Algorithms of the Intelligent Web

Haralambos Marmanis
Dmitry Babenko

SAMPLE CHAPTER

MANNING

Algorithms of the Intelligent Web

by Haralambos Marmanis
and Dmitry Babenko

Chapter 2

brief contents

2 Searching

This chapter covers:

- Searching with Lucene
- Calculating the PageRank vector
- Large-scale computing constraints

Let's say that you have a list of documents and you're interested in reading about those that are related to the phrase "Armageddon is near"—or perhaps something less macabre. How would you implement a solution to that problem? A brute force, and naïve, solution would be to read each document and keep only those in which you can find the term "Armageddon is near." You could even count how many times you found each of the words in your search term within each of the documents and sort them according to that count in descending order. That exercise is called *information retrieval (IR)* or simply searching. Searching isn't new functionality; nearly every application has some implementation of search, but intelligent searching goes beyond plain old searching.

Experimentation can convince you that the naïve IR solution is full of problems. For example, as soon as you increase the number of documents, or their size, its performance will become unacceptable for most purposes. Fortunately, there's an enormous amount of knowledge about IR and fairly sophisticated and robust libraries are available that offer scalability and high performance. The most successful IR library in the Java programming language is Lucene, a project created by Doug Cutting almost 10 years ago. Lucene can help you solve the IR problem by indexing all your documents and letting you search through them at lightning speeds! *Lucene in Action* by Otis Gospodnetić and Erik Hatcher, published by Manning, is a must-read, especially if you want to know how to index data and introduces search, sorting, filtering and highlighting search results.

State-of-the-art searching goes well beyond indexing. The fiercest competition among search engine companies doesn't involve the technology around indexing but rather subjects such as link analysis, user click analysis, and natural-language processing. These techniques strengthen the searching functionality, sometimes to the tune of billions of dollars, as was the case with Google.

In this chapter, we'll summarize the features of the Lucene library and demonstrate its use. We'll present the PageRank algorithm, which has been the most successful link analysis algorithm so far, and we'll present a probabilistic technique for conducting user click analysis. We'll combine all these techniques to demonstrate the improvement in the search results due to the synergies among them. The material is presented in a successive manner, so you can learn as much as you want about searching and come back to it later if you don't have enough time now. Without further ado, let's collect a number of documents and search for various terms in them by using Lucene.

2.1 Searching with Lucene

Searching with Lucene will be our baseline for the rest of the chapter. So, before we embark on advanced intelligent algorithms, we need to learn the traditional IR steps. On our journey, we'll show you how to use Lucene to search a set of collected documents, we'll present some of the inner workings of Lucene, and we'll provide an overview of the basic stages for building a search engine.

The data that you want to search could be in your database, on the internet, or on any other network that's accessible to your application. You can collect data from the internet by using a crawler. A number of crawlers are freely available, but we'll use a crawler that we wrote for the purposes of this book. We'll use a number of pages that we collected on November 6, 2006, so we can modify them in a controlled fashion and observe the effect of these changes in the results of the algorithms.

These pages have been cleaned up and changed to form a tiny representation of the internet. You can find these pages under the `data/ch02/` directory. It's important to know the content of these documents, so that you can appreciate what the algorithms do and understand how they work. Our 15 documents are (the choice of content was random):

- Seven documents related to business news; three are related to Google's expansion into newspaper advertisement, another three discuss primarily about the NVidia stock, and one about stock price and index movements.
- Three documents related to Lance Armstrong's attempt to run the marathon in New York.
- Four documents related to U.S. politics and, in particular, the congressional elections (circa 2006).
- Five documents related to world news; four about Ortega winning the elections in Nicaragua and one about global warming.

Lucene can help us analyze, index, and search these and any other document that can be converted into text, so it's not limited to web pages. The class that we'll use to quickly read the stored web pages is called `FetchAndProcessCrawler`; this class can also retrieve data from the internet. Its constructor takes three arguments:

- The base directory for storing the retrieved data.
- The depth of the link structure that should be traversed.
- The maximum number of total documents that should be retrieved.

Listing 2.1 shows how you can use it from the BeanShell.

Listing 2.1 Reading, indexing, and searching the default list of web pages

```
FetchAndProcessCrawler crawler =
➥ new FetchAndProcessCrawler("C:/iWeb2/data/ch02",5,200);

crawler.setDefaultUrls();     <— Load files

crawler.run();                <— Gather and process content

LuceneIndexer luceneIndexer =
➥ new LuceneIndexer(crawler.getRootDir());

luceneIndexer.run();     <— Index content in directory

MySearcher oracle = new MySearcher(luceneIndexer.getLuceneDir());

oracle.search("armstrong",5);     <— Search based on index just created
```

The crawling and preprocessing stage should take only a few seconds, and when it finishes you should have a new directory under the base directory. In our example, the base directory was `C:/iWeb2/data/ch02`. The new directory's name will start with the string `crawl-` and be followed by the numeric value of the crawl's timestamp in milliseconds—for example, `crawl-1200697910111`.

