, "science" and "sciences" have become "scienc;" "courses" and "course" have become "cours," and so on. The resulting terms are not actual words but still interpretable. If you insist on your terms remaining actual words, lemmatization would be the way to go.

With the data cleaning process "completed" (remark: a text mining cleansing exercise can almost never be fully completed), all that remains is a few data transformations to get the data in the bag of words format.

First, let's label all our data and also create a holdout sample of 100 observations per category, as shown in the following listing.

Listing 8.7 Final data transformation and data splitting before modeling

Holdout sample is comprised of unlabeled data from the two subreddits: 100 observations from each data set. The labels are kept in a separate data set.

Holdout sample will be used to determine the model's flaws by constructing a confusion matrix.

```
holdoutLength  = 100
```

```
labeled_data1 = [(word,'datascience') for word in
     data['datascience']['wordMatrix'][holdoutLength:]]
labeled_data2 = [(word,'gameofthrones') for word in
     data['gameofthrones']['wordMatrix'][holdoutLength:]]
labeled_data = []
labeled_data.extend(labeled_data1)
labeled_data.extend(labeled_data2)
```

We create a single data set with every word vector tagged as being either 'datascience' or 'gameofthrones.' We keep part of the data aside for holdout sample.

```
holdout_data = data['datascience']['wordMatrix'][:holdoutLength]
holdout_data.extend(data['gameofthrones']['wordMatrix'][:holdoutLength])
holdout_data_labels = ([('datascience')
for _ in xrange(holdoutLength)] + [('gameofthrones') for _ in
     xrange(holdoutLength)])
```

```
data['datascience']['all_words_dedup'] =
list(OrderedDict.fromkeys(
data['datascience']['all_words']))
data['gameofthrones']['all_words_dedup'] =
list(OrderedDict.fromkeys(
data['gameofthrones']['all_words']))
all_words = []
all_words.extend(data['datascience']['all_words_dedup'])
all_words.extend(data['gameofthrones']['all_words_dedup'])
all_words_dedup = list(OrderedDict.fromkeys(all_words))
```

A list of all unique terms is created to build the bag of words data we need for training or scoring a model.

Data for model training and testing is first shuffled.

```
prepared_data = [({word: (word in x[0]) for word
in all_words_dedup}, x[1]) for x in labeled_data]
prepared_holdout_data = [({word: (word in x[0])
for word in all_words_dedup})
for x in holdout_data]
```

Data is turned into a binary bag of words format.

```
random.shuffle(prepared_data)
train_size = int(len(prepared_data) * 0.75)
train = prepared_data[:train_size]
test = prepared_data[train_size:]
```

Size of training data will be 75% of total and remaining 25% will be used for testing model performance.

The holdout sample will be used for our final test of the model and the creation of a confusion matrix. A *confusion matrix* is a way of checking how well a model did on previously unseen data. The matrix shows how many observations were correctly and incorrectly classified.

Before creating or training and testing data we need to take one last step: pouring the data into a bag of words format where every term is given either a "True" or "False" label depending on its presence in that particular post. We also need to do this for the unlabeled holdout sample.

Our prepared data now contains every term for each vector, as shown in figure 8.20.

```
print prepared_data[0]
```

```
print prepared_data[0]
({u'sunspear': False, u'profici': False, u'pardon': False, u'selye
s': False, u'four': False, u'davo': False, u'sleev': False, u'slee
                                  ⋮
u'daeron': False, u'portion': False, u'emerg': False, u'fifti': Fals
e, u'decemb': False, u'defend': False, u'sincer': False}, 'datascien
ce')
```

Figure 8.20 A binary bag of words ready for modeling is very sparse data.

We created a big but sparse matrix, allowing us to apply techniques from chapter 5 if it was too big to handle on our machine. With such a small table, however, there's no need for that now and we can proceed to shuffle and split the data into a training and test set.

While the biggest part of your data should always go to the model training, an optimal split ratio exists. Here we opted for a 3-1 split, but feel free to play with this. The more observations you have, the more freedom you have here. If you have few observations you'll need to allocate relatively more to training the model. We're now ready to move on to the most rewarding part: data analysis.

