MODULE-5

Natural Language Processing

They have been at a great feast of languages, and stolen the scraps.

William Shakespeare

Natural language processing (NLP) refers to computational techniques involving language. It's a broad field, but we'll look at a few techniques both simple and not simple.

Word Clouds

In Chapter 1, we computed word counts of users' interests. One approach to visualizing words and counts is word clouds, which artistically lay out the words with sizes proportional to their counts.

Generally, though, data scientists don't think much of word clouds, in large part because the placement of the words doesn't mean anything other than "here's some space where I was able to fit a word."

If you ever are forced to create a word cloud, think about whether you can make the axes convey something. For example, imagine that, for each of some collection of data science–related buzzwords, you have two numbers between 0 and 100 — the first representing how frequently it appears in job postings, the second how frequently it appears on resumes:

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
         ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 60),
         ("data science", 60, 70), ("analytics", 90, 3),
         ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0),
         ("actionable insights", 40, 30), ("think out of the box", 45, 10),
         ("self-starter", 30, 50), ("customer focus", 65, 15),
         ("thought leadership", 35, 35)]
```

The word cloud approach is just to arrange the words on a page in a cool-looking font (Figure 20-1).

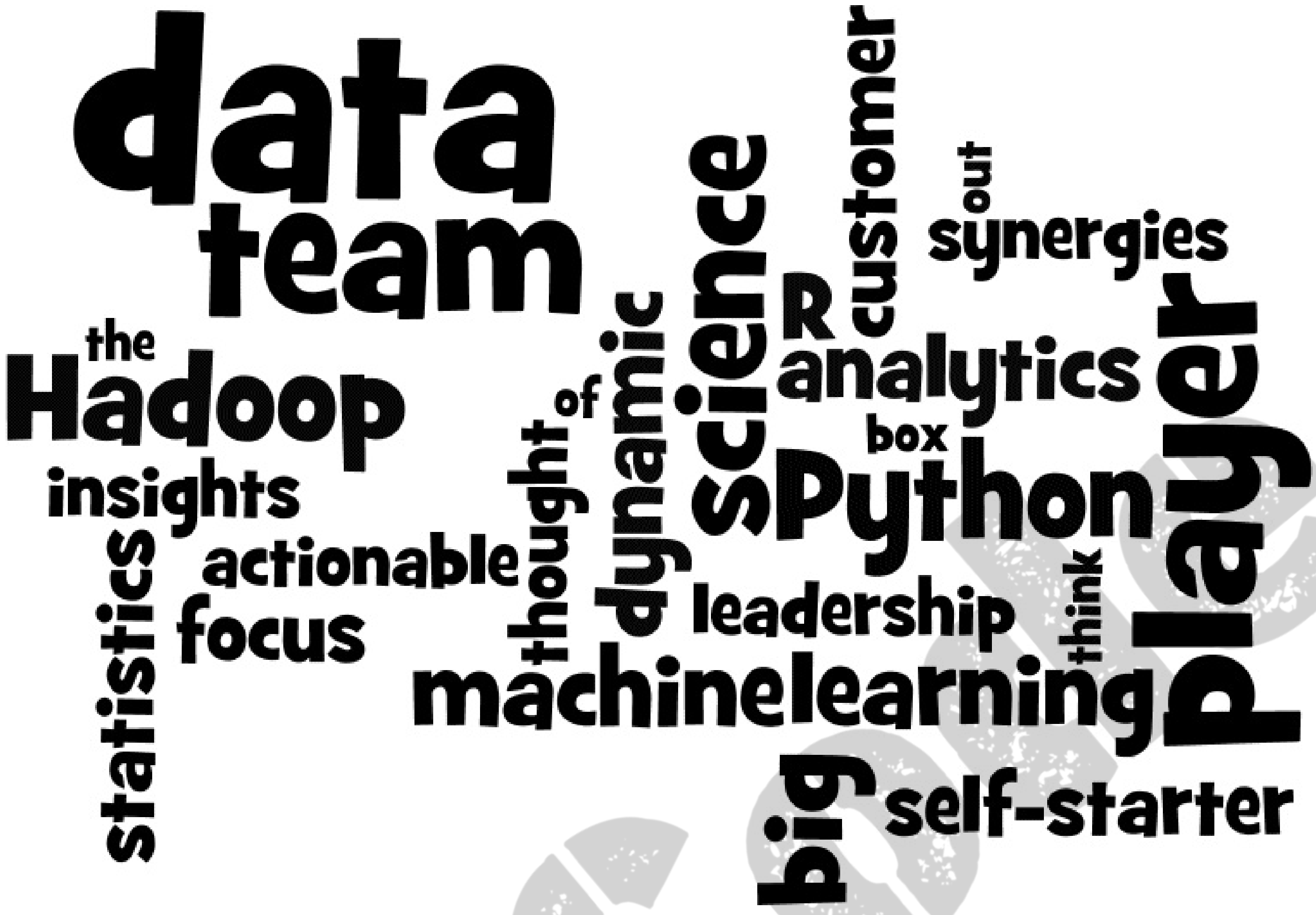

Figure 20-1. Buzzword cloud

This looks neat but doesn’t really tell us anything. A more interesting approach might be to scatter them so that horizontal position indicates posting popularity and vertical position indicates resume popularity, which produces a visualization that conveys a few insights (Figure 20-2):

```
def text_size(total):
    """equals 8 if total is 0, 28 if total is 200"""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Popularity on Job Postings")
plt.ylabel("Popularity on Resumes")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