You can change the content of the documents, or add more documents, and rerun the preprocessing and indexing of the files in order to observe the differences in your search results. Figure 2.1 is a snapshot of executing the code from listing 2.1 in the BeanShell, and it includes the results of the search for the term "armstrong."

```
bsh % FetchAndProcessCrawler c =
new FetchAndProcessCrawler("c:/iWeb2/data/ch02",5,200);
bsh % c.setDefaultUrls();
bsh % c.run();
There are no unprocessed urls.
Timer (s): [Crawler fetched data] --> 5.5
Timer (s): [Crawler processed data] --> 0.485
bsh %
bsh % LuceneIndexer lidx = new LuceneIndexer(c.getRootDir());
bsh % lidx.run();
Starting the indexing ... Indexing completed!

bsh % MySearcher oracle = new MySearcher(lidx.getLuceneDir());
bsh % oracle.search("armstrong",5);

Search results using Lucene index scores:
Query: armstrong

Document Title: Lance Armstrong meets goal in painful marathon
debut
Document URL: file:/c:/iWeb2/data/ch02/sport-01.html -->
Relevance Score: 0.397706508636475
___________________________________
Document Title: New York 'tour' Lance's toughest
Document URL: file:/c:/iWeb2/data/ch02/sport-03.html -->
Relevance Score: 0.312822639942169
___________________________________
Document Title: New York City Marathon
Document URL: file:/c:/iWeb2/data/ch02/sport-02.html -->
Relevance Score: 0.226110160350800
___________________________________
```

Figure 2.1 An example of retrieving, parsing, analyzing, indexing, and searching a set of web pages with a few lines of code

Those are the high-level mechanics: load, index, search. It doesn't get any simpler than that! But how does it really work? What are the essential elements that participate in each stage?

2.1.1 Understanding the Lucene code

Let's examine the sequence of events that allowed us to perform our search. The job of the `FetchAndProcessCrawler` class is to retrieve the data and parse it. The result of that processing is stored in the subdirectory called `processed`. Take a minute to look in that folder. For every group of documents that are processed, there are four subdirectories—`fetched`, `knownurls`, `pagelinks`, and `processed`. Note we've dissected the web pages by separating metadata from the core content and by extracting the links from one page to another—the so-called *outlinks*. The `FetchAndProcessCrawler` class doesn't use any code from the Lucene API.

The next thing that we did was create an instance of the LuceneIndexer class and call its run() method. This is where we use Lucene to index our processed content. The Lucene index files will be stored in a separate directory called lucene-index. The LuceneIndexer class is a convenience wrapper that helps us invoke the LuceneIndexBuilder class from the Bean shell. The LuceneIndexBuilder class is where the Lucene API is used. Figure 2.2 shows the complete UML diagram of the main classes involved in retrieving and indexing the documents.

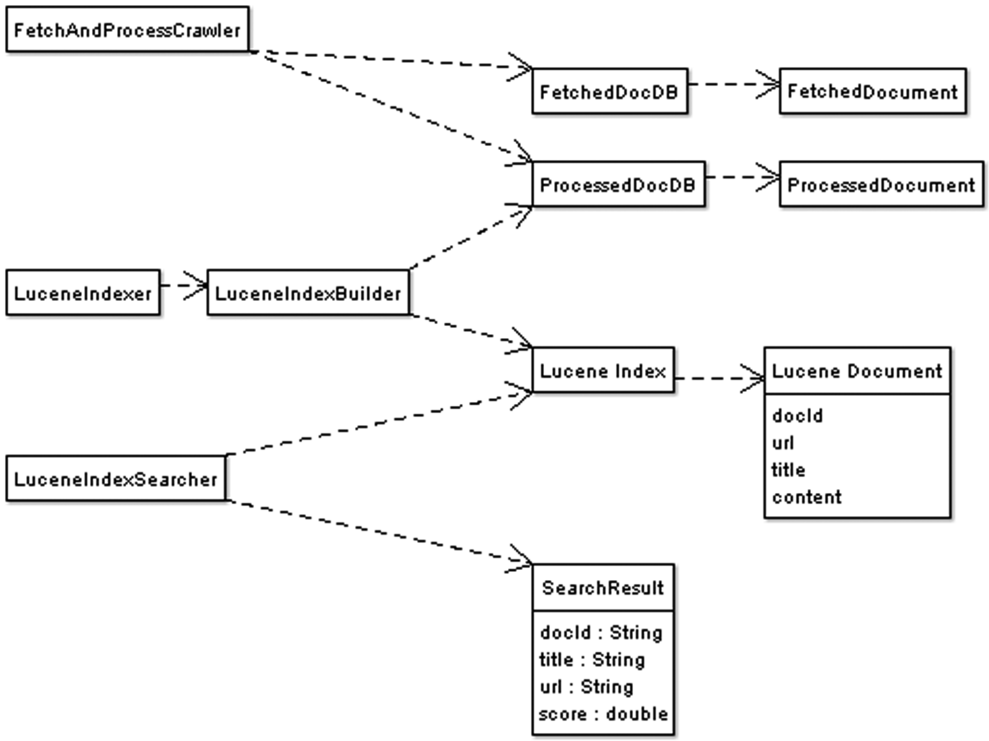

Figure 2.2 A UML diagram of the classes that we used to crawl, index, and search a set of web pages

Listing 2.2 shows the entire code from the LuceneIndexBuilder class.

