CHAPTER 4

Searching and Ranking

This chapter covers full-text search engines, which allow people to search a large set of documents for a list of words, and which rank results according to how relevant the documents are to those words. Algorithms for full-text searches are among the most important collective intelligence algorithms, and many fortunes have been made by new ideas in this field. It is widely believed that Google's rapid rise from an academic project to the world's most popular search engine was based largely on the PageRank algorithm, a variation that you'll learn about in this chapter.

Information retrieval is a huge field with a long history. This chapter will only be able to cover a few key concepts, but we'll go through the construction of a search engine that will index a set of documents and leave you with ideas on how to improve things further. Although the focus will be on algorithms for searching and ranking rather than on the infrastructure requirements for indexing large portions of the Web, the search engine you build should have no problem with collections of up to 100,000 pages. Throughout this chapter, you'll learn all the necessary steps to crawl, index, and search a set of pages, and even rank their results in many different ways.

What's in a Search Engine?

The first step in creating a search engine is to develop a way to collect the documents. In some cases, this will involve *crawling* (starting with a small set of documents and following links to others) and in other cases it will begin with a fixed collection of documents, perhaps from a corporate intranet.

After you collect the documents, they need to be indexed. This usually involves creating a big table of the documents and the locations of all the different words. Depending on the particular application, the documents themselves do not necessarily have to be stored in a database; the index simply has to store a reference (such as a file system path or URL) to their locations.

The final step is, of course, returning a ranked list of documents from a query. Retrieving every document with a given set of words is fairly straightforward once you have an index, but the real magic is in how the results are sorted. A huge number of metrics can be generated, and there is no shortage of ways you can tweak them to change the sort order. Just learning all the different metrics might make you wish that the big search engines would let you control more of them ("Why can't I tell Google that my words must be close together?"). This chapter will look at several metrics based on the content of the page, such as word frequency, and then cover metrics based on information external to the content of the page, such as the PageRank algorithm, which looks at how other pages link to the page in question.

Finally, you'll build a *neural network* for ranking queries. The neural network will learn to associate searches with results based on what links people click on after they get a list of search results. The neural network will use this information to change the ordering of the results to better reflect what people have clicked on in the past.

To work through the examples in this chapter, you'll need to create a Python module called *searchengine*, which has two classes: one for crawling and creating the database, and the other for doing full-text searches by querying the database. The examples will use *SQLite*, but they can easily be adapted to work with a traditional client-server database.

To start, create a new file called *searchengine.py* and add the following `crawler` class and method signatures, which you'll be filling in throughout this chapter:

```
class crawler:
  # Initialize the crawler with the name of database
  def __init__(self,dbname):
    pass

  def __del__(self):
    pass
  def dbcommit(self):
    pass

  # Auxilliary function for getting an entry id and adding
  # it if it's not present
  def getentryid(self,table,field,value,createnew=True):
    return None

  # Index an individual page
  def addtoindex(self,url,soup):
    print 'Indexing %s' % url

  # Extract the text from an HTML page (no tags)
  def gettextonly(self,soup):
    return None

  # Separate the words by any non-whitespace character
  def separatewords(self,text):
    return None
```

```
  # Return true if this url is already indexed
  def isindexed(self,url):
    return False

  # Add a link between two pages
  def addlinkref(self,urlFrom,urlTo,linkText):
    pass

  # Starting with a list of pages, do a breadth
  # first search to the given depth, indexing pages
  # as we go
  def crawl(self,pages,depth=2):
    pass

  # Create the database tables
  def createindextables(self):
    pass
```

A Simple Crawler

I'll assume for now that you don't have a big collection of HTML documents sitting on your hard drive waiting to be indexed, so I'll show you how to build a simple crawler. It will be seeded with a small set of pages to index and will then follow any links on that page to find other pages, whose links it will also follow. This process is called crawling or *spidering*.

To do this, your code will have to download the pages, pass them to the indexer (which you'll build in the next section), and then parse the pages to find all the links to the pages that have to be crawled next. Fortunately, there are a couple of libraries that can help with this process.

For the examples in this chapter, I have set up a copy of several thousand files from Wikipedia, which will remain static at *http://kiwitobes.com/wiki*.

You're free to run the crawler on any set of pages you like, but you can use this site if you want to compare your results to those in this chapter.

Using urllib2

urllib2 is a library bundled with Python that makes it easy to download pages—all you have to do is supply the URL. You'll use it in this section to download the pages that will be indexed. To see it in action, start up your Python interpreter and try this:

```
>> import urllib2
>> c=urllib2.urlopen('http://kiwitobes.com/wiki/Programming_language.html')
>> contents=c.read( )
>> print contents[0:50]
'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Trans'
```

All you have to do to store a page's HTML code into a string is create a connection and read its contents.

Crawler Code

The crawler will use the Beautiful Soup API that was introduced in Chapter 3, an excellent library that builds a structured representation of web pages. It is very tolerant of web pages with broken HTML, which is useful when constructing a crawler because you never know what pages you might come across. For more information on downloading and installing Beautiful Soup, see Appendix A.

Using urllib2 and Beautiful Soup you can build a crawler that will take a list of URLs to index and crawl their links to find other pages to index. First, add these `import` statements to the top of *searchengine.py*:

```
import urllib2
from BeautifulSoup import *
from urlparse import urljoin

# Create a list of words to ignore
ignorewords=set(['the','of','to','and','a','in','is','it'])
```

Now you can fill in the code for the crawler function. It won't actually save anything it crawls yet, but it will print the URLs as it goes so you can see that it's working. You need to put this at the end of the file (so it's part of the `crawler` class):

```
  def crawl(self,pages,depth=2):
    for i in range(depth):
      newpages=set( )
      for page in pages:
        try:
          c=urllib2.urlopen(page)
        except:
          print "Could not open %s" % page
          continue
        soup=BeautifulSoup(c.read( ))
        self.addtoindex(page,soup)

        links=soup('a')
        for link in links:
          if ('href' in dict(link.attrs)):
            url=urljoin(page,link['href'])
            if url.find("'")!=-1: continue
            url=url.split('#')[0]  # remove location portion
            if url[0:4]=='http' and not self.isindexed(url):
              newpages.add(url)
            linkText=self.gettextonly(link)
            self.addlinkref(page,url,linkText)

        self.dbcommit( )

      pages=newpages
```

This function loops through the list of pages, calling `addtoindex` on each one (right now this does nothing except print the URL, but you'll fill it in the next section). It

then uses Beautiful Soup to get all the links on that page and adds their URLs to a set called `newpages`. At the end of the loop, `newpages` becomes `pages`, and the process repeats.

This function can be defined recursively so that each link calls the function again, but doing a breadth-first search allows for easier modification of the code later, either to keep crawling continuously or to save a list of unindexed pages for later crawling. It also avoids the risk of overflowing the stack.

You can test this function in the Python interpreter (there's no need to let it finish, so press Ctrl-C when you get bored):

```
>> import searchengine
>> pagelist=['http://kiwitobes.com/wiki/Perl.html']
>> crawler=searchengine.crawler('')
>> crawler.crawl(pagelist)
Indexing http://kiwitobes.com/wiki/Perl.html
Could not open http://kiwitobes.com/wiki/Module_%28programming%29.html
Indexing http://kiwitobes.com/wiki/Open_Directory_Project.html
Indexing http://kiwitobes.com/wiki/Common_Gateway_Interface.html
```

You may notice that some pages are repeated. There is a placeholder in the code for another function, `isindexed`, which will determine if a page has been indexed recently before adding it to `newpages`. This will let you run this function on any list of URLs at any time without worrying about doing unnecessary work.

Building the Index

The next step is to set up the database for the full-text index. As I mentioned earlier, the index is a list of all the different words, along with the documents in which they appear and their locations in the documents. In this example, you'll be looking at the actual text on the page and ignoring nontext elements. You'll also be indexing individual words with all the punctuation characters removed. The method for separating words is not perfect, but it will suffice for building a basic search engine.