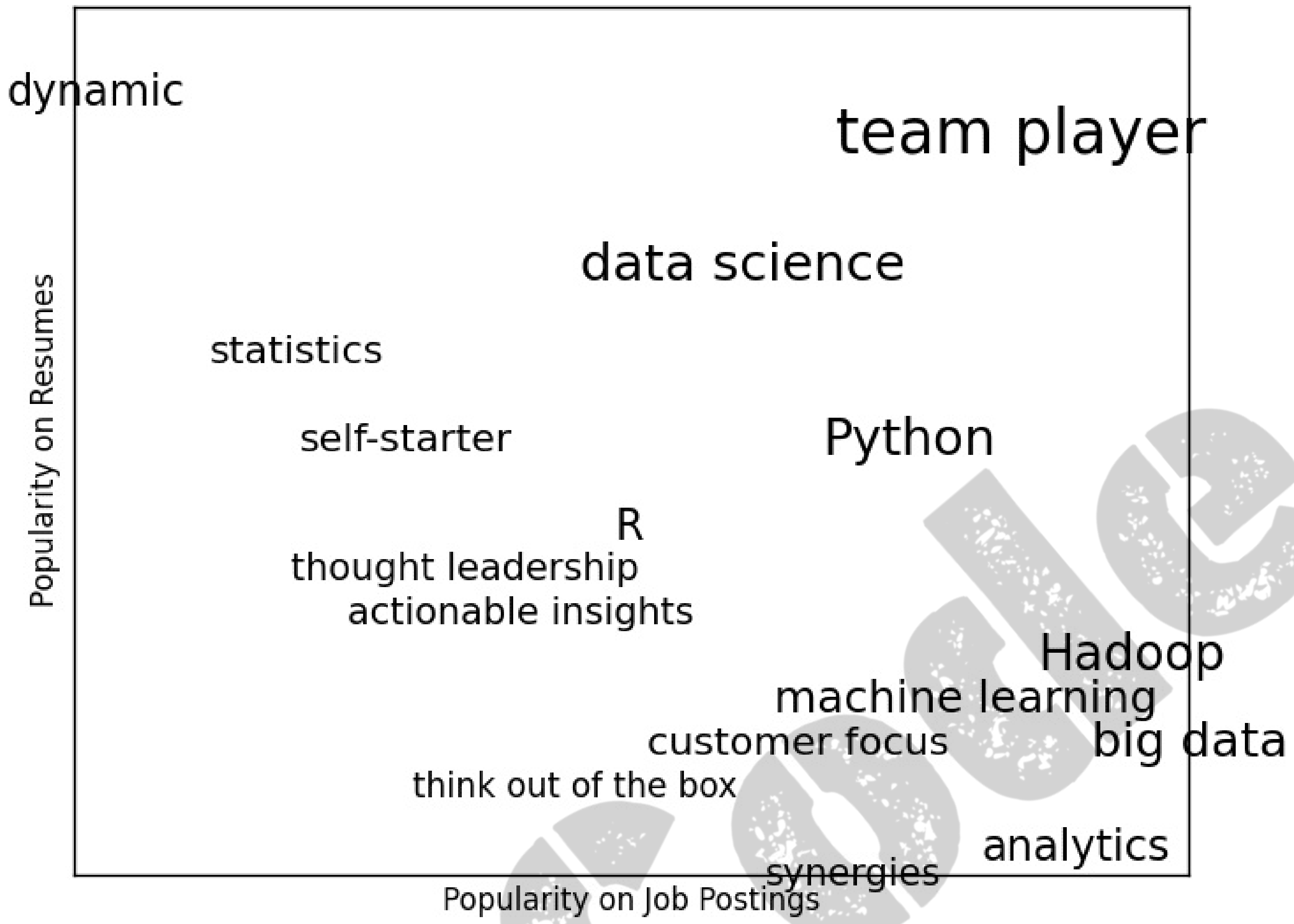

Figure 20-2. A more meaningful (if less attractive) word cloud

n-gram Models

The DataSciencester VP of Search Engine Marketing wants to create thousands of web pages about data science so that your site will rank higher in search results for data science–related terms. (You attempt to explain to her that search engine algorithms are clever enough that this won't actually work, but she refuses to listen.)

Of course, she doesn't want to write thousands of web pages, nor does she want to pay a horde of "content strategists" to do so. Instead she asks you whether you can somehow programatically generate these web pages. To do this, we'll need some way of modeling language.

One approach is to start with a corpus of documents and learn a statistical model of language. In our case, we'll start with Mike Loukides's essay "What is data science?"

As in Chapter 9, we'll use `requests` and `BeautifulSoup` to retrieve the data. There are a couple of issues worth calling attention to.

The first is that the apostrophes in the text are actually the Unicode character `u"\u2019"`. We'll create a helper function to replace them with normal apostrophes:

```
def fix_unicode(text):
    return text.replace(u"\u2019", "'")
```

The second issue is that once we get the text of the web page, we'll want to split it into a sequence of words and periods (so that we can tell where sentences end). We can do this using `re.findall()`:

```
from bs4 import BeautifulSoup
import requests
url = "http://radar.oreilly.com/2010/06/what-is-data-science.html"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')

content = soup.find("div", "entry-content")   # find entry-content div
regex = r"[\w']+|[\.]"                         # matches a word or a period

document = []

for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

We certainly could (and likely should) clean this data further. There is still some amount of extraneous text in the document (for example, the first word is "Section"), and we've split on midsentence periods (for example, in "Web 2.0"), and there are a handful of captions and lists sprinkled throughout. Having said that, we'll work with the `document` as it is.

Now that we have the text as a sequence of words, we can model language in the following way: given some starting word (say "book") we look at all the words that follow it in the source documents (here "isn't," "a," "shows," "demonstrates," and "teaches"). We randomly choose one of these to be the next word, and we repeat the process until we get to a period, which signifies the end of the sentence. We call this a *bigram model*, as it is determined completely by the frequencies of the bigrams (word pairs) in the original data.

What about a starting word? We can just pick randomly from words that *follow* a period. To start, let's precompute the possible word transitions. Recall that `zip` stops when any of its inputs is done, so that `zip(document, document[1:])` gives us precisely the pairs of consecutive elements of `document`:

```
bigrams = zip(document, document[1:])
transitions = defaultdict(list)
for prev, current in bigrams:
    transitions[prev].append(current)
```

Now we're ready to generate sentences:

```
def generate_using_bigrams():
    current = "."   # this means the next word will start a sentence
    result = []
    while True:
        next_word_candidates = transitions[current]    # bigrams (current, _)
        current = random.choice(next_word_candidates)  # choose one at random
        result.append(current)                         # append it to results
        if current == ".": return " ".join(result)     # if "." we're done
```

The sentences it produces are gibberish, but they're the kind of gibberish you might put on your website if you were trying to sound data-sciencey. For example:

> If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.
>
> Bigram Model

We can make the sentences less gibberishy by looking at *trigrams*, triplets of consecutive words. (More generally, you might look at *n-grams* consisting of *n* consecutive words, but three will be plenty for us.) Now the transitions will depend on the previous *two* words:

```
trigrams = zip(document, document[1:], document[2:])
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in trigrams:

    if prev == ".":              # if the previous "word" was a period
        starts.append(current)   # then this is a start word

    trigram_transitions[(prev, current)].append(next)
```

Notice that now we have to track the starting words separately. We can generate sentences in pretty much the same way:

```
def generate_using_trigrams():
    current = random.choice(starts)   # choose a random starting word
    prev = "."                        # and precede it with a '.'
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)

        prev, current = current, next_word
        result.append(current)

        if current == ".":
            return " ".join(result)
```

This produces better sentences like:

> In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That’s not a question we could even have asked a few years there has been instrumented.
>
> Trigram Model

Of course, they sound better because at each step the generation process has fewer choices, and at many steps only a single choice. This means that you frequently generate sentences (or at least long phrases) that were seen verbatim in the original data. Having more data would help; it would also work better if you collected *n*-grams from multiple essays about data science.

Grammars

A different approach to modeling language is with *grammars*, rules for generating acceptable sentences. In elementary school, you probably learned about parts of speech and how to combine them. For example, if you had a really bad English teacher, you might say that a sentence necessarily consists of a *noun* followed by a *verb*. If you then have a list of nouns and verbs, you can generate sentences according to the rule.

We'll define a slightly more complicated grammar:

```
grammar = {
    "_S"  : ["_NP _VP"],
    "_NP" : ["_N",
             "_A _NP _P _A _N"],
    "_VP" : ["_V",
             "_V _NP"],
    "_N"  : ["data science", "Python", "regression"],
    "_A"  : ["big", "linear", "logistic"],
    "_P"  : ["about", "near"],
    "_V"  : ["learns", "trains", "tests", "is"]
}
```

I made up the convention that names starting with underscores refer to *rules* that need further expanding, and that other names are *terminals* that don't need further processing.

So, for example, `"_S"` is the "sentence" rule, which produces a `"_NP"` ("noun phrase") rule followed by a `"_VP"` ("verb phrase") rule.

The verb phrase rule can produce either the `"_V"` ("verb") rule, or the verb rule followed by the noun phrase rule.

Notice that the `"_NP"` rule contains itself in one of its productions. Grammars can be recursive, which allows even finite grammars like this to generate infinitely many different sentences.

How do we generate sentences from this grammar? We'll start with a list containing the sentence rule `["_S"]`. And then we'll repeatedly expand each rule by replacing it with a randomly chosen one of its productions. We stop when we have a list consisting solely of terminals.

For example, one such progression might look like:

```
['_S']
['_NP','_VP']
['_N','_VP']
['Python','_VP']
['Python','_V','_NP']
['Python','trains','_NP']
['Python','trains','_A','_NP','_P','_A','_N']
['Python','trains','logistic','_NP','_P','_A','_N']
['Python','trains','logistic','_N','_P','_A','_N']
['Python','trains','logistic','data science','_P','_A','_N']
['Python','trains','logistic','data science','about','_A', '_N']
['Python','trains','logistic','data science','about','logistic','_N']
['Python','trains','logistic','data science','about','logistic','Python']
```

How do we implement this? Well, to start, we'll create a simple helper function to identify terminals:

```python
def is_terminal(token):
    return token[0] != "_"
```

Next we need to write a function to turn a list of tokens into a sentence. We'll look for the first nonterminal token. If we can't find one, that means we have a completed sentence and we're done.

If we do find a nonterminal, then we randomly choose one of its productions. If that production is a terminal (i.e., a word), we simply replace the token with it. Otherwise it's a sequence of space-separated nonterminal tokens that we need to `split` and then splice into the current tokens. Either way, we repeat the process on the new set of tokens.

Putting it all together we get:

```python
def expand(grammar, tokens):
    for i, token in enumerate(tokens):

        # skip over terminals
        if is_terminal(token): continue

        # if we get here, we found a non-terminal token
        # so we need to choose a replacement at random
        replacement = random.choice(grammar[token])

        if is_terminal(replacement):
            tokens[i] = replacement
        else:
            tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