Listing 2.2 The LuceneIndexBuilder creates a Lucene index

```
public class LuceneIndexBuilder implements CrawlDataProcessor {
 private File indexDir;
 public LuceneIndexBuilder(File indexDir) {
    this.indexDir = indexDir;
     try {
        IndexWriter indexWriter =                    Create Lucene index
        new IndexWriter(indexDir, new StandardAnalyzer(), true);
        indexWriter.close();
```

```
} catch(IOException ioX) {
         throw new RuntimeException("Error: ", ioX);
      }
   }
   public void run(CrawlData crawlData) {

      List<String> allGroups =                                    Get all document
        crawlData.getProcessedDocsDB().getAllGroupIds();          groups

      for(String groupId : allGroups) {
       buildLuceneIndex(groupId, crawlData.getProcessedDocsDB());
      }
   }

   private void buildLuceneIndex(String groupId,
➥ ProcessedDocsDB parsedDocsService) {

      try {                                              Get all documents
                                                         for group
         List<String> docIdList =
parsedDocsService.getDocumentIds(groupId);

         IndexWriter indexWriter =

new IndexWriter(indexDir, new StandardAnalyzer(), false);

         for(String docId : docIdList) {                  Index all
                                                          documents
            indexDocument(indexWriter,
➥ parsedDocsService.loadDocument(docId));
         }

         indexWriter.close();

      } catch(IOException ioX) {
         throw new RuntimeException("Error: ", ioX);
      }
   }

   private void indexDocument(IndexWriter iw,
➥ ProcessedDocument parsedDoc) throws IOException {

      org.apache.lucene.document.Document doc =
➥ new org.apache.lucene.document.Document();

      doc.add(new Field("content", parsedDoc.getText(),
➥ Field.Store.NO, Field.Index.TOKENIZED));

      doc.add(new Field("url",
➥ parsedDoc.getDocumentURL().toExternalForm(),
➥ Field.Store.YES, Field.Index.NO));

      doc.add(new Field("docid", parsedDoc.getDocumentId(),
➥ Field.Store.YES, Field.Index.NO));

      doc.add(new Field("title", parsedDoc.getDocumentTitle(),
➥ Field.Store.YES, Field.Index.NO));

      doc.add(new Field("doctype", parsedDoc.getDocumentType(),
➥ Field.Store.YES,Field.Index.NO));
      iw.addDocument(doc);
   }
}
```

The `IndexWriter` class is what Lucene uses to create an index. It comes with a large number of constructors, which you can peruse in the Javadocs. The specific constructor that we use in our code takes three arguments:

- The directory where we want to store the index.
- The analyzer that we want to use—we'll talk about analyzers later in this section.
- A Boolean variable that determines whether we need to override the existing directory.

As you can see in listing 2.2, we iterate over the groups of documents that our crawler has accumulated. The first group corresponds to the content of the initial URL list. The second group contains the documents that we found while reading the content of the initial URL list. The third group will contain the documents that are reachable from the second group, and so on. Note that the structure of these directories changes if you vary the parameter `maxBatchSize` of the `BasicWebCrawler` class. To keep the described structure intact, make sure that the value of that parameter is set to a sufficiently large number; for the purposes of this book, it's set to 50.

This directory structure will be useful when you use our crawler to retrieve a much larger dataset from the internet. For the simple web page structure that we'll use in the book, you can see the effect of grouping if you add only a few URLs—by using the `addUrl` method of the `FetchAndProcessCrawler` class—and let the crawler discover the rest of the files.

For each document within a group, we index its content. This takes place inside the `indexDocument` method, which is shown at the bottom of listing 2.2. The Lucene `Document` class encapsulates the documents that we've retrieved so that we can add them in the index; that same class can be used to encapsulate not only web pages but also emails, PDF files, and anything else that you can parse and transform into plain text. Every instance of the `Document` class is a virtual document that represents a collection of fields. Note that we're using our dissection of the retrieved documents to create various `Field` instances for each document:

- The `content` field, which corresponds to the text representation of each document, stripped of all the formatting tags and other annotations. You can find these documents under the subdirectory `processed/1/txt`.
- The `url` field represents the URL that was used to retrieve this document.
- The `docid` field, which uniquely identifies each document.
- The `title` field, which stores the title of each document.
- The `doctype` field, which stores the document type of each document, such as HTML or Microsoft Word.

The field content of every document is indexed but isn't stored with the index files; the other fields are stored with the index files but they aren't indexed. The reason being we want to be able to query against the content but we want to retrieve from the index files the URL, the ID, and the title of each retrieved document.

This practice is common. You typically store a few pointers that allow you to identify what you've found in the index, but you don't include the content inside the index files unless you have good reasons for doing so (you may need part of the content immediately and the original source isn't directly accessible). In that case, pay attention to the size of the files that you're creating during the indexing stage.

We use the `MySearcher` class to search through our newly created index. Listing 2.3 shows all the code in that class. It requires a single argument to construct it—the directory where we stored the Lucene index—and then it allows us to search through the `search` method, which uses two arguments:

- A string that contains the query that we want to execute against the index
- The maximum number of documents that we want to retrieve

Listing 2.3 MySearcher: retrieving search results based on Lucene indexing

```
public class MySearcher {

  private static final Logger log =
➥   Logger.getLogger(MySearcher.class);

  private String indexDir;

  public MySearcher(String indexDir) {
    this.indexDir = indexDir;
  }

  public SearchResult[] search(String query, int numberOfMatches) {

    SearchResult[] docResults = new SearchResult[0];
    IndexSearcher is = null;

    try {

      is = new IndexSearcher(FSDirectory.getDirectory(indexDir));    Open Lucene index

    } catch (IOException ioX) {
      log.error(ioX.getMessage());
    }

QueryParser qp = new QueryParser("content",                Create query parser
                            new StandardAnalyzer());
    Query q = null;
    try {

      q = qp.parse(query);                 Transform text query into Lucene query

    } catch (ParseException pX) {
      log.error(pX.getMessage());
    }

    Hits hits = null;
    try {

      hits = is.search(q);             Search index

      int n = Math.min(hits.length(), numberOfMatches);
      docResults = new SearchResult[n];
```

```
    for (int i = 0; i < n; i++) {                          <-- Collect first
                                                               N search
      docResults[i] = new SearchResult(hits.doc(i).get("docid"),   results
                           hits.doc(i).get("doctype"),
                           hits.doc(i).get("title"),
                           hits.doc(i).get("url"),
                           hits.score(i));              <-- Score for i-th
                                                            document
   // report the results
      System.out.println(docResults[i].print());
    }
    is.close();

  } catch (IOException ioX) {
     log.error(ioX.getMessage());
  }
  return docResults;
 }
}
```