8.3.7 Step 5: Data analysis

For our analysis we'll fit two classification algorithms to our data: Naïve Bayes and decision trees. Naïve Bayes was explained in chapter 3 and decision tree earlier in this chapter.

Let's first test the performance of our Naïve Bayes classifier. NLTK comes with a classifier, but feel free to use algorithms from other packages such as SciPy.

```
classifier  = nltk.NaiveBayesClassifier.train(train)
```

With the classifier trained we can use the test data to get a measure on overall accuracy.

```
nltk.classify.accuracy(classifier, test)
```

```
nltk.classify.accuracy(classifier, test)
```

```
0.9681528662420382
```

Figure 8.21 Classification accuracy is a measure representing what percentage of observations was correctly classified on the test data.

The accuracy on the test data is estimated to be greater than 90%, as seen in figure 8.21. *Classification accuracy* is the number of correctly classified observations as a percentage of the total number of observations. Be advised, though, that this can be different in your case if you used different data.

```
nltk.classify.accuracy(classifier, test)
```

That's a good number. We can now lean back and relax, right? No, not really. Let's test it again on the 200 observations holdout sample and this time create a confusion matrix.

```
classified_data = classifier.classify_many(prepared_holdout_data)
cm = nltk.ConfusionMatrix(holdout_data_labels, classified_data)
print cm
```

The confusion matrix in figure 8.22 shows us the 97% is probably over the top because we have 28 (23 + 5) misclassified cases. Again, this can be different with your data if you filled the SQLite file yourself.

```
                |     g |
                |     a |
                |  d  m |
                |  a  e |
                |  t  o |
                |  a  f |
                |  s  t |
                |  c  h |
                |  i  r |
                |  e  o |
                |  n  n |
                |  c  e |
                |  e  s |
----------------+-------+
  datascience |<77>23 |
gameofthrones |  5<95>|
----------------+-------+
(row = reference; col = test)
```

Figure 8.22 Naïve Bayes model confusion matrix shows 28 (23 + 5) observations out of 200 were misclassified

Twenty-eight misclassifications means we have an 86% accuracy on the holdout sample. This needs to be compared to randomly assigning a new post to either the "datascience" or "gameofthrones" group. If we'd randomly assigned them, we could expect an

accuracy of 50%, and our model seems to perform better than that. Let's look at what it uses to determine the categories by digging into the most informative model features.

```
print(classifier.show_most_informative_features(20))
```

Figure 8.23 shows the top 20 terms capable of distinguishing between the two categories.

```
Most Informative Features
                   data = True           datasc : gameof =    365.1 : 1.0
                  scene = True           gameof : datasc =     63.8 : 1.0
                 season = True           gameof : datasc =     62.4 : 1.0
                   king = True           gameof : datasc =     47.6 : 1.0
                     tv = True           gameof : datasc =     45.1 : 1.0
                   kill = True           gameof : datasc =     31.5 : 1.0
                 compani = True          datasc : gameof =     28.5 : 1.0
                analysi = True           datasc : gameof =     27.1 : 1.0
                process = True           datasc : gameof =     25.5 : 1.0
                  appli = True           datasc : gameof =     25.5 : 1.0
               research = True           datasc : gameof =     23.2 : 1.0
                episod = True           gameof : datasc =     22.2 : 1.0
                market = True           datasc : gameof =     21.7 : 1.0
                 watch = True           gameof : datasc =     21.6 : 1.0
                   man = True           gameof : datasc =     21.0 : 1.0
                 north = True           gameof : datasc =     20.8 : 1.0
                    hi = True           datasc : gameof =     20.4 : 1.0
                 level = True           datasc : gameof =     19.1 : 1.0
                 learn = True           datasc : gameof =     16.9 : 1.0
                   job = True           datasc : gameof =     16.6 : 1.0
```

Figure 8.23 The most important terms in the Naïve Bayes classification model

The term "data" is given heavy weight and seems to be the most important indicator of whether a topic belongs in the data science category. Terms such as "scene," "season," "king," "tv," and "kill" are good indications the topic is *Game of Thrones* rather than data science. All these things make perfect sense, so the model passed both the accuracy and the sanity check.