        # now call expand on the new list of tokens
        return expand(grammar, tokens)

    # if we get here we had all terminals and are done
    return tokens
```

And now we can start generating sentences:

```python
def generate_sentence(grammar):
    return expand(grammar, ["_S"])
```

Try changing the grammar — add more words, add more rules, add your own parts of speech — until you're ready to generate as many web pages as your company needs.

Grammars are actually more interesting when they're used in the other direction. Given a sentence we can use a grammar to *parse* the sentence. This then allows us to identify subjects and verbs and helps us make sense of the sentence.

Using data science to generate text is a neat trick; using it to *understand* text is more magical. (See "For Further Investigation" for libraries that you could use for this.)

An Aside: Gibbs Sampling

Generating samples from some distributions is easy. We can get uniform random variables with:

```
random.random()
```

and normal random variables with:

```
inverse_normal_cdf(random.random())
```

But some distributions are harder to sample from. *Gibbs sampling* is a technique for generating samples from multidimensional distributions when we only know some of the conditional distributions.

For example, imagine rolling two dice. Let *x* be the value of the first die and *y* be the sum of the dice, and imagine you wanted to generate lots of (x, y) pairs. In this case it's easy to generate the samples directly:

```
def roll_a_die():
    return random.choice([1,2,3,4,5,6])

def direct_sample():
    d1 = roll_a_die()
    d2 = roll_a_die()
    return d1, d1 + d2
```

But imagine that you only knew the conditional distributions. The distribution of *y* conditional on *x* is easy — if you know the value of *x*, *y* is equally likely to be *x* + 1, *x* + 2, *x* + 3, *x* + 4, *x* + 5, or *x* + 6:

```
def random_y_given_x(x):
    """equally likely to be x + 1, x + 2, ... , x + 6"""
    return x + roll_a_die()
```

The other direction is more complicated. For example, if you know that *y* is 2, then necessarily *x* is 1 (since the only way two dice can sum to 2 is if both of them are 1). If you know *y* is 3, then *x* is equally likely to be 1 or 2. Similarly, if *y* is 11, then *x* has to be either 5 or 6:

```
def random_x_given_y(y):
    if y <= 7:
        # if the total is 7 or less, the first die is equally likely to be
        # 1, 2, ..., (total - 1)
        return random.randrange(1, y)
    else:
        # if the total is 7 or more, the first die is equally likely to be
        # (total - 6), (total - 5), ..., 6
        return random.randrange(y - 6, 7)
```

The way Gibbs sampling works is that we start with any (valid) value for *x* and *y* and then repeatedly alternate replacing *x* with a random value picked conditional on *y* and replacing *y* with a random value picked conditional on *x*. After a number of iterations, the resulting values of *x* and *y* will represent a sample from the unconditional joint distribution:

```
def gibbs_sample(num_iters=100):
    x, y = 1, 2 # doesn't really matter
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

You can check that this gives similar results to the direct sample:

```
def compare_distributions(num_samples=1000):
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

We'll use this technique in the next section.

Topic Modeling

When we built our Data Scientists You Should Know recommender in Chapter 1, we simply looked for exact matches in people’s stated interests.

A more sophisticated approach to understanding our users’ interests might try to identify the *topics* that underlie those interests. A technique called *Latent Dirichlet Analysis* (LDA) is commonly used to identify common topics in a set of documents. We’ll apply it to documents that consist of each user’s interests.

LDA has some similarities to the Naive Bayes Classifier we built in Chapter 13, in that it assumes a probabilistic model for documents. We’ll gloss over the hairier mathematical details, but for our purposes the model assumes that:

- There is some fixed number *K* of topics.
- There is a random variable that assigns each topic an associated probability distribution over words. You should think of this distribution as the probability of seeing word *w* given topic *k*.
- There is another random variable that assigns each document a probability distribution over topics. You should think of this distribution as the mixture of topics in document *d*.
- Each word in a document was generated by first randomly picking a topic (from the document’s distribution of topics) and then randomly picking a word (from the topic’s distribution of words).

In particular, we have a collection of `documents` each of which is a `list` of words. And we have a corresponding collection of `document_topics` that assigns a topic (here a number between 0 and *K* – 1) to each word in each document.

So that the fifth word in the fourth document is:

```
documents[3][4]
```

and the topic from which that word was chosen is:

```
document_topics[3][4]
```

This very explicitly defines each document’s distribution over topics, and it implicitly defines each topic’s distribution over words.

We can estimate the likelihood that topic 1 produces a certain word by comparing how many times topic 1 produces that word with how many times topic 1 produces *any* word. (Similarly, when we built a spam filter in Chapter 13, we compared how many times each word appeared in spams with the total number of words appearing in spams.)

Although these topics are just numbers, we can give them descriptive names by looking at the words on which they put the heaviest weight. We just have to somehow generate the `document_topics`. This is where Gibbs sampling comes into play.

We start by assigning every word in every document a topic completely at random. Now we go through each document one word at a time. For that word and document, we construct weights for each topic that depend on the (current) distribution of topics in that document and the (current) distribution of words for that topic. We then use those weights to sample a new topic for that word. If we iterate this process many times, we will end up with a joint sample from the topic-word distribution and the document-topic distribution.

To start with, we'll need a function to randomly choose an index based on an arbitrary set of weights:

```
def sample_from(weights):
    """returns i with probability weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()      # uniform between 0 and total
    for i, w in enumerate(weights):
        rnd -= w                       # return the smallest i such that
        if rnd <= 0: return i          # weights[0] + ... + weights[i] >= rnd
```

For instance, if you give it weights `[1, 1, 3]` then one-fifth of the time it will return 0, one-fifth of the time it will return 1, and three-fifths of the time it will return 2.

Our documents are our users' interests, which look like:

```
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

And we'll try to find `K = 4` topics.

In order to calculate the sampling weights, we'll need to keep track of several counts. Let's first create the data structures for them.

How many times each topic is assigned to each document:

```
# a list of Counters, one for each document
document_topic_counts = [Counter() for _ in documents]
```

How many times each word is assigned to each topic:

```
# a list of Counters, one for each topic
```

```
topic_word_counts = [Counter() for _ in range(K)]
```

The total number of words assigned to each topic:

```
# a list of numbers, one for each topic
topic_counts = [0 for _ in range(K)]
```

The total number of words contained in each document:

```
# a list of numbers, one for each document
document_lengths = map(len, documents)
```

The number of distinct words:

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

And the number of documents:

```
D = len(documents)
```

For example, once we populate these, we can find, for example, the number of words in `documents[3]` associated with topic 1 as:

```
document_topic_counts[3][1]
```

And we can find the number of times *nlp* is associated with topic 2 as:

```
topic_word_counts[2]["nlp"]
```

Now we're ready to define our conditional probability functions. As in Chapter 13, each has a smoothing term that ensures every topic has a nonzero chance of being chosen in any document and that every word has a nonzero chance of being chosen for any topic:

```
def p_topic_given_document(topic, d, alpha=0.1):
    """the fraction of words in document _d_
    that are assigned to _topic_ (plus some smoothing)"""

    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word, topic, beta=0.1):
    """the fraction of words assigned to _topic_
    that equal _word_ (plus some smoothing)"""

    return ((topic_word_counts[topic][word] + beta) /
            (topic_counts[topic] + W * beta))
```

We'll use these to create the weights for updating topics:

```
def topic_weight(d, word, k):
    """given a document and a word in that document,
    return the weight for the kth topic"""

    return p_word_given_topic(word, k) * p_topic_given_document(k, d)
```

```
def choose_new_topic(d, word):
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

There are solid mathematical reasons why `topic_weight` is defined the way it is, but their details would lead us too far afield. Hopefully it makes at least intuitive sense that — given a word and its document — the likelihood of any topic choice depends on both how likely that topic is for the document and how likely that word is for the topic.

This is all the machinery we need. We start by assigning every word to a random topic, and populating our counters appropriately:

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

Our goal is to get a joint sample of the topics-words distribution and the documents-topics distribution. We do this using a form of Gibbs sampling that uses the conditional probabilities defined previously:

```
for iter in range(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):