Let's review the steps in listing 2.3:

1. We use an instance of the Lucene `IndexSearcher` class to open our index for searching.
2. We create an instance of the Lucene `QueryParser` class by providing the name of the field that we query against and the analyzer that must be used for tokenizing the query text.
3. We use the `parse` method of the `QueryParser` to transform the human-readable query into a `Query` instance that Lucene can understand.
4. We search the index and obtain the results in the form of a Lucene `Hits` object.
5. We loop over the first *n* results and collect them in the form of our own `SearchResult` objects. Note that Lucene's `Hits` object contains only references to the underlying documents. We use these references to collect the required fields; for example, the call `hits.doc(i).get("url")` will return the URL that we stored in the index.
6. The *relevance score* for each retrieved document is recorded. This score is a number between 0 and 1.

Those elements constitute the mechanics of our specific implementation. Let's take a step back and view the bigger picture of conducting searches based on indexing. This will help us understand the individual contributions of index-based search engines, and will prepare us for a discussion about more advanced search features.

2.1.2 *Understanding the basic stages of search*

If we could travel back in time (let's say to 1998), what would be the basic stages of work we'd need to perform to build a search engine? These stages are the same today as they were in 1998 but we've improved their effectiveness and computational performance. Figure 2.3 depicts the basic stages in conventional searching:

- Crawling
- Parsing
- Analyzing
- Indexing
- Searching

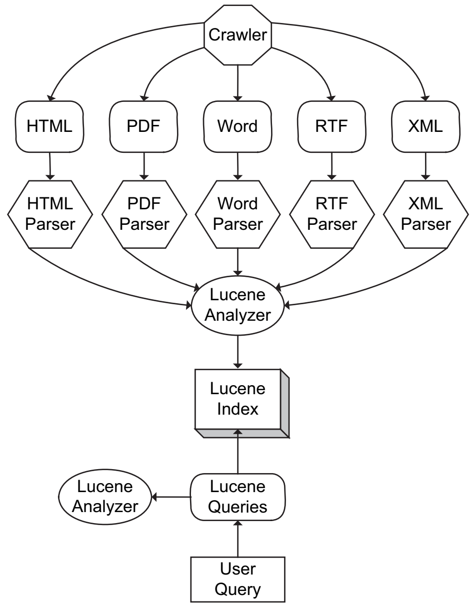

Figure 2.3 An overview of searching for a set of documents with different formats

Crawling refers to the process of gathering the documents on which we want to enable the search functionality. It may not be necessary if the documents exist or have been collected already. Parsing is necessary for transforming the documents (XML, HTML, Word, PDF) into a common structure that will represent the fields of indexing in a purely textual form. For our examples, we're using the code from the NekoHTML project. NekoHTML contains a simple HTML parser that can scan HTML files and "fix" many common mistakes that occur in HTML documents, adding missing parent elements, automatically closing elements with optional end tags, and handling mismatched inline element tags. NekoHTML is fairly robust and sufficiently fast, but if you're crawling special sites, you may want to write your own parser.

If you plan to index PDF documents, you can use the code from the PDFBox project (http://www.pdfbox.org/); it's released under the BSD license and has plenty of documentation. PDFBox includes the class `LucenePDFDocument`, which can be used to obtain a Lucene `Document` object immediately with a single line of code such as the following:

```
Document doc = LucenePDFDocument.convertDocument(File file)
```

Look at the Javadocs for additional information. Similar to PDF documents, there are also parsers for Word documents. For example, the Apache POI project (http://poi.apache.org/) provides APIs for manipulating file formats based on Microsoft's OLE 2 Compound Document format using pure Java. In addition, the TextMining code, available at http://www.textmining.org/, provides a Java library for extracting text from Microsoft Word 97, 2000, XP, and 2003 documents.

The stage of analyzing the documents is very important. In listing 2.2 and listing 2.3, the Lucene class `StandardAnalyzer` was used in two crucial places in the code, but we didn't discuss it before now. As figure 2.3 indicates, our parsers will be used to extract text from their respective documents, but before the textual content is indexed, it's processed by a Lucene analyzer. The work of an analyzer is crucial because analyzers are responsible for tokenizing the text that's to be indexed. This means that they'll keep some words from the text that they consider to be important while they ignore everything else. If you ignore something that's of interest to you during the analysis stage then you'll never find it during your search, no matter how sophisticated your indexing algorithm is.

Of course, analyzers can't select the appropriate fields for you. As an example, in listing 2.2, we've explicitly defined the four fields that we're interested in. The `StandardAnalyzer` will process the `content` field, which is the only field indexed. This default analyzer is the most general purpose built-in analyzer for Lucene. It intelligently tokenizes alphanumerics, acronyms, company names, email addresses, computer host names, and even CJK (Chinese, Japanese, and Korean) characters, among other things.