Because covering different database software or setting up a database server is outside the scope of this book, this chapter will show you how to store the index using SQLite. SQLite is an embedded database that is very easy to set up and stores a whole database in one file. SQLite uses SQL for queries, so it shouldn't be too difficult to convert the sample code to use a different database. The Python implementation is called pysqlite, and you can download it from *http://initd.org/tracker/pysqlite*.

There is a Windows installer as well as instructions for installing it on other operating systems. Appendix A contains more information on getting and installing pysqlite.

Once you have SQLite installed, add this line to the start of *searchengine.py*:

```
from pysqlite2 import dbapi2 as sqlite
```

You'll also need to change the `__init__`, `__del__`, and `dbcommit` methods to open and close the database:

```
def __init__(self,dbname):
  self.con=sqlite.connect(dbname)

def __del__(self):
  self.con.close( )

def dbcommit(self):
  self.con.commit( )
```

Setting Up the Schema

Don't run the code just yet—you still need to prepare the database. The schema for the basic index is five tables. The first table (`urllist`) is the list of URLs that have been indexed. The second table (`wordlist`) is the list of words, and the third table (`wordlocation`) is a list of the locations of words in the documents. The remaining two tables specify links between documents. The `link` table stores two URL IDs, indicating a link from one table to another, and `linkwords` uses the `wordid` and `linkid` columns to store which words are actually used in that link. The schema is shown in Figure 4-1.

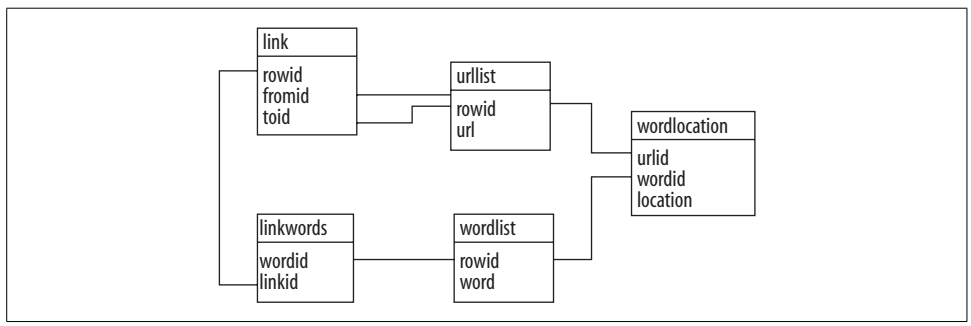

Figure 4-1. Schema for the search engine

All tables in SQLite have a field called `rowid` by default, so there's no need to explicitly specify an ID for these tables. To create a function for adding all the tables, add this code to the end of *searchengine.py* so that it's part of the `crawler` class:

```
def createindextables(self):
  self.con.execute('create table urllist(url)')
  self.con.execute('create table wordlist(word)')
  self.con.execute('create table wordlocation(urlid,wordid,location)')
  self.con.execute('create table link(fromid integer,toid integer)')
  self.con.execute('create table linkwords(wordid,linkid)')
  self.con.execute('create index wordidx on wordlist(word)')
  self.con.execute('create index urlidx on urllist(url)')
  self.con.execute('create index wordurlidx on wordlocation(wordid)')
```

```
    self.con.execute('create index urltoidx on link(toid)')
    self.con.execute('create index urlfromidx on link(fromid)')
    self.dbcommit( )
```

This function will create the schema for all the tables that you will be using, along with some indices to speed up searching. These indices are important, since the dataset can potentially get very large. Enter these commands in your Python session to create a database called *searchindex.db*:

```
>> reload(searchengine)
>> crawler=searchengine.crawler('searchindex.db')
>> crawler.createindextables( )
```

Later you'll be adding an additional table to the schema for a scoring metric based on counting inbound links.

Finding the Words on a Page

The files that you're downloading from the Web are HTML and thus contain a lot of tags, properties, and other information that doesn't belong in the index. The first step is to extract all the parts of the page that are text. You can do this by searching the soup for text nodes and collecting all their content. Add this code to your `gettextonly` function:

```
def gettextonly(self,soup):
  v=soup.string
  if v==None:
    c=soup.contents
    resulttext=''
    for t in c:
      subtext=self.gettextonly(t)
      resulttext+=subtext+'\n'
    return resulttext
  else:
    return v.strip( )
```

The function returns a long string containing all the text on the page. It does this by recursively traversing down the HTML document object model, looking for text nodes. Text that was in separate sections is separated into different paragraphs. It's important to preserve the order of the sections for some of the metrics you'll be calculating later.

Next is the `separatewords` function, which splits a string into a list of separate words so that they can be added to the index. It's not as easy as you might think to do this perfectly, and there has been a lot of research into improving the technique. However, for these examples it will suffice to consider anything that isn't a letter or a number to be a separator. You can do this using a regular expression. Replace the definition of `separatewords` with the following:

```
def separatewords(self,text):
  splitter=re.compile('\\W*')
  return [s.lower( ) for s in splitter.split(text) if s!='']
```

Because this function considers anything nonalphanumeric to be a separator, it will have no problem extracting English words, but it won't properly handle terms like "C++" (no trouble searching for "python," though). You can experiment with the regular expression to make it work better for different kinds of searches.

Another possibility is to remove suffixes from the words using a *stemming* algorithm. These algorithms attempt to convert the words to their stems. For example, the word "indexing" becomes "index" so that people searching for the word "index" are also shown documents containing the word "indexing." To do this, stem the words while crawling documents and also stem the words in the search query. A full discussion of stemming is outside the scope of this chapter, but you can find a Python implementation of the well-known *Porter Stemmer* at *http://www.tartarus.org/~martin/PorterStemmer/index.html*.

Adding to the Index

You're ready to fill in the code for the `addtoindex` method. This method will call the two functions that were defined in the previous section to get a list of words on the page. Then it will add the page and all the words to the index, and will create links between them with their locations in the document. For this example, the location will be the index within the list of words.

Here is the code for `addtoindex`:

```
def addtoindex(self,url,soup):
  if self.isindexed(url): return
  print 'Indexing '+url

  # Get the individual words
  text=self.gettextonly(soup)
  words=self.separatewords(text)

  # Get the URL id
  urlid=self.getentryid('urllist','url',url)

  # Link each word to this url
  for i in range(len(words)):
    word=words[i]
    if word in ignorewords: continue
    wordid=self.getentryid('wordlist','word',word)
    self.con.execute("insert into wordlocation(urlid,wordid,location) \
      values (%d,%d,%d)" % (urlid,wordid,i))
```

You'll also need this to update the helper function `getentryid`. All this does is return the ID of an entry. If the entry doesn't exist, it is created and the ID is returned:

```
def getentryid(self,table,field,value,createnew=True):
  cur=self.con.execute(
  "select rowid from %s where %s='%s'" % (table,field,value))
  res=cur.fetchone( )
  if res==None:
    cur=self.con.execute(
    "insert into %s (%s) values ('%s')" % (table,field,value))
    return cur.lastrowid
  else:
    return res[0]
```

Finally, you'll need to fill in the code for `isindexed`, which determines whether the page is already in the database, and if so, whether there are any words associated with it:

```
def isindexed(self,url):
  u=self.con.execute \
    ("select rowid from urllist where url='%s'" % url).fetchone( )
  if u!=None:
    # Check if it has actually been crawled
    v=self.con.execute(
    'select * from wordlocation where urlid=%d' % u[0]).fetchone( )
    if v!=None: return True
  return False
```

Now you can rerun the crawler and have it actually index the pages as it goes. You can do this in your interactive session:

```
>> reload(searchengine)
>> crawler=searchengine.crawler('searchindex.db')
>> pages= \
.. ['http://kiwitobes.com/wiki/Categorical_list_of_programming_languages.html']
>> crawler.crawl(pages)
```

The crawler will probably take a long time to run. Instead of waiting for it to finish, I recommend that you download a preloaded copy of *searchindex.db* from *http://kiwitobes.com/db/searchindex.db* and save it in the directory with your Python code.