            # remove this word / topic from the counts
            # so that it doesn't influence the weights
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
            topic_counts[topic] -= 1
            document_lengths[d] -= 1

            # choose a new topic based on the weights
            new_topic = choose_new_topic(d, word)
            document_topics[d][i] = new_topic

            # and now add it back to the counts
            document_topic_counts[d][new_topic] += 1
            topic_word_counts[new_topic][word] += 1
            topic_counts[new_topic] += 1
            document_lengths[d] += 1
```

What are the topics? They're just numbers 0, 1, 2, and 3. If we want names for them we have to do that ourselves. Let's look at the five most heavily weighted words for each (Table 20-1):

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0: print k, word, count
```

Table 20-1. Most common words per topic

Topic 0	Topic 1	Topic 2	Topic 3

Java	R	HBase	regression
Big Data	statistics	Postgres	libsvm
Hadoop	Python	MongoDB	scikit-learn
deep learning	probability	Cassandra	machine learning
artificial intelligence	pandas	NoSQL	neural networks

Based on these I'd probably assign topic names:

```
topic_names = ["Big Data and programming languages",
               "Python and statistics",
               "databases",
               "machine learning"]
```

at which point we can see how the model assigns topics to each user's interests:

```
for document, topic_counts in zip(documents, document_topic_counts):
    print document
    for topic, count in topic_counts.most_common():
        if count > 0:
            print topic_names[topic], count,
    print
```

which gives:

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

and so on. Given the "ands" we needed in some of our topic names, it's possible we should use more topics, although most likely we don't have enough data to successfully learn them.

Network Analysis

Your connections to all the things around you literally define who you are.

Aaron O’Connell

Many interesting data problems can be fruitfully thought of in terms of *networks*, consisting of *nodes* of some type and the *edges* that join them.

For instance, your Facebook friends form the nodes of a network whose edges are friendship relations. A less obvious example is the World Wide Web itself, with each web page a node, and each hyperlink from one page to another an edge.

Facebook friendship is mutual — if I am Facebook friends with you than necessarily you are friends with me. In this case, we say that the edges are *undirected*. Hyperlinks are not — my website links to whitehouse.gov, but (for reasons inexplicable to me) whitehouse.gov refuses to link to my website. We call these types of edges *directed*. We’ll look at both kinds of networks.

Betweenness Centrality

In Chapter 1, we computed the key connectors in the DataSciencester network by counting the number of friends each user had. Now we have enough machinery to look at other approaches. Recall that the network (Figure 21-1) comprised users:

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

and friendships:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4),
               (4, 5), (5, 6), (5, 7), (6, 8), (7, 8), (8, 9)]
```

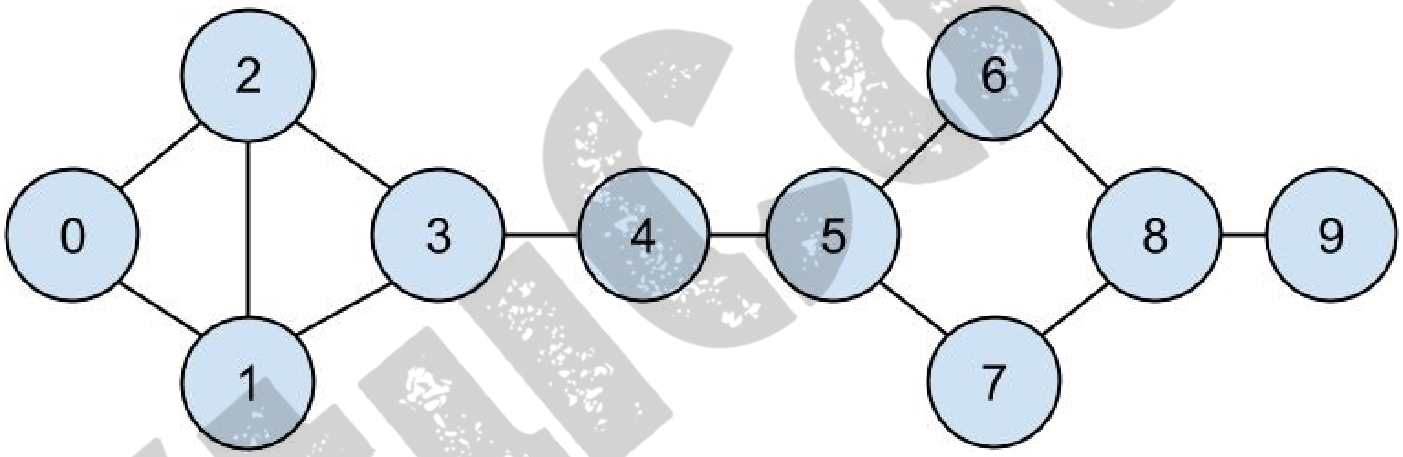

Figure 21-1. The DataSciencester network

We also added friend lists to each user `dict`:

```
for user in users:
    user["friends"] = []

for i, j in friendships:
    # this works because users[i] is the user whose id is i
    users[i]["friends"].append(users[j]) # add i as a friend of j
    users[j]["friends"].append(users[i]) # add j as a friend of i
```

When we left off we were dissatisfied with our notion of *degree centrality*, which didn't really agree with our intuition about who were the key connectors of the network.

An alternative metric is *betweenness centrality*, which identifies people who frequently are on the shortest paths between pairs of other people. In particular, the betweenness centrality of node *i* is computed by adding up, for every other pair of nodes *j* and *k*, the proportion of shortest paths between node *j* and node *k* that pass through *i*.

That is, to figure out Thor's betweenness centrality, we'll need to compute all the shortest paths between all pairs of people who aren't Thor. And then we'll need to count how many of those shortest paths pass through Thor. For instance, the only shortest path between Chi (`id` 3) and Clive (`id` 5) passes through Thor, while neither of the two shortest paths between Hero (`id` 0) and Chi (`id` 3) does.

So, as a first step, we'll need to figure out the shortest paths between all pairs of people. There are some pretty sophisticated algorithms for doing so efficiently, but (as is almost always the case) we will use a less efficient, easier-to-understand algorithm.

This algorithm (an implementation of breadth-first search) is one of the more complicated ones in the book, so let's talk through it carefully:

1. Our goal is a function that takes a `from_user` and finds *all* shortest paths to every other user.

2. We'll represent a path as `list` of user IDs. Since every path starts at `from_user`, we won't include her ID in the list. This means that the length of the list representing the path will be the length of the path itself.

3. We'll maintain a dictionary `shortest_paths_to` where the keys are user IDs and the values are lists of paths that end at the user with the specified ID. If there is a unique shortest path, the list will just contain that one path. If there are multiple shortest paths, the list will contain all of them.

4. We'll also maintain a queue `frontier` that contains the users we want to explore in the order we want to explore them. We'll store them as pairs `(prev_user, user)` so that we know how we got to each one. We initialize the queue with all the neighbors of `from_user`. (We haven't ever talked about queues, which are data structures optimized for "add to the end" and "remove from the front" operations. In Python, they are implemented as `collections.deque` which is actually a double-ended queue.)

5. As we explore the graph, whenever we find new neighbors that we don't already know shortest paths to, we add them to the end of the queue to explore later, with the current user as `prev_user`.

6. When we take a user off the queue, and we've never encountered that user before, we've definitely found one or more shortest paths to him — each of the shortest paths to `prev_user` with one extra step added.

7. When we take a user off the queue and we *have* encountered that user before, then either we've found another shortest path (in which case we should add it) or we've found a longer path (in which case we shouldn't).

8. When no more users are left on the queue, we've explored the whole graph (or, at least, the parts of it that are reachable from the starting user) and we're done.

We can put this all together into a (large) function:

```
from collections import deque

def shortest_paths_from(from_user):

    # a dictionary from "user_id" to *all* shortest paths to that user
    shortest_paths_to = { from_user["id"] : [[]] }

    # a queue of (previous user, next user) that we need to check.
    # starts out with all pairs (from_user, friend_of_from_user)
    frontier = deque((from_user, friend)
                     for friend in from_user["friends"])

    # keep going until we empty the queue
    while frontier:

        prev_user, user = frontier.popleft()   # remove the user who's
        user_id = user["id"]                   # first in the queue