The latest version of Lucene (2.3 at the time of this writing) uses a lexical analyzer that's written in Java and called JFlex (http://jflex.de/). The Lucene `StandardTokenizer` is a grammar-based tokenizer that's constructed with JFlex, and it's used in the `StandardAnalyzer`. To convince you of the analyzer's importance, replace the `StandardAnalyzer` with the `WhitespaceAnalyzer` and observe the difference in the resulting scores. Lucene analyzers provide a wealth of capabilities, such as the ability to add synonyms, modify stop words (words that are explicitly removed from the text before indexing), and deal with non-English languages. We'll use Lucene analyzers throughout the book, even in chapters that don't deal with search. The general idea of identifying the unique characteristics of a text description is crucial when we deal with documents. Thus, analyzers become very relevant in areas such as the development of spam filters, recommendations that are based on text, enterprise, or tax compliance applications, and so on.

The Lucene indexing stage is completely transparent to the end user but it's also powerful. In a single index, you can have Lucene `Documents` that correspond to different entities (such as emails, memos, legal documents) and therefore are characterized by different fields. You can also remove or update `Documents` from an index. Another interesting feature of Lucene's indexing is *boosting*. Boosting allows you to mark certain documents as more or less important than other documents. In the method `indexDocument` described in the listing 2.2, you could add a statement such as the following:

```
if ( parsedDoc.getDocumentId().equals("g1-d14")) {
      doc.setBoost(2);
}
```

You can find this statement in the code, commented out and marked as "To Do." If you remove the comments, compile the code, and run again the script of listing 2.1, you'll notice that the last document is now first. Boosting has increased—in fact, it has doubled—the score of every `Field` for this document. You can also boost individual `Fields` in order to achieve more granular results from your boosting.

Searching with Lucene can't be easier. As you've seen, using our `MySearcher` wrapper, it's a matter of two lines of code. Although we used a simple word in our example of listing 2.1, Lucene provides sophisticated query expression parsing through the

`QueryParser` class. Sometimes you may have to use different means for creating the Lucene `Query`. To search for the term "nasdaq index" and allow for the possibility of results that refer to "nasdaq composite index," you'd use the class `PhraseQuery`. In this case, the term "index" can be a term apart from the term "nasdaq". The maximum number of terms that can separate "nasdaq" and "index" is set by a parameter called *slope*. By setting the slope equal to 1, we can achieve the desired result. For this and more powerful features of searching with Lucene, we encourage you to explore the Lucene APIs and documentation.

2.2 Why search beyond indexing?

Now that we've showed you how to quickly index your documents with Lucene and execute queries against those indices, you're probably convinced that using Lucene is easy and wonderful. You may wonder: "If Lucene is so sophisticated and efficient, why bother with anything else?" In this section we'll demonstrate why searching beyond indexing is necessary. We mentioned the reasons in passing in chapter 1, but in this section we'll discuss the issue in more depth. Let's add a new document to our list of seeding URLs. Listing 2.4 is similar to listing 2.1, but it now includes a URL that contains spam.

Listing 2.4 Reading, indexing, and searching web pages that contain spam

```
FetchAndProcessCrawler crawler =
➥ new FetchAndProcessCrawler("C:/iWeb2/data/ch02",5,200);

crawler.setDefaultUrls();

crawler.addUrl("file:///c:/iWeb2/data/ch02/spam-01.html");    // Add web page with spam

crawler.run();

LuceneIndexer luceneIndexer =
➥ new LuceneIndexer(crawler.getRootDir());    // Build Lucene index

luceneIndexer.run();

MySearcher oracle = new MySearcher(luceneIndexer.getLuceneDir());    // Build plain search engine

oracle.search("armstrong",5);
```

Figure 2.4 shows the results of the search for "Armstrong." You can see that the carefully crafted spam web page catapulted to first place in our ranking. You can create three or more similar spam pages and add them to your URL list to convince yourself that pretty soon the truly relevant content will be lost in a sea of spam pages!

Unlike a set of documents in a database or on your hard drive, the content of the Web isn't regulated. Hence, the deliberate creation of deceptive web pages can render traditional IR techniques practically useless. If search engines relied solely on traditional IR techniques then web surfing for learning or entertainment—our national online sport—wouldn't be possible. Enter a new brave world: *link analysis*! Link analysis was the first (and a significant) contribution toward fast and accurate searching on a set of documents that are linked to each other explicitly, such as internet web pages.

```
bsh % oracle.search("armstrong",5);

Search results using Lucene index scores:
Query: armstrong

Document Title: Cheap medicine--low interest loans
Document URL: file:/c:/iWeb2/data/ch02/spam-01.html --> Relevance
Score: 0.591894507408142
___________________________________________________________
Document Title: Lance Armstrong meets goal in painful marathon
debut
Document URL: file:/c:/iWeb2/data/ch02/sport-01.html -->
Relevance Score: 0.370989531278610
___________________________________________________________
Document Title: New York 'tour' Lance's toughest
Document URL: file:/c:/iWeb2/data/ch02/sport-03.html -->
Relevance Score: 0.291807949542999
_______________________________________________________
Document Title: New York City Marathon
Document URL: file:/c:/iWeb2/data/ch02/sport-02.html -->
Relevance Score: 0.210920616984367
___________________________________________________________

bsh %
```

Figure 2.4 A single deceptive web page significantly altered the ranking of the results for the query "Armstrong."

It propelled Google from anonymity to world domination in that space and advanced many other areas of research and development.