If you'd like to make sure that the crawl worked properly, you can try checking the entries for a word by querying the database:

```
>> [row for row in crawler.con.execute(
.. 'select rowid from wordlocation where wordid=1')]
[(1,), (46,), (330,), (232,), (406,), (271,), (192,),...
```

The list that is returned is the list of all the URL IDs containing "word," which means that you've successfully run a full-text search. This is a great start, but it will only work with one word at a time, and will just return the documents in the order in which they were loaded. The next section will show you how to expand this functionality by doing these searches with multiple words in the query.

Querying

You now have a working crawler and a big collection of documents indexed, and you're ready to set up the search part of the search engine. First, create a new class in *searchengine.py* that you'll use for searching:

```
class searcher:
  def __init__(self,dbname):
    self.con=sqlite.connect(dbname)

  def __del__(self):
    self.con.close( )
```

The `wordlocation` table gives an easy way to link words to tables, so it is quite easy to see which pages contain a single word. However, a search engine is pretty limited unless it allows multiple-word searches. To do this, you'll need a query function that takes a query string, splits it into separate words, and constructs a SQL query to find only those URLs containing all the different words. Add this function to the definition for the `searcher` class:

```
  def getmatchrows(self,q):
    # Strings to build the query
    fieldlist='w0.urlid'
    tablelist=''
    clauselist=''
    wordids=[]

    # Split the words by spaces
    words=q.split(' ')
    tablenumber=0

    for word in words:
      # Get the word ID
      wordrow=self.con.execute(
        "select rowid from wordlist where word='%s'" % word).fetchone( )
      if wordrow!=None:
        wordid=wordrow[0]
        wordids.append(wordid)
        if tablenumber>0:
          tablelist+=','
          clauselist+=' and '
          clauselist+='w%d.urlid=w%d.urlid and ' % (tablenumber-1,tablenumber)
        fieldlist+=',w%d.location' % tablenumber
        tablelist+='wordlocation w%d' % tablenumber
        clauselist+='w%d.wordid=%d' % (tablenumber,wordid)
        tablenumber+=1

    # Create the query from the separate parts
    fullquery='select %s from %s where %s' % (fieldlist,tablelist,clauselist)
    cur=self.con.execute(fullquery)
    rows=[row for row in cur]

    return rows,wordids
```

This function looks a bit complicated, but it's just creating a reference to the `wordlocation` table for each word in the list and joining them all on their URL IDs (Figure 4-2).

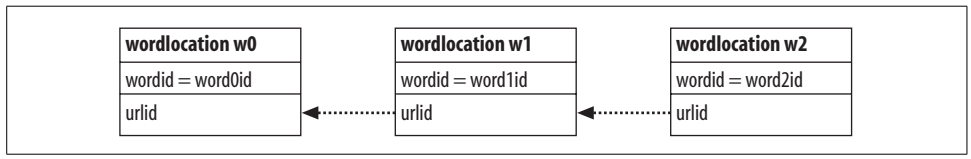

Figure 4-2. Table joins for getmatchrows

So a query for two words with the IDs 10 and 17 becomes:

```
select w0.urlid,w0.location,w1.location
from  wordlocation w0,wordlocation w1
where w0.urlid=w1.urlid
and   w0.wordid=10
and   w1.wordid=17
```

Try calling this function with your first multiple-word search:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.getmatchrows('functional programming')
([(1, 327, 23), (1, 327, 162), (1, 327, 243), (1, 327, 261),
  (1, 327, 269), (1, 327, 436), (1, 327, 953),..
```

You'll notice that each URL ID is returned many times with different combinations of word locations. The next few sections will cover some ways to rank the results. *Content-based ranking* uses several possible metrics with just the content of the page to determine the relevance of the query. *Inbound-link ranking* uses the link structure of the site to determine what's important. We will also explore a way to look at what people actually click on when they search in order to improve the rankings over time.

Content-Based Ranking

So far you've managed to retrieve pages that match the queries, but the order in which they are returned is simply the order in which they were crawled. In a large set of pages, you would be stuck sifting through a lot of irrelevant content for any mention of each of the query terms in order to find the pages that are really related to your search. To address this issue, you need ways to give pages a *score* for a given query, as well as the ability to return them with the highest scoring results first.

This section will look at several ways to calculate a score based only on the query and the content of the page. These scoring metrics include:

Word frequency
: The number of times the words in the query appear in the document can help determine how relevant the document is.

Document location
: The main subject of a document will probably appear near the beginning of the document.

Word distance
: If there are multiple words in the query, they should appear close together in the document.

The earliest search engines often worked with only these types of metrics and were able to give usable results. Later sections will cover ways to improve results with information external to the page, such as the number and quality of incoming links.

First, you'll need a new method that will take a query, get the rows, put them in a dictionary, and display them in a formatted list. Add these functions to your `searcher` class:

```
def getscoredlist(self,rows,wordids):
  totalscores=dict([(row[0],0) for row in rows])

  # This is where you'll later put the scoring functions
  weights=[]

  for (weight,scores) in weights:
    for url in totalscores:
      totalscores[url]+=weight*scores[url]

  return totalscores

def geturlname(self,id):
  return self.con.execute(
  "select url from urllist where rowid=%d" % id).fetchone( )[0]

def query(self,q):
  rows,wordids=self.getmatchrows(q)
  scores=self.getscoredlist(rows,wordids)
  rankedscores=sorted([(score,url) for (url,score) in scores.items( )],reverse=1)
  for (score,urlid) in rankedscores[0:10]:
    print '%f\t%s' % (score,self.geturlname(urlid))
```

Right now the query method doesn't apply any scoring to the results, but it does display the URLs along with a placeholder for their scores:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.query('functional programming')
0.000000 http://kiwitobes.com/wiki/XSLT.html
0.000000 http://kiwitobes.com/wiki/XQuery.html
0.000000 http://kiwitobes.com/wiki/Unified_Modeling_Language.html
...
```

The important function here is `getscoredlist`, which you'll be filling in throughout this section. As you add scoring functions, you can add calls to the `weights` list (the line in bold) and start to get some real scores.

Normalization Function

All the scoring methods introduced here return dictionaries of the URL IDs and a numerical score. To complicate things, sometimes a larger score is better and sometimes a smaller score is better. In order to compare the results from different methods, you need a way to *normalize* them; that is, to get them all within the same range and direction.

The normalization function will take a dictionary of IDs and scores and return a new dictionary with the same IDs, but with scores between 0 and 1. Each score is scaled according to how close it is to the best result, which will always have a score of 1. All you have to do is pass the function a list of scores and indicate whether a lower or higher score is better:

```
def normalizescores(self,scores,smallIsBetter=0):
  vsmall=0.00001 # Avoid division by zero errors
  if smallIsBetter:
    minscore=min(scores.values( ))
    return dict([(u,float(minscore)/max(vsmall,l)) for (u,l) \
      in scores.items( )])
  else:
    maxscore=max(scores.values( ))
    if maxscore==0: maxscore=vsmall
    return dict([(u,float(c)/maxscore) for (u,c) in scores.items( )])
```

Each of the scoring functions calls this function to normalize its results and return a value between 0 and 1.

Word Frequency

The word frequency metric scores a page based on how many times the words in the query appear on that page. If I search for "python," I'd rather get a page about Python (or pythons) with many mentions of the word, and not a page about a musician who happens to mention near the end that he has a pet python.