        # because of the way we're adding to the queue,
        # necessarily we already know some shortest paths to prev_user
        paths_to_prev_user = shortest_paths_to[prev_user["id"]]
        new_paths_to_user = [path + [user_id] for path in paths_to_prev_user]

        # it's possible we already know a shortest path
        old_paths_to_user = shortest_paths_to.get(user_id, [])

        # what's the shortest path to here that we've seen so far?
        if old_paths_to_user:
            min_path_length = len(old_paths_to_user[0])
        else:
            min_path_length = float('inf')

        # only keep paths that aren't too long and are actually new
        new_paths_to_user = [path
                             for path in new_paths_to_user
                             if len(path) <= min_path_length
                             and path not in old_paths_to_user]

        shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user

        # add never-seen neighbors to the frontier
        frontier.extend((user, friend)
                        for friend in user["friends"]
                        if friend["id"] not in shortest_paths_to)

    return shortest_paths_to
```

Now we can store these `dicts` with each node:

```
for user in users:
    user["shortest_paths"] = shortest_paths_from(user)
```

And we're finally ready to compute betweenness centrality. For every pair of nodes *i* and *j*, we know the *n* shortest paths from *i* to *j*. Then, for each of those paths, we just add *1/n* to the centrality of each node on that path:

```
for user in users:
    user["betweenness_centrality"] = 0.0

for source in users:
    source_id = source["id"]
    for target_id, paths in source["shortest_paths"].iteritems():
        if source_id < target_id:      # don't double count
            num_paths = len(paths)     # how many shortest paths?
            contrib = 1 / num_paths    # contribution to centrality
            for path in paths:
```

```
            for id in path:
                if id not in [source_id, target_id]:
                    users[id]["betweenness_centrality"] += contrib
```

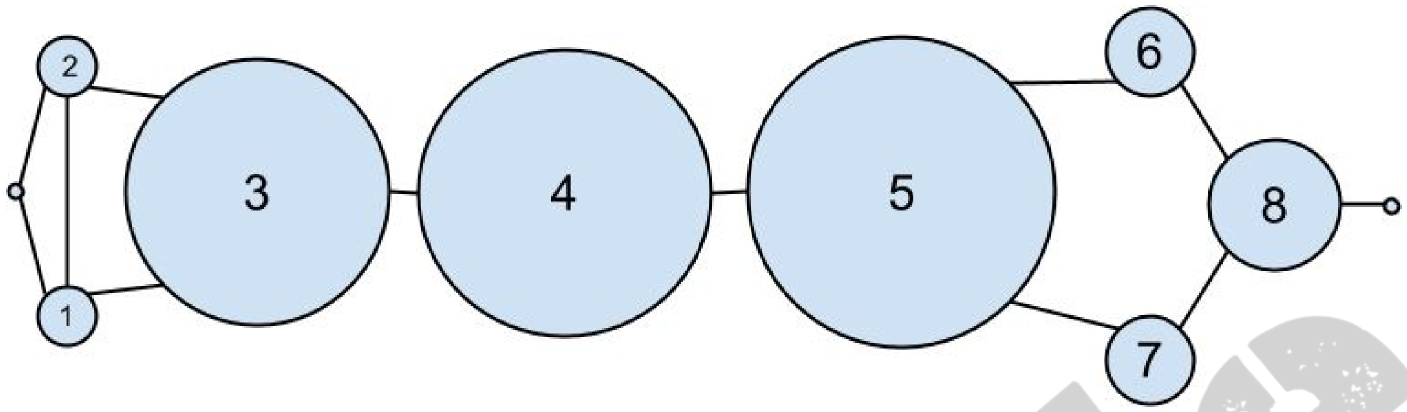

Figure 21-2. The DataSciencester network sized by betweenness centrality

As shown in Figure 21-2, users 0 and 9 have centrality 0 (as neither is on any shortest path between other users), whereas 3, 4, and 5 all have high centralities (as all three lie on many shortest paths).

> **NOTE**
>
> Generally the centrality numbers aren't that meaningful themselves. What we care about is how the numbers for each node compare to the numbers for other nodes.

Another measure we can look at is *closeness centrality*. First, for each user we compute her *farness*, which is the sum of the lengths of her shortest paths to each other user. Since we've already computed the shortest paths between each pair of nodes, it's easy to add their lengths. (If there are multiple shortest paths, they all have the same length, so we can just look at the first one.)

```
def farness(user):
    """the sum of the lengths of the shortest paths to each other user"""
    return sum(len(paths[0])
               for paths in user["shortest_paths"].values())
```

after which it's very little work to compute closeness centrality (Figure 21-3):

```
for user in users:
    user["closeness_centrality"] = 1 / farness(user)
```

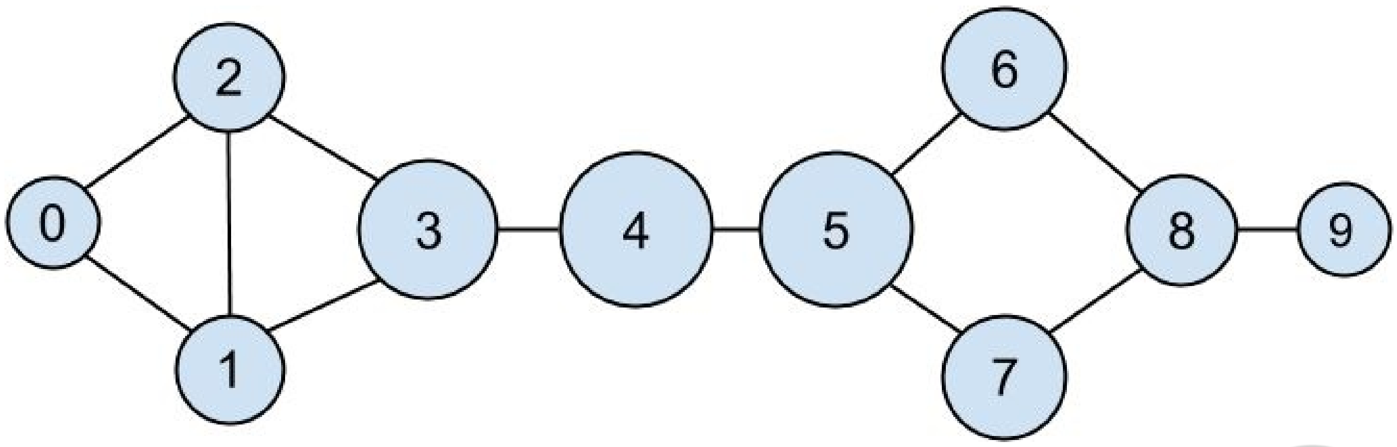

Figure 21-3. The DataSciencester network sized by closeness centrality

There is much less variation here — even the very central nodes are still pretty far from the nodes out on the periphery.

As we saw, computing shortest paths is kind of a pain. For this reason, betweenness and closeness centrality aren't often used on large networks. The less intuitive (but generally easier to compute) *eigenvector centrality* is more frequently used.

Eigenvector Centrality

In order to talk about eigenvector centrality, we have to talk about eigenvectors, and in order to talk about eigenvectors, we have to talk about matrix multiplication.

Matrix Multiplication

If *A* is a $n_1 \times k_1$ matrix and *B* is a $n_2 \times k_2$ matrix, and if $k_1 = n_2$, then their product *AB* is the $n_1 \times k_2$ matrix whose (i,j)th entry is:

$$A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{ik}B_{kj}$$

Which is just the `dot` product of the *i*th row of *A* (thought of as a vector) with the *j*th column of *B* (also thought of as a vector):

```
def matrix_product_entry(A, B, i, j):
    return dot(get_row(A, i), get_column(B, j))
```

after which we have:

```
def matrix_multiply(A, B):
    n1, k1 = shape(A)
    n2, k2 = shape(B)
    if k1 != n2:
        raise ArithmeticError("incompatible shapes!")

    return make_matrix(n1, k2, partial(matrix_product_entry, A, B))
```

Notice that if *A* is a $n \times k$ matrix and *B* is a $k \times 1$ matrix, then *AB* is a $n \times 1$ matrix. If we treat a vector as a one-column matrix, we can think of *A* as a function that maps *k*-dimensional vectors to *n*-dimensional vectors, where the function is just matrix multiplication.

Previously we represented vectors simply as lists, which isn't quite the same:

```
v = [1, 2, 3]
v_as_matrix = [[1],
               [2],
               [3]]
```

So we'll need some helper functions to convert back and forth between the two representations:

```
def vector_as_matrix(v):
    """returns the vector v (represented as a list) as a n x 1 matrix"""
    return [[v_i] for v_i in v]

def vector_from_matrix(v_as_matrix):
    """returns the n x 1 matrix as a list of values"""
    return [row[0] for row in v_as_matrix]
```

after which we can define the matrix operation using `matrix_multiply`:

```
def matrix_operate(A, v):
    v_as_matrix = vector_as_matrix(v)
    product = matrix_multiply(A, v_as_matrix)
    return vector_from_matrix(product)
```

When *A* is a *square* matrix, this operation maps *n*-dimensional vectors to other *n*-dimensional vectors. It's possible that, for some matrix *A* and vector *v*, when *A* operates on *v* we get back a scalar multiple of *v*. That is, that the result is a vector that points in the same direction as *v*. When this happens (and when, in addition, *v* is not a vector of all zeroes), we call *v* an *eigenvector* of *A*. And we call the multiplier an *eigenvalue*.

One possible way to find an eigenvector of *A* is by picking a starting vector *v*, applying `matrix_operate`, rescaling the result to have magnitude 1, and repeating until the process converges:

```
def find_eigenvector(A, tolerance=0.00001):
    guess = [random.random() for __ in A]

    while True:
        result = matrix_operate(A, guess)
        length = magnitude(result)
        next_guess = scalar_multiply(1/length, result)

        if distance(guess, next_guess) < tolerance:
            return next_guess, length   # eigenvector, eigenvalue

        guess = next_guess
```

By construction, the returned `guess` is a vector such that, when you apply `matrix_operate` to it and rescale it to have length 1, you get back (a vector very close to) itself. Which means it's an eigenvector.

Not all matrices of real numbers have eigenvectors and eigenvalues. For example the matrix:

```
rotate = [[ 0, 1],
          [-1, 0]]
```

rotates vectors 90 degrees clockwise, which means that the only vector it maps to a scalar multiple of itself is a vector of zeroes. If you tried `find_eigenvector(rotate)` it would run forever. Even matrices that have eigenvectors can sometimes get stuck in cycles. Consider the matrix:

```
flip = [[0, 1],
        [1, 0]]
```

This matrix maps any vector `[x, y]` to `[y, x]`. This means that, for example, `[1, 1]` is an eigenvector with eigenvalue 1. However, if you start with a random vector with unequal coordinates, `find_eigenvector` will just repeatedly swap the coordinates forever. (Not-from-scratch libraries like NumPy use different methods that would work in this case.) Nonetheless, when `find_eigenvector` does return a result, that result is indeed an eigenvector.

Centrality

How does this help us understand the DataSciencester network?

To start with, we'll need to represent the connections in our network as an `adjacency_matrix`, whose (i,j)th entry is either 1 (if user *i* and user *j* are friends) or 0 (if they're not):

```
def entry_fn(i, j):
    return 1 if (i, j) in friendships or (j, i) in friendships else 0

n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

The eigenvector centrality for each user is then the entry corresponding to that user in the eigenvector returned by `find_eigenvector` (Figure 21-4):

> **NOTE**
>
> For technical reasons that are way beyond the scope of this book, any nonzero adjacency matrix necessarily has an eigenvector all of whose values are non-negative. And fortunately for us, for this `adjacency_matrix` our `find_eigenvector` function finds it.

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

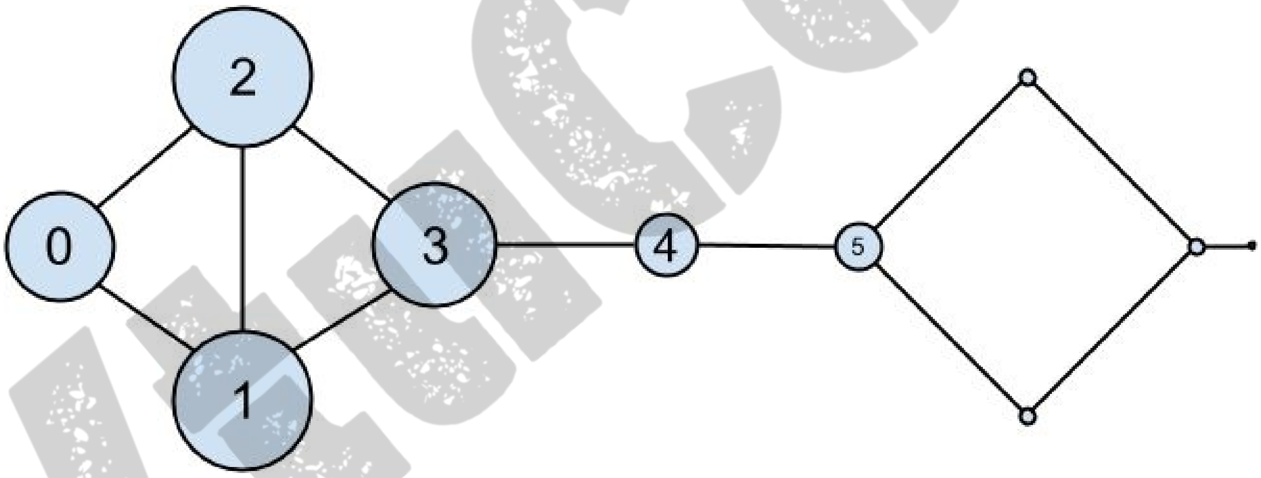

Figure 21-4. The DataSciencester network sized by eigenvector centrality

Users with high eigenvector centrality should be those who have a lot of connections and connections to people who themselves have high centrality.

Here users 1 and 2 are the most central, as they both have three connections to people who are themselves highly central. As we move away from them, people's centralities steadily drop off.

On a network this small, eigenvector centrality behaves somewhat erratically. If you try adding or subtracting links, you'll find that small changes in the network can dramatically change the centrality numbers. In a much larger network this would not particularly be the case.

We still haven't motivated why an eigenvector might lead to a reasonable notion of centrality. Being an eigenvector means that if you compute:

```
matrix_operate(adjacency_matrix, eigenvector_centralities)
```

the result is a scalar multiple of `eigenvector_centralities`.

If you look at how matrix multiplication works, `matrix_operate` produces a vector whose *i*th element is:

```
dot(get_row(adjacency_matrix, i), eigenvector_centralities)
```

which is precisely the sum of the eigenvector centralities of the users connected to user *i*.

In other words, eigenvector centralities are numbers, one per user, such that each user's value is a constant multiple of the sum of his neighbors' values. In this case centrality means being connected to people who themselves are central. The more centrality you are directly connected to, the more central you are. This is of course a circular definition — eigenvectors are the way of breaking out of the circularity.

Another way of understanding this is by thinking about what `find_eigenvector` is doing here. It starts by assigning each node a random centrality. It then repeats the following two steps until the process converges:

1. Give each node a new centrality score that equals the sum of its neighbors' (old) centrality scores.

2. Rescale the vector of centralities to have magnitude 1.

Although the mathematics behind it may seem somewhat opaque at first, the calculation itself is relatively straightforward (unlike, say, betweenness centrality) and is pretty easy to perform on even very large graphs.

Directed Graphs and PageRank

DataSciencester isn't getting much traction, so the VP of Revenue considers pivoting from a friendship model to an endorsement model. It turns out that no one particularly cares which data scientists are *friends* with one another, but tech recruiters care very much which data scientists are respected by other data scientists.