The Naïve Bayes does well, so let's have a look at the decision tree in the following listing.

Listing 8.8 Decision tree model training and evaluation

```
classifier2 = nltk.DecisionTreeClassifier.train(train)
nltk.classify.accuracy(classifier2, test)
classified_data2 = classifier2.classify_many(prepared_holdout_data)
cm = nltk.ConfusionMatrix(holdout_data_labels, classified_data2)
print cm
```

Create confusion matrix based on classification results and actual labels

Train decision tree classifier

Test classifier accuracy

Attempt to classify holdout data (scoring)

Show confusion matrix

```
nltk.classify.accuracy(classifier2, test)
```

```
0.9333333333333333
```

Figure 8.24 Decision tree model accuracy

As shown in figure 8.24, the promised accuracy is 93%.

We now know better than to rely solely on this single test, so once again we turn to a confusion matrix on a second set of data, as shown in figure 8.25.

Figure 8.25 shows a different story. On these 200 observations of the holdout sample the decision tree model tends to classify well when the post is about *Game of Thrones* but fails miserably when confronted with the data science posts. It seems the model has a preference for *Game of Thrones*, and can you blame it? Let's have a look at the actual model, even though in this case we'll use the Naïve Bayes as our final model.

```
print(classifier2.pseudocode(depth=4))
```

```
                |     g |
                |     a |
                |  d  m |
                |  a  e |
                |  t  o |
                |  a  f |
                |  s  t |
                |  c  h |
                |  i  r |
                |  e  o |
                |  n  n |
                |  c  e |
                |  e  s |
--------------+-------+
  datascience |<26>74 |
gameofthrones |  2<98>|
--------------+-------+
(row = reference; col = test)
```

Figure 8.25 Confusion matrix on decision tree model

The decision tree has, as the name suggests, a tree-like model, as shown in figure 8.26.

The Naïve Bayes considers all the terms and has weights attributed, but the decision tree model goes through them sequentially, following the path from the root to the outer branches and leaves. Figure 8.26 only shows the top four layers, starting with the term "data." If "data" is present in the post, it's always data science. If "data" can't be found, it checks for the term "learn," and so it continues. A possible reason why this decision tree isn't performing well is the lack of pruning. When a decision tree is built it has many leaves, often too many. A tree is then pruned to a certain level to minimize overfitting. A big advantage of decision trees is the implicit interaction effects between words it

```
if data == False:
  if learn == False:
    if python == False:
      if tool == False: return 'gameofthrones'
      if tool == True: return 'datascience'
    if python == True: return 'datascience'
  if learn == True:
    if go == False:
      if wrong == False: return 'datascience'
      if wrong == True: return 'gameofthrones'
    if go == True:
      if upload == False: return 'gameofthrones'
      if upload == True: return 'datascience'
if data == True: return 'datascience'
```

Figure 8.26 Decision tree model tree structure representation

takes into account when constructing the branches. When multiple terms together create a stronger classification than single terms, the decision tree will actually outperform the Naïve Bayes. We won't go into the details of that here, but consider this one of the next steps you could take to improve the model.

We now have two classification models that give us insight into how the two contents of the subreddits differ. The last step would be to share this newfound information with other people.

8.3.8 Step 6: Presentation and automation

As a last step we need to use what we learned and either turn it into a useful application or present our results to others. The last chapter of this book discusses building an interactive application, as this is a project in itself. For now we'll content ourselves with a nice way to convey our findings. A nice graph or, better yet, an interactive graph, can catch the eye; it's the icing on the presentation cake. While it's easy and tempting to represent the numbers as such or a bar chart at most, it could be nice to go one step further.

For instance, to represent the Naïve Bayes model, we could use a force graph (figure 8.27), where the bubble and link size represent how strongly related a word is to the "game of thrones" or "data science" subreddits. Notice how the words on the bubbles are often cut off; remember this is because of the stemming we applied.