The word frequency function looks like this. You can add it to your `searcher` class:

```
def frequencyscore(self,rows):
  counts=dict([(row[0],0) for row in rows])
  for row in rows: counts[row[0]]+=1
  return self.normalizescores(counts)
```

This function creates a dictionary with an entry for every unique URL ID in *rows*, and counts how many times each item appears. It then normalizes the scores (bigger is better, in this case) and returns the result.

To activate frequency scoring in your results, change the `weights` line in `getscoredlist` to read:

```
weights=[(1.0,self.frequencyscore(rows))]
```

Now you can try another search and see how well this works as a scoring metric:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.query('functional programming')
1.000000 http://kiwitobes.com/wiki/Functional_programming.html
0.262476 http://kiwitobes.com/wiki/Categorical_list_of_programming_languages.html
0.062310 http://kiwitobes.com/wiki/Programming_language.html
0.043976 http://kiwitobes.com/wiki/Lisp_programming_language.html
0.036394 http://kiwitobes.com/wiki/Programming_paradigm.html
...
```

This returns the page on "Functional programming" in first place, followed by several other relevant pages. Notice that "Functional programming" scored four times better than the result directly below it. Most search engines don't report scores to end users, but these scores can be very useful for some applications. For instance, you might want to take the user directly to the top result if it exceeds a certain threshold, or display results in a font size proportional to the relevance of the result.

Document Location

Another simple metric for determining a page's relevance to a query is the search term's location in the page. Usually, if a page is relevant to the search term, it will appear closer to the top of the page, perhaps even in the title. To take advantage of this, the search engine can score results higher if the query term appears early in the document. Fortunately for us, when the pages were indexed earlier, the locations of the words were recorded, and the title of the page is first in the list.

Add this method to `searcher`:

```
def locationscore(self,rows):
  locations=dict([(row[0],1000000) for row in rows])
  for row in rows:
    loc=sum(row[1:])
    if loc<locations[row[0]]: locations[row[0]]=loc

  return self.normalizescores(locations,smallIsBetter=1)
```

Remember that the first item in each `row` element is the URL ID, followed by the locations of all the different search terms. Each ID can appear multiple times, once for every combination of locations. For each row, the method sums the locations of all the words and determines how this result compares to the best result for that URL so far. It then passes the final results to the `normalize` function. Note that `smallIsBetter` means that the URL with the lowest location sum gets a score of 1.0.

To see what the results look like using only the location score, change the `weights` line to this:

```
weights=[(1.0,self.locationscore(rows))]
```

Now try the query again in your interpreter:

```
>> reload(searchengine)
>> e=searchengine.searcher('searchindex.db')
>> e.query('functional programming')
```

You'll notice that "Functional programming" is still the winner, but the other top results are now examples of functional programming languages. The previous search returned results in which the words were mentioned several times, but these tended to be discussions about programming languages in general. With this search, however, the presence of the words in the opening sentence (e.g., "Haskell is a standardized pure functional programming language") gave them a much higher score.

It's important to realize that neither one of the metrics shown so far is better in every case. Both of these lists are valid depending on the searcher's intent, and different combinations of weights are required to give the best results for a particular set of documents and applications. You can try experimenting with different weights for the two metrics by changing your `weights` line to something like this:

```
weights=[(1.0,self.frequencyscore(rows)),
         (1.5,self.locationscore(rows))]
```

Experiment with different weights and queries and see how your results are affected.

Location is a more difficult metric to cheat than word frequency, since page authors can only put one word first in a document and repeating it doesn't make any difference to the results.

Word Distance

When a query contains multiple words, it is often useful to seek results in which the words in the query are close to each other in the page. Most of the time, when people make multiple-word queries, they are interested in a page that conceptually relates the different words. This is a little looser than the quoted-phrase searches supported by most search engines where the words must appear in the correct order with no additional words—in this case, the metric will tolerate a different order and additional words between the query words.

The `distancescore` function looks pretty similar to `locationscore`:

```
def distancescore(self,rows):
  # If there's only one word, everyone wins!
  if len(rows[0])<=2: return dict([(row[0],1.0) for row in rows])

  # Initialize the dictionary with large values
  mindistance=dict([(row[0],1000000) for row in rows])

  for row in rows:
    dist=sum([abs(row[i]-row[i-1]) for i in range(2,len(row))])
    if dist<mindistance[row[0]]: mindistance[row[0]]=dist
  return self.normalizescores(mindistance,smallIsBetter=1)
```

The main difference here is that when the function loops through the locations (on the line shown in bold), it takes the difference between each location and the previous location. Since every combination of distances is returned by the query, it is guaranteed to find the smallest total distance.

You can try the word distance metric by itself if you like, but it really works better when combined with other metrics. Try adding `distancescore` to the `weights` list and changing the numbers to see how it affects the results of different queries.

Using Inbound Links

The scoring metrics discussed so far have all been based on the content of the page. Although many search engines still work this way, the results can often be improved by considering information that others have provided about the page, specifically, who has linked to the page and what they have said about it. This is particularly useful when indexing pages of dubious value or pages that might have been created by spammers, as these are less likely to be linked than pages with real content.

The crawler that you built at the beginning of the chapter already captures all the important information about the links, so there's no need to change it. The `links` table has the URL IDs for the source and target of every link that it has encountered, and the `linkwords` table connects the words with the links.

Simple Count

The easiest thing to do with inbound links is to count them on each page and use the total number of links as a metric for the page. Academic papers are often rated in this way, with their importance tied to the number of other papers that reference them. The scoring function below creates a dictionary of counts by querying the link table for every unique URL ID in `rows`, and then it returns the normalized scores:

```
def inboundlinkscore(self,rows):
  uniqueurls=set([row[0] for row in rows])
  inboundcount=dict([(u,self.con.execute( \
    'select count(*) from link where toid=%d' % u).fetchone()[0]) \
      for u in uniqueurls])
  return self.normalizescores(inboundcount)
```

Obviously, using this metric by itself will simply return all the pages containing the search terms, ranked solely on how many inbound links they have. In the dataset, "Programming language" has many more inbound links than "Python," but you'd rather see "Python" first in the results if that's what you searched for. To combine relevance with ranking, you need to use the inbound-links metric in combination with one of the metrics shown earlier.

This algorithm also weights every inbound link equally, which, while nice and egalitarian, is open to manipulation because someone can easily set up several sites pointing to a page whose score they want to increase. It's also possible that people are more interested in results that have attracted the attention of very popular sites. Next, you'll see how to make links from popular pages worth more in calculating rankings.

The PageRank Algorithm

The PageRank algorithm was invented by the founders of Google, and variations on the idea are now used by all the large search engines. This algorithm assigns every page a score that indicates how important that page is. The importance of the page is calculated from the importance of all the other pages that link to it and from the number of links each of the other pages has.

In theory, PageRank (named after one of its inventors, Larry Page) calculates the probability that someone randomly clicking on links will arrive at a certain page. The more inbound links the page has from other popular pages, the more likely it is that someone will end up there purely by chance. Of course, if the user keeps clicking forever, they'll eventually reach every page, but most people stop surfing after a while. To capture this, PageRank also uses a *damping factor* of 0.85, indicating that there is an 85 percent chance that a user will continue clicking on links at each page.

Figure 4-3 shows an example set of pages and links.

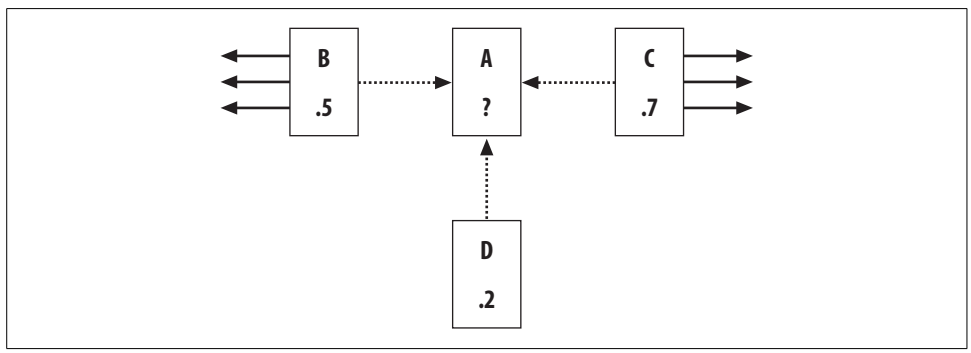

Figure 4-3. Calculating the PageRank of A

Pages B, C, and D all link to A, and they already have their PageRanks calculated. B also links to three other pages and C links to four other pages. D only links to A. To get A's PageRank, take the PageRank (PR) of each of the pages that links to A divided by the total number of links on that page, then multiply this by a damping factor of 0.85, and add a minimum value of 0.15. The calculation for `PR(A)` is:

```
PR(A) = 0.15 + 0.85 * ( PR(B)/links(B) + PR(C)/links(C) + PR(D)/links(D) )
      = 0.15 + 0.85 * ( 0.5/4 + 0.7/5 + 0.2/1 )
      = 0.15 + 0.85 * ( 0.125 + 0.14 + 0.2)
      = 0.15 + 0.85 * 0.465
      = 0.54525
```

You'll notice that D actually contributes more to A's PageRank than either B or C does, even though it has a lower PageRank of its own, because it links exclusively to A and is able to contribute its entire score.

Pretty easy, right? Well, there's a small catch—in this example, all the pages linking to A already had PageRanks. You can't calculate a page's score until you know the scores of all the pages that link there, and you can't calculate their scores without doing the same for all the pages that link to *them*. How is it possible to calculate PageRanks for a whole set of pages that don't already have PageRanks?

The solution is to set all the PageRanks to an initial arbitrary value (the code will use 1.0, but the actual value doesn't make any difference), and repeat the calculation over several iterations. After each iteration, the PageRank for each page gets closer to its true PageRank value. The number of iterations needed varies with the number of pages, but in the small set you're working with, 20 should be sufficient.

Because the PageRank is time-consuming to calculate and stays the same no matter what the query is, you'll be creating a function that precomputes the PageRank for every URL and stores it in a table. This function will recalculate all the PageRanks every time it is run. Add this function to the `crawler` class:

```
def calculatepagerank(self,iterations=20):
  # clear out the current PageRank tables
  self.con.execute('drop table if exists pagerank')
  self.con.execute('create table pagerank(urlid primary key,score)')

  # initialize every url with a PageRank of 1
  self.con.execute('insert into pagerank select rowid, 1.0 from urllist')
  self.dbcommit( )

  for i in range(iterations):
    print "Iteration %d" % (i)
    for (urlid,) in self.con.execute('select rowid from urllist'):
      pr=0.15

      # Loop through all the pages that link to this one
      for (linker,) in self.con.execute(
      'select distinct fromid from link where toid=%d' % urlid):
        # Get the PageRank of the linker
        linkingpr=self.con.execute(
        'select score from pagerank where urlid=%d' % linker).fetchone( )[0]

        # Get the total number of links from the linker
        linkingcount=self.con.execute(
        'select count(*) from link where fromid=%d' % linker).fetchone( )[0]
        pr+=0.85*(linkingpr/linkingcount)
      self.con.execute(
      'update pagerank set score=%f where urlid=%d' % (pr,urlid))
    self.dbcommit( )
```

This function initially sets the PageRank of every page to 1.0. It then loops over every URL and gets the PageRank and the total number of links for every inbound link. The line in bold shows the formula being applied for each of the inbound links.

Running this function will take a few minutes, but you only need to do it when you update the index.

```
>> reload(searchengine)
>> crawler=searchengine.crawler('searchindex.db')
>> crawler.calculatepagerank( )
Iteration 0
Iteration 1
...
```

If you're curious about which pages from the example dataset have the highest PageRanks, you can query the database directly:

```
>> cur=crawler.con.execute('select * from pagerank order by score desc')
>> for i in range(3): print cur.next( )
(438, 2.5285160000000002)
(2, 1.1614640000000001)
(543, 1.064252)
>> e.geturlname(438)
u'http://kiwitobes.com/wiki/Main_Page.html'
```

"Main Page" has the highest PageRank, which is not surprising since every other page in Wikipedia links to it. Now that you have a table of PageRank scores, using them is just a matter of creating a function to retrieve them from the database and to normalize the scores. Add this method to the `searcher` class:

```
  def pagerankscore(self,rows):
    pageranks=dict([(row[0],self.con.execute('select score from pagerank where
urlid=%d' % row[0]).fetchone( )[0]) for row in rows])
    maxrank=max(pageranks.values( ))
    normalizedscores=dict([(u,float(l)/maxrank) for (u,l) in pageranks.items( )])
    return normalizedscores
```

Once again, you should modify the `weights` list to include PageRank. For example, try:

```
    weights=[(1.0,self.locationscore(rows)),
             (1.0,self.frequencyscore(rows)),
             (1.0,self.pagerankscore(rows))]
```

The results for your searches will take into account content and ranking scores. The results for "Functional programming" now look even better:

```
2.318146 http://kiwitobes.com/wiki/Functional_programming.html
1.074506 http://kiwitobes.com/wiki/Programming_language.html
0.517633 http://kiwitobes.com/wiki/Categorical_list_of_programming_languages.html
0.439568 http://kiwitobes.com/wiki/Programming_paradigm.html
0.426817 http://kiwitobes.com/wiki/Lisp_programming_language.html
```

The value of the PageRank score is a little harder to see with this closed, tightly controlled set of documents, which likely contains fewer useless pages or pages intended solely to get attention than you'd find on the Web. However, even in this case, it's clear that PageRank is a useful metric for returning higher-level, more general pages.

Using the Link Text

Another very powerful way to rank searches is to use the text of the links to a page to decide how relevant the page is. Many times you will get better information from what the links to a page say about it than from the linking page itself, as site developers tend to include a short description of whatever it is they are linking to.

The method for scoring the pages by their link text takes an additional argument, which is the list of word IDs produced when you perform a query. You can add this method to `searcher`:

```
  def linktextscore(self,rows,wordids):
    linkscores=dict([(row[0],0) for row in rows])
    for wordid in wordids:
      cur=self.con.execute('select link.fromid,link.toid from linkwords,link where
wordid=%d and linkwords.linkid=link.rowid' % wordid)
      for (fromid,toid) in cur:
        if toid in linkscores:
          pr=self.con.execute('select score from pagerank where urlid=%d' % fromid).
fetchone()[0]
          linkscores[toid]+=pr
    maxscore=max(linkscores.values())
    normalizedscores=dict([(u,float(l)/maxscore) for (u,l) in linkscores.items()])
    return normalizedscores
```

This code loops through all the words in `wordids` looking for links containing these words. If the target of the link matches one of the search results, then the PageRank of the source of the link is added to the destination page's final score. A page with a lot of links from important pages that contain the query terms will get a very high score. Many of the pages in the results will have no links with the correct text, and will get a score of 0.

To enable link-text ranking, just add the following anywhere in your `weights` list:

```
(1.0,self.linktextscore(rows,wordids))
```

There is no standard set of weightings for these metrics that will work in all cases. Even the major search sites frequently change their methods of ranking results. The metrics you'll use and the weights you'll give them are highly dependent on the application you're trying to build.

Learning from Clicks

One of the major advantages of online applications is that they receive constant feedback in the form of user behavior. In the case of a search engine, each user will immediately provide information about how much he likes the results for a given search by clicking on one result and choosing not to click on the others. This section will look at a way to record when a user clicks on a result after a query, and how that record can be used to improve the rankings of the results.

To do this, you're going to build an *artificial neural network* that you'll train by giving it the words in the query, the search results presented to the user, and what the user decided to click. Once the network has been trained with many different queries, you can use it to change the ordering of the search results to better reflect what users actually clicked on in the past.

Design of a Click-Tracking Network

While there are many different kinds of neural networks, they all consist of a set of nodes (the *neurons*) and connections between them. The network you'll learn how to build is called a *multilayer perceptron* (MLP) network. This type of network consists of multiple layers of neurons, the first of which takes the input—in this case, the words entered by the user. The last layer gives the output, which in this example is a list of weightings for the different URLs that were returned.

There can be multiple middle layers, but the network in this example will just use a single one. This is called the *hidden layer* because the outside world never interacts with it directly, and it responds to combinations of inputs. In this case, a combination of inputs is a set of words, so you could also think of this as the *query layer*. Figure 4-4 shows the structure of the network. All the nodes in the input layer are connected to all the nodes in the hidden layer, and all the nodes in the hidden layer are connected to all the nodes in the output layer.

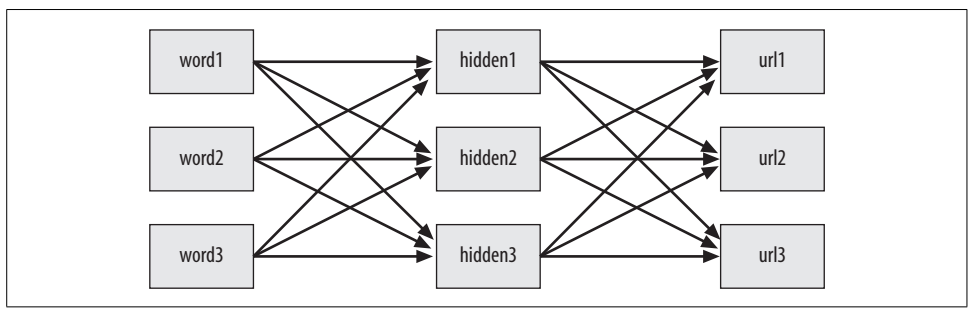

Figure 4-4. Design of a click-tracking neural network

To ask the neural network to get the best results for a query, the input nodes for the words in that query have their values set to 1. The outputs of those nodes are turned on and they attempt to activate the hidden layer. In turn, the nodes in the hidden layer that get a strong enough input will turn on their outputs and try to activate nodes in the output layer.

The nodes in the output layer then become active in various degrees, and their activity level can be used to determine how strongly a URL is associated with the words in the original query. Figure 4-5 shows a query for "world bank." The solid lines indicate strong connections, and the bold text indicates that a node has become very active.

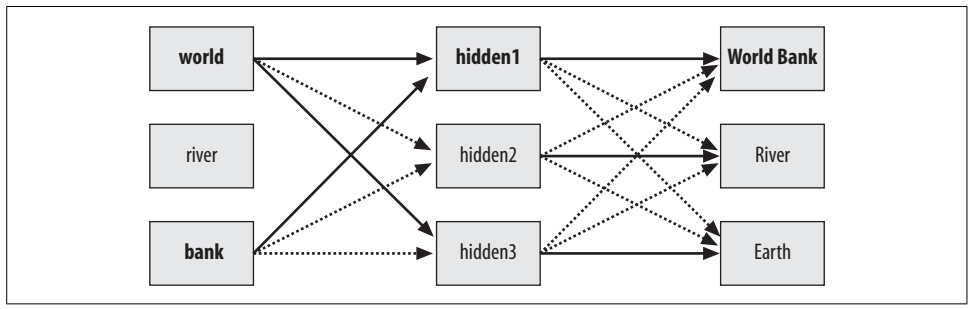

Figure 4-5. Neural network response to "world bank"

This, of course, depends on the connection strengths being correct. This is achieved by *training* the network every time someone performs a search and chooses one of the links out of the results. In the network pictured in Figure 4-5, a number of people had previously clicked the World Bank result after a search for "world bank," and this strengthened the associations between the words and the URL. This section will show you how the network is trained with an algorithm called `backpropagation`.

You might be wondering why you would need a sophisticated technique like a neural network instead of just remembering the query and counting how many times each result was clicked. The power of the neural network you're going to build is that it can make reasonable guesses about results for queries it has never seen before, based on their similarity to other queries. Also, neural networks are useful for a wide variety of applications and will be a great addition to your collective intelligence toolbox.

Setting Up the Database

Since the neural network will have to be trained over time as users perform queries, you'll need to store a representation of the network in the database. The database already has a table of words and URLs, so all that's needed is a table for the hidden layer (which will be called `hiddennode`) and two tables of connections (one from the word layer to the hidden layer, and one that links the hidden layer to the output layer).

Create a new file called *nn.py*, and create a new class in it called `searchnet`:

```
from math import tanh
from pysqlite2 import dbapi2 as sqlite

class searchnet:
    def __init__(self,dbname):
      self.con=sqlite.connect(dbname)

    def __del__(self):
      self.con.close( )

    def maketables(self):
      self.con.execute('create table hiddennode(create_key)')
      self.con.execute('create table wordhidden(fromid,toid,strength)')
      self.con.execute('create table hiddenurl(fromid,toid,strength)')
      self.con.commit( )
```

The tables currently have no indices, but you can add them later if speed is an issue.

You'll need to create a couple of methods to access the database. The first method, called `getstrength`, determines the current strength of a connection. Because new connections are only created when necessary, this method has to return a default value if there are no connections. For links from words to the hidden layer, the default value will be –0.2 so that, by default, extra words will have a slightly negative effect on the activation level of a hidden node. For links from the hidden layer to URLs, the method will return a default value of 0.

```
    def getstrength(self,fromid,toid,layer):
      if layer==0: table='wordhidden'
      else: table='hiddenurl'
      res=self.con.execute('select strength from %s where fromid=%d and toid=%d' %
(table,fromid,toid)).fetchone( )
      if res==None:
          if layer==0: return -0.2
          if layer==1: return 0
      return res[0]
```

You'll also need a `setstrength` method to determine if a connection already exists, and to update or create the connection with the new strength. This will be used by the code that trains the network:

```
    def setstrength(self,fromid,toid,layer,strength):
      if layer==0: table='wordhidden'
      else: table='hiddenurl'
      res=self.con.execute('select rowid from %s where fromid=%d and toid=%d' %
(table,fromid,toid)).fetchone( )
      if res==None:
        self.con.execute('insert into %s (fromid,toid,strength) values (%d,%d,%f)' %
(table,fromid,toid,strength))
      else:
        rowid=res[0]
        self.con.execute('update %s set strength=%f where rowid=%d' %
(table,strength,rowid))
```

Most of the time, when building a neural network, all the nodes in the network are created in advance. You could create a huge network up front with thousands of nodes in the hidden layer and all the connections already in place, but in this case, it will be faster and simpler to create new hidden nodes as they are needed.

This function will create a new node in the hidden layer every time it is passed a combination of words that it has never seen together before. The function then creates default-weighted links between the words and the hidden node, and between the query node and the URL results returned by this query.