Link analysis is a structural characteristic of the internet. Another characteristic of the internet is *user click analysis*, which is behavioral. In short, user click analysis refers to the recording of the user's clicks as she navigates the search pages, and the subsequent processing of these recordings for the purpose of improving the ranking of the results for this particular user. It's based on the premise that if you search for a term and find a page that's relevant (based on your criteria) you'll most likely click on that page. Conversely, you wouldn't click pages that are irrelevant to your search term and your *search intention*. We emphasize the term because this is a deviation from traditional applications, where the response of the system was based on the user's direct input alone. If the application can detect your intentions then it has achieved a major milestone toward intelligence, which is the ability to learn about the user without the programmer entering the answer from a "back door."

2.3 Improving search results based on link analysis

In our effort to search beyond indexing, we'll present the link analysis algorithm that makes Google special—PageRank. The PageRank algorithm was introduced in 1998, at the seventh international World Wide Web conference (WWW98), by Sergey Brin and

Larry Page in a paper titled "The anatomy of a large-scale hypertextual Web search engine." Around the same time, Jon Kleinberg at IBM Almaden had discovered the *Hypertext Induced Topic Search (HITS)* algorithm. Both algorithms are link analysis models, although HITS didn't have the degree of commercial success that PageRank did.

In this section, we'll introduce the basic concepts behind the PageRank algorithm and the mechanics of calculating ranking values. We'll also examine the so-called *teleportation mechanism* and the inner workings of the *power method*, which is at the heart of the PageRank algorithm. Lastly, we'll demonstrate the combination of index scores and PageRank scores for improving our search results.

2.3.1 An introduction to PageRank

The key idea of PageRank is to consider hyperlinks from one page to another as recommendations or endorsements. So, the more endorsements a page has the higher its importance should be. In other words, if a web page is pointed to by other, important pages, then it's also an important page. Hold on a second! If you need to know what pages are important in order to determine the important pages, how does it work? Let's take a specific example and work out the details.

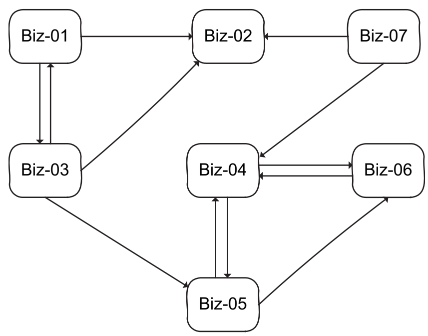

Figure 2.5 A directed graph that represents the linkage between the "biz" web pages.

Figure 2.5 shows the directed graph for all our sample web pages that start with the prefix *biz*. The titles of these articles and their file names are given in table 2.1.

If web page A has a link to web page B, there's an arrow pointing from A to B. Based on this figure, we'll introduce the *hyperlink matrix* H and a row vector p (the PageRank vector). Think of a matrix as nothing more than a table (a 2D array) and a vector as a

Table 2.1 The business news documents and their connection (see also figure 2.5)

Title	File name	Links to
Google Expands into Newspaper Ads	biz-01.html	biz-02, biz-03
Google's Sales Pitch to Newspapers	biz-02.html	(No outlink; *dangling node*)
Google Sells Newspaper Ads	biz-03.html	biz-01, biz-02, biz-05
NVidia Now a Supplier for MP3 Players	biz-04.html	biz-05, biz-06
Nvidia Shares Up on PortalPlayer Buy	biz-05.html	biz-04, biz-06
Chips Snap: Nvidia, Altera Shares Jump	biz-06.html	biz-04
Economic Stimulus Plan Helps Stock Prices	biz-07.html	biz-02, biz-04

single array in Java. Each row in the matrix H is constructed by counting the number of all the outlinks from page P_i, say $N(i)$ and assigning to column j the value $1/N(i)$ if there's an outlink from page P_i to page P_j, or assigning the value 0 otherwise. Thus, for the graph in Figure 2.5, our H matrix would look like table 2.2.

Table 2.2 The H matrix for the business news pages and their connection (see also figure 2.5)

0	1/2	1/2	0	0	0	0
0	0	0	0	0	0	0
1/3	1/3	0	0	1/3	0	0
0	0	0	0	1/2	1/2	0
0	0	0	1/2	0	1/2	0
0	0	0	1	0	0	0
0	1/2	0	1/2	0	0	0

A couple of things stand out:

- There are a lot of zeros in that matrix—we call these matrices *sparse*. That's not a curse; it's actually a good thing. It's the result of the fact that a web page typically links to only a small number of other web pages—small with respect to the total number of web pages on the internet. Sparse matrices are desirable because their careful implementation can save a lot of storage space and computational time.
- All values in the matrix are less than or equal to 1. This turns out to be very important. There's a connection between the "random" surfer that Brin and Page envisioned (see section 2.3.2) and the theory of transition probability matrices, also known as *Markov chain theory*. That connection guarantees certain desirable properties for the algorithm.

2.3.2 Calculating the PageRank vector

The PageRank algorithm calculates the vector p using the following iterative formula:

$$\mathbf{p}\ (k+1) = p\ (k) * H$$

The values of p are the PageRank values for every page in the graph. You start with a set of initial values such as $p(0) = 1/n$, where n is the number of pages in the graph, and use the formula to obtain $p(1)$, then $p(2)$, and so on, until the difference between two successive PageRank vectors is small enough; that arbitrary smallness is also known as the *convergence criterion* or *threshold*. This iterative method is the *power method* as applied to H. That, in a nutshell, is the PageRank algorithm.