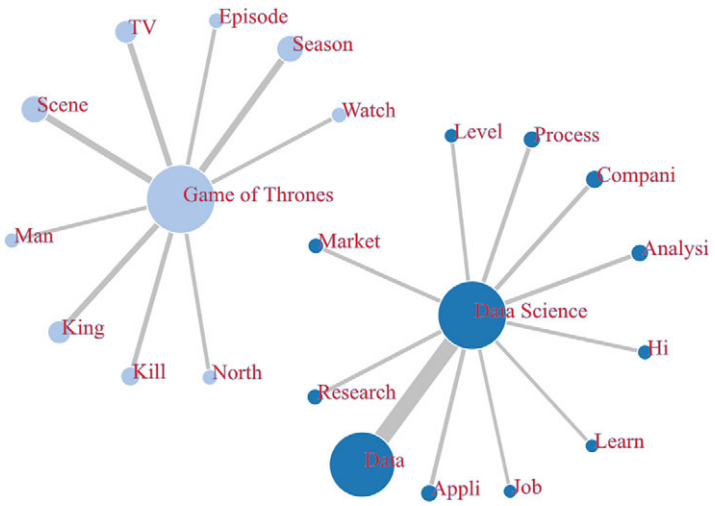

Figure 8.27 Interactive force graph with the top 20 Naïve Bayes significant terms and their weights

While figure 8.27 in itself is static, you can open the HTML file "forceGraph.html" to enjoy the d3.js force graph effect as explained earlier in this chapter. d3.js is outside of this book's scope but you don't need an elaborate knowledge of d3.js to use it. An extensive set of examples can be used with minimal adjustments to the code provided at https://github.com/mbostock/d3/wiki/Gallery. All you need is common sense and

a minor knowledge of JavaScript. The code for the force graph example can found at http://bl.ocks.org/mbostock/4062045.

We can also represent our decision tree in a rather original way. We could go for a fancy version of an actual tree diagram, but the following sunburst diagram is more original and equally fun to use.

Figure 8.28 shows the top layer of the sunburst diagram. It's possible to zoom in by clicking a circle segment. You can zoom back out by clicking the center circle. The code for this example can be found at http://bl.ocks.org/metmajer/5480307.

Figure 8.28 Sunburst diagram created from the top four branches of the decision tree model

Showing your results in an original way can be key to a successful project. People never appreciate the effort you've put into achieving your results if you can't communicate them and they're meaningful to them. An original data visualization here and there certainly helps with this.

8.4 Summary

- Text mining is widely used for things such as entity identification, plagiarism detection, topic identification, translation, fraud detection, spam filtering, and more.
- Python has a mature toolkit for text mining called NLTK, or the natural language toolkit. NLTK is good for playing around and learning the ropes; for real-life applications, however, Scikit-learn is usually considered more "production-ready." Scikit-learn is extensively used in previous chapters.
- The data preparation of textual data is more intensive than numerical data preparation and involves extra techniques, such as
  - *Stemming*—Cutting the end of a word in a smart way so it can be matched with some conjugated or plural versions of this word.
  - *Lemmatization*—Like stemming, it's meant to remove doubles, but unlike stemming, it looks at the meaning of the word.
  - *Stop word filtering*—Certain words occur too often to be useful and filtering them out can significantly improve models. Stop words are often corpus-specific.
  - *Tokenization*—Cutting text into pieces. Tokens can be single words, combinations of words (n-grams), or even whole sentences.
  - *POS Tagging*—Part-of-speech tagging. Sometimes it can be useful to know what the function of a certain word within a sentence is to understand it better.
- In our case study we attempted to distinguish Reddit posts on "Game of Thrones" versus posts on "data science." In this endeavor we tried both the Naïve Bayes and decision tree classifiers. Naïve Bayes assumes all features to be independent of one another; the decision tree classifier assumes dependency, allowing for different models.
- In our example, Naïve Bayes yielded the better model, but very often the decision tree classifier does a better job, usually when more data is available.
- We determined the performance difference using a confusion matrix we calculated after applying both models on new (but labeled) data.
- When presenting findings to other people, it can help to include an interesting data visualization capable of conveying your results in a memorable way.