```
def generatehiddennode(self,wordids,urls):
  if len(wordids)>3: return None
  # Check if we already created a node for this set of words
  createkey='_'.join(sorted([str(wi) for wi in wordids]))
  res=self.con.execute(
  "select rowid from hiddennode where create_key='%s'" % createkey).fetchone( )

  # If not, create it
  if res==None:
    cur=self.con.execute(
    "insert into hiddennode (create_key) values ('%s')" % createkey)
    hiddenid=cur.lastrowid
    # Put in some default weights
    for wordid in wordids:
      self.setstrength(wordid,hiddenid,0,1.0/len(wordids))
    for urlid in urls:
      self.setstrength(hiddenid,urlid,1,0.1)
    self.con.commit( )
```

In the Python interpreter, try creating a database and generating a hidden node with some example word and URL IDs:

```
>> import nn
>> mynet=nn.searchnet('nn.db')
>> mynet.maketables( )
>> wWorld,wRiver,wBank =101,102,103
>> uWorldBank,uRiver,uEarth =201,202,203
>> mynet.generatehiddennode([wWorld,wBank],[uWorldBank,uRiver,uEarth])
>> for c in mynet.con.execute('select * from wordhidden'): print c
(101, 1, 0.5)
(103, 1, 0.5)
>> for c in mynet.con.execute('select * from hiddenurl'): print c
(1, 201, 0.1)
(1, 202, 0.1)
...
```

A new node has been created in the hidden layer, and links to the new node have been created with default values. The function will initially respond whenever “world” and “bank” are entered together, but these connections may weaken over time.

Feeding Forward

You're now ready to make functions that will take the words as inputs, activate the links in the network, and give a set of outputs for the URLs.

First, choose a function that indicates how much each node should respond to its input. The neural network described here will use the *hyperbolic tangent* (tanh) function, shown in Figure 4-6.

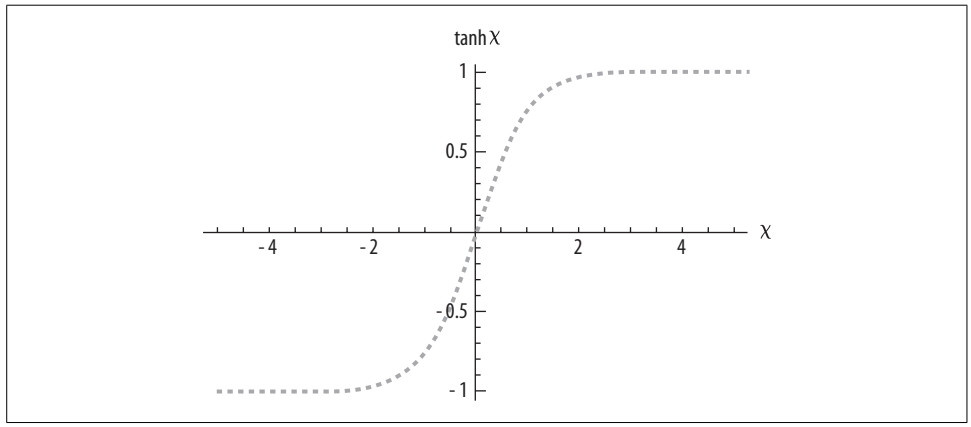

Figure 4-6. The tanh function

The x-axis is the total input to the node. As the input approaches 0, the output starts to climb quickly. With an input of 2, the output is almost at 1 and doesn't get much higher. This is a type of *sigmoid* function, all types of which have this S shape. Neural networks almost always use sigmoid functions to calculate the outputs of the neurons.

Before running the `feedforward` algorithm, the class will have to query the nodes and connections in the database, and build, in memory, the portion of the network that is relevant to a specific query. The first step is to create a function that finds all the nodes from the hidden layer that are relevant to a specific query—in this case, they must be connected to one of the words in the query or to one of the URLs in the results. Since the other nodes will not be used either to determine an outcome or to train the network, it's not necessary to include them:

```
def getallhiddenids(self,wordids,urlids):
  l1={}
  for wordid in wordids:
    cur=self.con.execute(
    'select toid from wordhidden where fromid=%d' % wordid)
    for row in cur: l1[row[0]]=1
  for urlid in urlids:
    cur=self.con.execute(
    'select fromid from hiddenurl where toid=%d' % urlid)
    for row in cur: l1[row[0]]=1
  return l1.keys( )
```

You will also need a method for constructing the relevant network with all the current weights from the database. This function sets a lot of instance variables for this class—the list of words, query nodes and URLs, the output level of every node, and the weights of every link between nodes. The weights are taken from the database using the functions that were defined earlier.

```
def setupnetwork(self,wordids,urlids):
    # value lists
    self.wordids=wordids
    self.hiddenids=self.getallhiddenids(wordids,urlids)
    self.urlids=urlids

    # node outputs
    self.ai = [1.0]*len(self.wordids)
    self.ah = [1.0]*len(self.hiddenids)
    self.ao = [1.0]*len(self.urlids)

    # create weights matrix
    self.wi = [[self.getstrength(wordid,hiddenid,0)
                for hiddenid in self.hiddenids]
               for wordid in self.wordids]
    self.wo = [[self.getstrength(hiddenid,urlid,1)
                for urlid in self.urlids]
               for hiddenid in self.hiddenids]
```

You're finally ready to create the `feedforward` algorithm. This takes a list of inputs, pushes them through the network, and returns the output of all the nodes in the output layer. In this case, since you've only constructed a network with words in the query, the output from all the input nodes will always be 1:

```
def feedforward(self):
    # the only inputs are the query words
    for i in range(len(self.wordids)):
        self.ai[i] = 1.0

    # hidden activations
    for j in range(len(self.hiddenids)):
        sum = 0.0
        for i in range(len(self.wordids)):
            sum = sum + self.ai[i] * self.wi[i][j]
        self.ah[j] = tanh(sum)

    # output activations
    for k in range(len(self.urlids)):
        sum = 0.0
        for j in range(len(self.hiddenids)):
            sum = sum + self.ah[j] * self.wo[j][k]
        self.ao[k] = tanh(sum)

    return self.ao[:]
```

The `feedforward` algorithm works by looping over all the nodes in the hidden layer and adding together all the outputs from the input layer multiplied by the strengths of the links. The output of each node is the `tanh` function of the sum of all the inputs, which is passed on to the output layer. The output layer does the same thing, multiplying the outputs of the previous layer by their strengths, and applies the `tanh` function to produce the final output. It is easy to extend the network to have more layers by continually using the output of one layer as the input to the next layer.

Now you can write a short function that will set up the network and use `feedforward` to get the outputs for a set of words and URLs:

```
def getresult(self,wordids,urlids):
  self.setupnetwork(wordids,urlids)
  return self.feedforward( )
```

You can use Python to try this in the network:

```
>> reload(nn)
>> mynet=nn.searchnet('nn.db')
>> mynet.getresult([wWorld,wBank],[uWorldBank,uRiver,uEarth])
[0.76,0.76,0.76]
```

The numbers in the returned list correspond to the relevance of the input URLs. Not surprisingly, because it hasn't yet had any training, the neural network gives the same answer for every URL.

Training with Backpropagation

Here's where things get interesting. The network will take inputs and give outputs, but because it hasn't been taught what a good result looks like, the results are pretty useless. You're now going to train the network by showing it some actual examples of what people searched for, which results were returned, and what the users decided to click on.

For this to work, you need an algorithm that alters the weights of the links between the nodes to better reflect what the network is being told is the right answer. The weights have to be adjusted slowly because you can't assume that the each user will click on an answer that's appropriate for everyone. The algorithm you'll use is called *backpropagation* because it moves backward through the network adjusting the weights.

When training a network, you always know the desired output of each node in the output layer. In this case, it should be pushed toward 1 if the user clicked on that result, and pushed toward 0 if he did not. The only way to change the output of a node is to change the total input to that node.

To determine how much the total input should be changed, the training algorithm has to know the slope of the `tanh` function at its current level of output. In the middle of the function, when the output is 0.0, the slope is very steep, so changing

the input by only a small amount gives a big change. As the outputs get closer to –1 or 1, changing the input has a smaller effect on the output. The slope of the function for any output value is specified by this function, which you can add to the start of *nn.py*:

```
def dtanh(y):
    return 1.0-y*y
```

Before running the backpropagation method, it's necessary to run `feedforward` so that the current output of every node will be stored in the instance variables. The backpropagation algorithm then performs the following steps.

For each node in the output layer:

1. Calculate the difference between the node's current output and what it should be.
2. Use the `dtanh` function to determine how much the node's total input has to change.
3. Change the strength of every incoming link in proportion to the link's current strength and the learning rate.

For each node in the hidden layer:

1. Change the output of the node by the sum of the strength of each output link multiplied by how much its target node has to change.
2. Use the `dtanh` function to determine how much the node's total input has to change.
3. Change the strength of every input link in proportion to the link's current strength and the learning rate.

The implementation of this algorithm actually calculates all the errors in advance and then adjusts the weights, because all the calculations rely on knowing the current weights rather than the updated weights. Here's the code for the algorithm, which you can add to the `searchnet` class:

```
def backPropagate(self, targets, N=0.5):
    # calculate errors for output
    output_deltas = [0.0] * len(self.urlids)
    for k in range(len(self.urlids)):
        error = targets[k]-self.ao[k]
        output_deltas[k] = dtanh(self.ao[k]) * error

    # calculate errors for hidden layer
    hidden_deltas = [0.0] * len(self.hiddenids)
    for j in range(len(self.hiddenids)):
        error = 0.0
        for k in range(len(self.urlids)):
            error = error + output_deltas[k]*self.wo[j][k]
        hidden_deltas[j] = dtanh(self.ah[j]) * error
```

```
        # update output weights
        for j in range(len(self.hiddenids)):
            for k in range(len(self.urlids)):
                change = output_deltas[k]*self.ah[j]
                self.wo[j][k] = self.wo[j][k] + N*change

        # update input weights
        for i in range(len(self.wordids)):
            for j in range(len(self.hiddenids)):
                change = hidden_deltas[j]*self.ai[i]
                self.wi[i][j] = self.wi[i][j] + N*change
```

Now all you need is a simple method that will set up the network, run `feedforward`, and run the backpropagation. This method takes the list of `wordids`, `urlids`, and a selected URL:

```
def trainquery(self,wordids,urlids,selectedurl):
  # generate a hidden node if necessary
  self.generatehiddennode(wordids,urlids)

  self.setupnetwork(wordids,urlids)
  self.feedforward( )
  targets=[0.0]*len(urlids)
  targets[urlids.index(selectedurl)]=1.0
  error = self.backPropagate(targets)
  self.updatedatabase( )
```

To save the results, you'll also need a method to update the database with the new weights, which are stored in the `wi` and `wo` instance variables:

```
def updatedatabase(self):
  # set them to database values
  for i in range(len(self.wordids)):
      for j in range(len(self.hiddenids)):
          self.setstrength(self.wordids[i],self. hiddenids[j],0,self.wi[i][j])
  for j in range(len(self.hiddenids)):
      for k in range(len(self.urlids)):
          self.setstrength(self.hiddenids[j],self.urlids[k],1,self.wo[j][k])
  self.con.commit( )
```

Now you can do a simple test with the query you tried earlier to see how the network responds to training:

```
>> reload(nn)
>> mynet=nn.searchnet('nn.db')
>> mynet.trainquery([wWorld,wBank],[uWorldBank,uRiver,uEarth],uWorldBank)
>> mynet.getresult([wWorld,wBank],[uWorldBank,uRiver,uEarth])
[0.335,0.055,0.055]
```

The output for the World Bank URL increased and the output for the other URLs decreased after the network learned that a particular user made that selection. The more users make this selection, the bigger the difference will get.

Training Test

So far you've seen that training with one sample result increases the output for that result. Although that's useful, it doesn't really demonstrate what neural networks are capable of—that is, reasoning about inputs they've never seen before. Try this code in your interactive Python session:

```
>> allurls=[uWorldBank,uRiver,uEarth]
>> for i in range(30):
...     mynet.trainquery([wWorld,wBank],allurls,uWorldBank)
...     mynet.trainquery([wRiver,wBank],allurls,uRiver)
...     mynet.trainquery([wWorld],allurls,uEarth)
...
>> mynet.getresult([wWorld,wBank],allurls)
[0.861, 0.011, 0.016]
>> mynet.getresult([wRiver,wBank],allurls)
[-0.030, 0.883, 0.006]
>> mynet.getresult([wBank],allurls)
[0.865, 0.001, -0.85]
```

Even though the network has never seen a query for "bank" by itself before, it gives a reasonable guess. Not only that, it gives the World Bank URL a much better score than the River URL, even though in the training sample queries "bank" was associated just as often with "river" as it was with World Bank. The network has not only learned which URLs are related to which queries, it has also learned what the important words are in a particular query—something that could not have been achieved with a simple query-URL correlation.

Connecting to the Search Engine

The `query` method of the `searcher` class gets a list of URL IDs and word IDs in the course of creating and printing the results. You can have the method return these results by adding the following line to the end of the `query` in *searchengine.py*:

```
return wordids,[r[1] for r in rankedscores[0:10]]
```

These can be passed directly to the `trainquery` method of `searchnet`.

The method for capturing which of the results the user liked best is specific to the design of your application. It's possible that a web page could include an intermediate page that captures the click and calls `trainquery` before redirecting to the actual search, or you could even let users vote on the relevance of search results to help improve your algorithm.

The final step in building the artificial neural network is creating a new method in the `searcher` class to allow you to weight the results. This function looks pretty similar to the other weighting functions. The first thing you'll need to do is import the neural network class in *searchengine.py*:

```
import nn
mynet=nn.searchnet('nn.db')
```

And add this method to the searcher class:

```
def nnscore(self,rows,wordids):
  # Get unique URL IDs as an ordered list
  urlids=[urlid for urlid in set([row[0] for row in rows])]
  nnres=mynet.getresult(wordids,urlids)
  scores=dict([(urlids[i],nnres[i]) for i in range(len(urlids))])
  return self.normalizescores(scores)
```

Again, you can experiment by including this in your weights list with various weights. In practice, it's better to hold off on including it as part of your scoring until the network has been trained on a large number of different examples.

This chapter has covered a wide range of possibilities for developing a search engine, but it's still very limited compared to what's possible. The exercises will explore some further ideas. This chapter has not focused on performance—which would require work to index millions of pages—but what you've built will perform adequately on a set of 100,000 pages, enough for a news site or corporate intranet.

Exercises

1. *Word separation*. The separatewords method currently considers any nonalphanumeric character to be a separator, meaning it will not properly index entries like "C++," "$20," "Ph.D.," or "617-555-1212." What is a better way to separate words? Does using whitespace as a separator work? Write a better word separation function.
2. *Boolean operations*. Many search engines support Boolean queries, which allow users to construct searches like "python OR perl." An OR search can work by doing the queries separately and combining the results, but what about "python AND (program OR code)"? Modify the query methods to support some basic Boolean operations.
3. *Exact matches*. Search engines often support "exact match" queries, where the words in the page must match the words in the query in the same order with no additional words in between. Create a new version of getrows that only returns results that are exact matches. (Hint: you can use subtraction in SQL to get the difference between the word locations.)
4. *Long/short document search*. Sometimes the length of a page will be a determining factor in whether it is relevant to a particular search application or user. A user may be interested in finding a long article about a difficult subject or a quick reference page for a command-line tool. Write a weighting function that will give preference to longer or shorter documents depending on its parameters.
5. *Word frequency bias*. The "word count" metric is biased to favor longer documents, since a long document has more words and can therefore contain the target words more often. Write a new metric that calculates frequency as a percentage of the number of words in the document.

6. *Inbound link searching*. Your code can rank items based on the text of the inbound links, but they must already be results based on the content of the page. Sometimes the most relevant page doesn't contain the query text at all, but rather a lot of links with the text pointing to it—this is often the case with links to images. Modify the search code to also include results where an inbound link contains some of the search terms.
7. *Different training options*. The neural network is trained with a set of 0s for all the URLs that a user did not click, and a 1 for the URL that she did click. Alter the training function so that it works instead for an application where users get to rate results from 1 to 5.
8. *Additional layers*. Your neural network has only one hidden layer. Update the class to support an arbitrary number of hidden layers, which can be specified upon initialization.