For technical reasons—the *convergence* of the iterations to a *unique* PageRank vector—the matrix H is replaced by another matrix, usually denoted by G (the Google matrix), which has better mathematical properties. We won't review the mathematical details of the PageRank algorithm here, but let's describe the rationale behind PageRank and the problems that lead us to alter the matrix so that you have a better idea of what's going on.

The PageRank algorithm begins by envisioning a user who "randomly" surfs the Web. Our surfer can start from any given web page with outlinks. From there, by following one of the provided outlinks, he lands on another page. Then, he selects a new outlink to follow, and so on. After several clicks and trips through the graph, the proportion of time that our surfer spends on a given page is a measure of the relative importance that the page has with respect to the other pages on the graph. If the surfing is truly random—without an explicit bias—our surfer will visit pages that are pointed to by other pages, thus rendering those pages more important. That's all good and straightforward, but there are two problems.

The first problem is that on the internet there are some pages that don't point to any other pages; in our example, such a web page is biz-02 in figure 2.5. We call these pages of the graph *dangling nodes*. These nodes are a problem because they trap our surfer; without outlinks, there's nowhere to go! They correspond to rows that have value equal to zero for all their cells in the *H* matrix. To fix this problem, we introduce a random *jump*, which means that once our surfer reaches a dangling node, he may go to the address bar of his browser and type the URL of any one of the graph's pages. In terms of the *H* matrix, this corresponds to setting all the zeros (of a dangling node row) equal to $1/n$, where n is the number of pages in the graph. Technically, this correction of the *H* matrix is referred to as the *stochasticity adjustment*.

The second problem is that sometimes our surfer may get bored, or interrupted, and may jump to another page without following the linked structure of the web pages; the equivalent of *Star Trek*'s teleportation beam. To account for these arbitrary jumps, we introduce a new parameter that, in our code, we call `alpha`. This parameter determines the amount of time that our surfer will surf by following the links versus jumping arbitrarily from one page to another page; this parameter is sometimes referred to as the *damping factor*. Technically, this correction of the *H* matrix is referred to as the *primitivity adjustment*.

In the code, you'll find explicit annotations for these two problems. You don't need to worry about the mathematical details, but if you do, *Google's PageRank and Beyond: The Science of Search Engine Rankings* by Amy Langville and Carl Meyer is an excellent reference. So, let's get into action and get the *H* matrix by running some code. Listing 2.5 shows how to load just the web pages that belong to the business news and calculate the PageRank that corresponds to them.

Listing 2.5 Calculating the PageRank vector

```
FetchAndProcessCrawler crawler =
➥ new FetchAndProcessCrawler("C:/iWeb2/data/ch02",5,200);

crawler.setUrls("biz");     <— Load business web pages
crawler.run();
```

```
PageRank pageRank = new PageRank(crawler.getCrawlData());
pageRank.setAlpha(0.8);
pageRank.setEpsilon(0.0001);
pageRank.build();
```

Build PageRank instance

Find PageRank values

Figure 2.6 shows a screenshot of the results. The page with the lowest relevance is biz-07.html; the most important page, according to PageRank, is biz-04.html. We've calculated a measure of relevance for each page that doesn't depend on the search term! We've calculated the PageRank values for our network.

```
Iteration: 8,   PageRank convergence error:
1.4462733376210263E-4
Index: 0-->  PageRank: 0.03944811976367004
Index: 1-->  PageRank: 0.09409188129468615
Index: 2-->  PageRank: 0.32404719855854225
Index: 3-->  PageRank: 0.24328037107628753
Index: 4-->  PageRank: 0.18555028886849476
Index: 5-->  PageRank: 0.05593157626783124
Index: 6-->  PageRank: 0.061816733771795335

 Iteration: 9,   PageRank convergence error:
5.2102415715682415E-5
Index: 0-->  PageRank: 0.039443819850858625
Index: 1-->  PageRank: 0.09407831778282823
Index: 2-->  PageRank: 0.3240636997004271
Index: 3-->  PageRank: 0.24328782624042117
Index: 4-->  PageRank: 0.18555238603685822
Index: 5-->  PageRank: 0.0559269660757835
Index: 6-->  PageRank: 0.06181315844717868

_______________  Calculation Results  ________________
Page U RL: file:/c:/iWeb2/data/ch02/biz-04.html  -->  Rank:
0.324063699700427
Page URL: file:/c:/iWeb2/data/ch02/biz-06.html  -->  Rank:
0.243287826240421
Page URL: file:/c:/iWeb2/data/ch02/biz-05.html  -->  Rank:
0.185552386036858
Page URL: file:/c:/iWeb2/data/ch02/biz-02.html   -->  Rank:
0.094078317782828
Page URL: file:/c:/iWeb2/data/ch02/biz-03.html   -->  Rank:
0.061813158447179
Page URL: file:/c:/iWeb2/data/ch02/biz- 01.html   -->  Rank:
0.055926966075784
Page URL: file:/c:/iWeb2/data/ch02/biz-07.html   -->  Rank:
0.039443819850859
________________________________________________________
```

Figure 2.6 The calculation of the PageRank vector for the small network of the business news web pages

2.3.3 alpha: The effect of teleportation between web pages

Let's vary the value of `alpha` from 0.8 to some other value between 0 and 1, in order to observe the effect of the teleportation between web pages on the PageRank values. As `alpha` approaches zero, the PageRank values for all pages tends to the value 1/7 (approximately equal to the decimal value 0.142857), which is exactly what you'd expect because our surfer is choosing his next destination at random, not on the basis of the links. On the other hand, as `alpha` approaches one, the PageRank values will converge to the PageRank vector that corresponds to a surfer who closely follows the links.