In this new model, we'll track endorsements `(source, target)` that no longer represent a reciprocal relationship, but rather that `source` endorses `target` as an awesome data scientist (Figure 21-5). We'll need to account for this asymmetry:

```
endorsements = [(0, 1), (1, 0), (0, 2), (2, 0), (1, 2),
                (2, 1), (1, 3), (2, 3), (3, 4), (5, 4),
                (5, 6), (7, 5), (6, 8), (8, 7), (8, 9)]

for user in users:
    user["endorses"] = []       # add one list to track outgoing endorsements
    user["endorsed_by"] = []    # and another to track endorsements

for source_id, target_id in endorsements:
    users[source_id]["endorses"].append(users[target_id])
    users[target_id]["endorsed_by"].append(users[source_id])
```

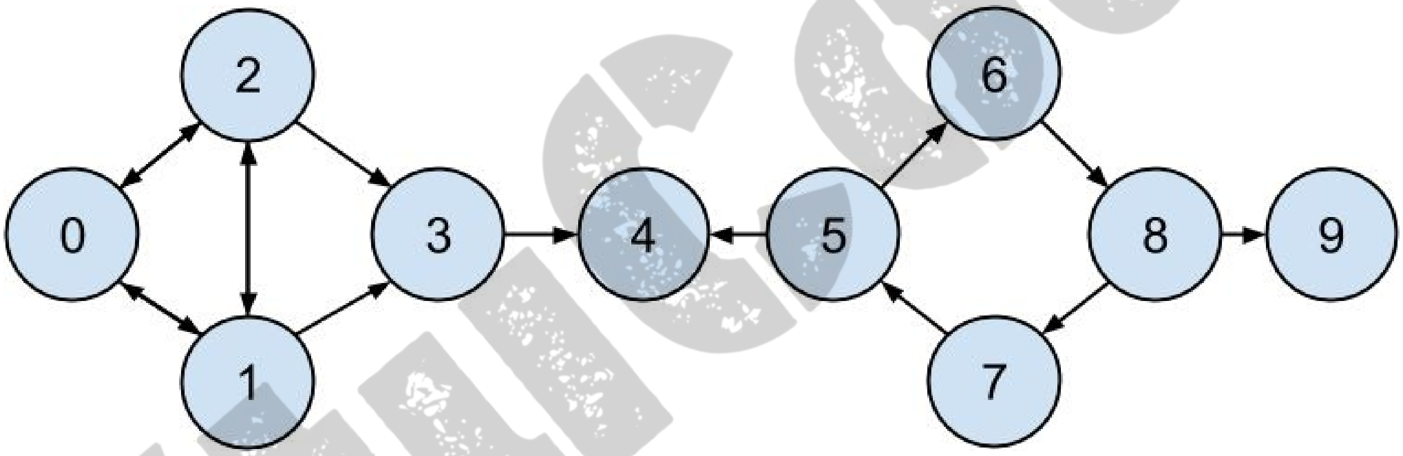

Figure 21-5. The DataSciencester network of endorsements

after which we can easily find the `most_endorsed` data scientists and sell that information to recruiters:

```
endorsements_by_id = [(user["id"], len(user["endorsed_by"]))
                      for user in users]

sorted(endorsements_by_id,
       key=lambda (user_id, num_endorsements): num_endorsements,
       reverse=True)
```

However, "number of endorsements" is an easy metric to game. All you need to do is create phony accounts and have them endorse you. Or arrange with your friends to endorse each other. (As users 0, 1, and 2 seem to have done.)

A better metric would take into account *who* endorses you. Endorsements from people who have a lot of endorsements should somehow count more than endorsements from people with few endorsements. This is the essence of the PageRank algorithm, used by

Google to rank websites based on which other websites link to them, which other websites link to those, and so on.

(If this sort of reminds you of the idea behind eigenvector centrality, it should.)

A simplified version looks like this:

1. There is a total of 1.0 (or 100%) PageRank in the network.

2. Initially this PageRank is equally distributed among nodes.

3. At each step, a large fraction of each node’s PageRank is distributed evenly among its outgoing links.

4. At each step, the remainder of each node’s PageRank is distributed evenly among all nodes.

```
def page_rank(users, damping = 0.85, num_iters = 100):

    # initially distribute PageRank evenly
    num_users = len(users)
    pr = { user["id"] : 1 / num_users for user in users }

    # this is the small fraction of PageRank
    # that each node gets each iteration
    base_pr = (1 - damping) / num_users

    for __ in range(num_iters):
        next_pr = { user["id"] : base_pr for user in users }
        for user in users:
            # distribute PageRank to outgoing links
            links_pr = pr[user["id"]] * damping
            for endorsee in user["endorses"]:
                next_pr[endorsee["id"]] += links_pr / len(user["endorses"])

        pr = next_pr

    return pr
```

PageRank (Figure 21-6) identifies user 4 (Thor) as the highest ranked data scientist.

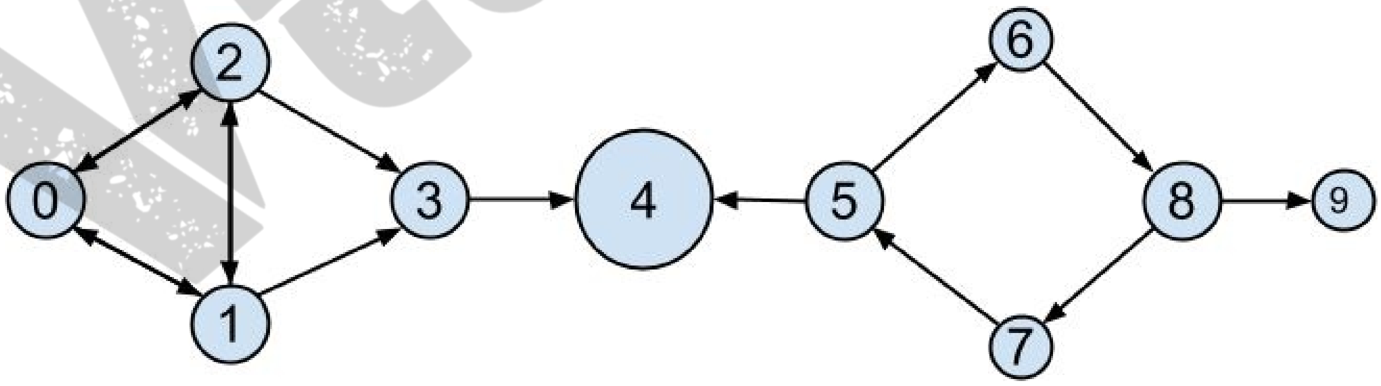

Figure 21-6. The DataSciencester network sized by PageRank

Even though he has fewer endorsements (2) than users 0, 1, and 2, his endorsements carry with them rank from their endorsements. Additionally, both of his endorsers endorsed only him, which means that he doesn’t have to divide their rank with anyone else.

Recommender Systems

O nature, nature, why art thou so dishonest, as ever to send men with these false recommendations into the world!

Henry Fielding

Another common data problem is producing *recommendations* of some sort. Netflix recommends movies you might want to watch. Amazon recommends products you might want to buy. Twitter recommends users you might want to follow. In this chapter, we'll look at several ways to use data to make recommendations.

In particular, we'll look at the data set of `users_interests` that we've used before:

```
users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"],
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intelligence"],
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

And we'll think about the problem of recommending new interests to a user based on her currently specified interests.

Manual Curation

Before the Internet, when you needed book recommendations you would go to the library, where a librarian was available to suggest books that were relevant to your interests or similar to books you liked.

Given DataSciencester's limited number of users and interests, it would be easy for you to spend an afternoon manually recommending interests for each user. But this method doesn't scale particularly well, and it's limited by your personal knowledge and imagination. (Not that I'm suggesting that your personal knowledge and imagination are limited.) So let's think about what we can do with *data*.

Recommending What’s Popular

One easy approach is to simply recommend what’s popular:

```
popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests).most_common()
```

which looks like:

```
[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 # ...
]
```

Having computed this, we can just suggest to a user the most popular interests that he’s not already interested in:

```
def most_popular_new_interests(user_interests, max_results=5):
    suggestions = [(interest, frequency)
                   for interest, frequency in popular_interests
                   if interest not in user_interests]
    return suggestions[:max_results]
```

So, if you are user 1, with interests:

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

then we’d recommend you:

```
most_popular_new_interests(users_interests[1], 5)

# [('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics', 3)]
```

If you are user 3, who’s already interested in many of those things, you’d instead get:

```
[('Java', 3),
 ('HBase', 3),
 ('Big Data', 3),
 ('neural networks', 2),
 ('Hadoop', 2)]
```

Of course, “lots of people are interested in Python so maybe you should be too” is not the most compelling sales pitch. If someone is brand new to our site and we don’t know anything about them, that’s possibly the best we can do. Let’s see how we can do better by basing each user’s recommendations on her interests.

User-Based Collaborative Filtering

One way of taking a user’s interests into account is to look for users who are somehow *similar* to him, and then suggest the things that those users are interested in.

In order to do that, we’ll need a way to measure how similar two users are. Here we’ll use a metric called *cosine similarity*. Given two vectors, `v` and `w`, it’s defined as:

```
def cosine_similarity(v, w):
    return dot(v, w) / math.sqrt(dot(v, v) * dot(w, w))
```

It measures the “angle” between `v` and `w`. If `v` and `w` point in the same direction, then the numerator and denominator are equal, and their cosine similarity equals 1. If `v` and `w` point in opposite directions, then their cosine similarity equals -1. And if `v` is 0 whenever `w` is not (and vice versa) then `dot(v, w)` is 0 and so the cosine similarity will be 0.

We’ll apply this to vectors of 0s and 1s, each vector `v` representing one user’s interests. `v[i]` will be 1 if the user is specified the *i*th interest, 0 otherwise. Accordingly, “similar users” will mean “users whose interest vectors most nearly point in the same direction.” Users with identical interests will have similarity 1. Users with no identical interests will have similarity 0. Otherwise the similarity will fall in between, with numbers closer to 1 indicating “very similar” and numbers closer to 0 indicating “not very similar.”

A good place to start is collecting the known interests and (implicitly) assigning indices to them. We can do this by using a set comprehension to find the unique interests, putting them in a list, and then sorting them. The first interest in the resulting list will be interest 0, and so on:

```
unique_interests = sorted(list({ interest
                                 for user_interests in users_interests
                                 for interest in user_interests }))
```

This gives us a list that starts:

```
['Big Data',
 'C++',
 'Cassandra',
 'HBase',
 'Hadoop',
 'Haskell',
 # ...
]
```

Next we want to produce an “interest” vector of 0s and 1s for each user. We just need to iterate over the `unique_interests` list, substituting a 1 if the user has each interest, a 0 if not:

```
def make_user_interest_vector(user_interests):
    """given a list of interests, produce a vector whose ith element is 1
    if unique_interests[i] is in the list, 0 otherwise"""
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

after which, we can create a matrix of user interests simply by `map`-ping this function against the list of lists of interests:

```
user_interest_matrix = map(make_user_interest_vector, users_interests)
```

Now `user_interest_matrix[i][j]` equals 1 if user `i` specified interest `j`, 0 otherwise.

Because we have a small data set, it's no problem to compute the pairwise similarities between all of our users:

```
user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
                      for interest_vector_j in user_interest_matrix]
                     for interest_vector_i in user_interest_matrix]
```

after which, `user_similarities[i][j]` gives us the similarity between users *i* and *j*.

For instance, `user_similarities[0][9]` is 0.57, as those two users share interests in Hadoop, Java, and Big Data. On the other hand, `user_similarities[0][8]` is only 0.19, as users 0 and 8 share only one interest, Big Data.

In particular, `user_similarities[i]` is the vector of user *i*'s similarities to every other user. We can use this to write a function that finds the most similar users to a given user. We'll make sure not to include the user herself, nor any users with zero similarity. And we'll sort the results from most similar to least similar:

```
def most_similar_users_to(user_id):
    pairs = [(other_user_id, similarity)                      # find other
             for other_user_id, similarity in                 # users with
                enumerate(user_similarities[user_id])         # nonzero
             if user_id != other_user_id and similarity > 0]  # similarity

    return sorted(pairs,                                      # sort them
                  key=lambda (_, similarity): similarity,     # most similar
                  reverse=True)                               # first
```

For instance, if we call `most_similar_users_to(0)` we get:

```
[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]
```

How do we use this to suggest new interests to a user? For each interest, we can just add up the user-similarities of the other users interested in it:

```
def user_based_suggestions(user_id, include_current_interests=False):
    # sum up the similarities
    suggestions = defaultdict(float)
    for other_user_id, similarity in most_similar_users_to(user_id):
        for interest in users_interests[other_user_id]:
            suggestions[interest] += similarity

    # convert them to a sorted list
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, weight): weight,
                         reverse=True)

    # and (maybe) exclude already-interests
    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

If we call `user_based_suggestions(0)`, the first several suggested interests are:

```
[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
 ('Postgres', 0.50709255283711),
 ('NoSQL', 0.3380617018914066),
 ('neural networks', 0.1889822365046136),
 ('deep learning', 0.1889822365046136),
 ('artificial intelligence', 0.1889822365046136),
 #...
]
```

These seem like pretty decent suggestions for someone whose stated interests are “Big Data” and database-related. (The weights aren’t intrinsically meaningful; we just use them for ordering.)

This approach doesn’t work as well when the number of items gets very large. Recall the curse of dimensionality from Chapter 12 — in large-dimensional vector spaces most vectors are very far apart (and therefore point in very different directions). That is, when there are a large number of interests the “most similar users” to a given user might not be similar at all.

Imagine a site like Amazon.com, from which I’ve bought thousands of items over the last couple of decades. You could attempt to identify similar users to me based on buying patterns, but most likely in all the world there’s no one whose purchase history looks even remotely like mine. Whoever my “most similar” shopper is, he’s probably not similar to me at all, and his purchases would almost certainly make for lousy recommendations.

Item-Based Collaborative Filtering

An alternative approach is to compute similarities between interests directly. We can then generate suggestions for each user by aggregating interests that are similar to her current interests.

To start with, we’ll want to *transpose* our user-interest matrix so that rows correspond to interests and columns correspond to users:

```
interest_user_matrix = [[user_interest_vector[j]
                         for user_interest_vector in user_interest_matrix]
                        for j, _ in enumerate(unique_interests)]
```

What does this look like? Row `j` of `interest_user_matrix` is column `j` of `user_interest_matrix`. That is, it has 1 for each user with that interest and 0 for each user without that interest.

For example, `unique_interests[0]` is Big Data, and so `interest_user_matrix[0]` is:

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

because users 0, 8, and 9 indicated interest in Big Data.

We can now use cosine similarity again. If precisely the same users are interested in two topics, their similarity will be 1. If no two users are interested in both topics, their similarity will be 0:

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                          for user_vector_j in interest_user_matrix]
                         for user_vector_i in interest_user_matrix]
```

For example, we can find the interests most similar to Big Data (interest 0) using:

```
def most_similar_interests_to(interest_id):
    similarities = interest_similarities[interest_id]
    pairs = [(unique_interests[other_interest_id], similarity)
             for other_interest_id, similarity in enumerate(similarities)
             if interest_id != other_interest_id and similarity > 0]
    return sorted(pairs,
                  key=lambda (_, similarity): similarity,
                  reverse=True)
```

which suggests the following similar interests:

```
[('Hadoop', 0.8164965809277261),
 ('Java', 0.6666666666666666),
 ('MapReduce', 0.5773502691896258),
 ('Spark', 0.5773502691896258),
 ('Storm', 0.5773502691896258),
 ('Cassandra', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('neural networks', 0.4082482904638631),
 ('HBase', 0.3333333333333333)]
```

Now we can create recommendations for a user by summing up the similarities of the interests similar to his:

```
def item_based_suggestions(user_id, include_current_interests=False):
    # add up the similar interests
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_matrix[user_id]
    for interest_id, is_interested in enumerate(user_interest_vector):
        if is_interested == 1:
            similar_interests = most_similar_interests_to(interest_id)
            for interest, similarity in similar_interests:
                suggestions[interest] += similarity

    # sort them by weight
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, similarity): similarity,
                         reverse=True)

    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

For user 0, this generates the following (seemingly reasonable) recommendations:

```
[('MapReduce', 1.861807319565799),
 ('Postgres', 1.3164965809277263),
 ('MongoDB', 1.3164965809277263),
 ('NoSQL', 1.2844570503761732),
 ('programming languages', 0.5773502691896258),
 ('MySQL', 0.5773502691896258),
 ('Haskell', 0.5773502691896258),
 ('databases', 0.5773502691896258),
 ('neural networks', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('C++', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('Python', 0.2886751345948129),
 ('R', 0.2886751345948129)]
```