Another effect you should observe as the value of `alpha` approaches one is the number of iterations, which are required for convergence, increases. In fact, for our small web page network, we have table 2.3 (we keep the error tolerance equal to 10^{-10}).

`Alpha`	Number of iterations
0.50	13
0.60	15
0.75	19
0.85	23
0.95	29
0.99	32

Table 2.3 Effect of increasing alpha values on the number of iterations for the biz set of web pages

As you can see, the number of iterations grows rapidly as the value of `alpha` increases. For seven web pages, the effect is practically insignificant, but for 8 billion pages (roughly the number of pages that Google uses), a careful selection of `alpha` is crucial. In essence, the selection of `alpha` is a trade-off between adherence to the structure of the Web and computational efficiency. The value that Google is allegedly using for `alpha` is equal to 0.85. A value between 0.7 and 0.9 should provide you with a good trade-off between effectiveness and efficiency in your application, depending on the nature of your graph and user browsing habits.

There are techniques that can accelerate the convergence of the power method as well as methods that don't rely on the power method at all, the so-called *direct methods*. The latter are more appropriate for smaller networks (such as a typical intranet) and high values of `alpha` (for example, 0.99). We'll provide references at the end of this chapter, if you're interested in learning more about these methods.

2.3.4 Understanding the power method

Let's examine the code that calculates the PageRank values in more detail. Listing 2.6 shows an excerpt of the code responsible for evaluating the matrix H based on the link information; it's from the class `iweb2.ch2.ranking.PageRankMatrixH`.

Listing 2.6 Evaluating the matrix *H* based on the links between web pages

```
public void addLink(String pageUrl) {
   indexMapping.getIndex(pageUrl);
}

public void addLink(String fromPageUrl,
➥ String toPageUrl, double weight) {

   int i = indexMapping.getIndex(fromPageUrl);
   int j = indexMapping.getIndex(toPageUrl);

   try {

       matrix[i][j] = weight;

   } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("fromPageUrl:" + fromPageUrl
➥   + ", toPageUrl: " + toPageUrl);
   }
 }

public void addLink(String fromPageUrl, String toPageUrl) {
      addLink(fromPageUrl, toPageUrl, 1);
   }

public void calculate() {

   for(int i = 0, n = matrix.length; i < n; i++) {

     double rowSum = 0;

     for(int j = 0, k = matrix.length; j < k; j++) {

        rowSum += matrix[i][j];
     }

     if( rowSum > 0 ) {

        for(int j = 0, k = matrix.length; j < k; j++) {

          if( matrix[i][j] > 0 ) {

             matrix[i][j] =
➥  (double)matrix[i][j] / (double) rowSum;
          }
        }
      } else {

        numberOfPagesWithNoLinks++;
      }
   }
}

/**
 * A dangling node corresponds to a web page that has no outlinks.
 * These nodes result in an H row that has all its values equal to 0.
 */
public int[] getDangling() {

  int   n = getSize();
  int[] d = new int[n];
```

1. **Assign initial values**
2. **Calculate substochastic version of matrix**
3. **Handle dangling node entries**

```
    boolean foundOne = false;

    for (int i=0; i < n; i++) {

      for (int j=0; j < n; j++) {

      if (matrix[i][j] > 0) {

         foundOne = true;
         break;
      }
       }

       if (foundOne) {
         d[i] = 0;
       } else {
         d[i] = 1;
       }

       foundOne = false;
    }
    return d;
}
```

1. The addLink methods allow us to assign initial values to the matrix variable, based on the links that exist between the pages.
2. The calculate method sums up the total number of weights across a row (outlinks) and replaces the existing values with their weighted counterparts. Once that's done, if we add up all the entries in a row, the result should be equal to 1 for every nondangling node. This is the substochastic version of the original matrix.
3. The dangling nodes are treated separately, since they have no outlinks. The getDangling() method will evaluate what rows correspond to the dangling nodes and will return the dangling vector.

Recall that we've separated the final matrix composition into three parts: the basic link contribution, the dangling node contribution, and the teleportation contribution. Let's see how we combine them to get the final matrix values that we'll use for the evaluation of the PageRank. Listing 2.7 shows the code that's responsible for assembling the various contributions and executing the power method. This code can be found in the iweb2.ch2.ranking.Rank class.

Listing 2.7 Applying the power method for the calculation of PageRank

```
public void findPageRank(double alpha, double epsilon) {

  // A counter for our iterations
  int k = 0;

  // auxiliary variable
  PageRankMatrixH matrixH = getH();

  // The H matrix has size nxn and the PageRank vector has size n
  int n = matrixH.getSize();

  //auxiliary variable - inverse of n
  double inv_n = (double)1/n;
```

```
// This is the actual nxn matrix of double values
double[][] H = matrixH.getMatrix();

// A dummy variable that holds our error, arbitrarily set to a value of 1
double error = 1;

// This holds the values of the PageRank vector
pR = new double[n];

// PageRank copy from the previous iteration
// The only reason that we need this is for evaluating the error
double[] tmpPR = new double[n];

// Set the initial values (ad hoc)
for (int i=0; i < n; i++) {
 pR[i] = inv_n;
}

// Book Section 2.3 -- Altering the H matrix: Dangling nodes

double[][] dNodes= getDanglingNodeMatrix();

// Book Section 2.3 -- Altering the H matrix: Teleportation

double tNodes=(1 - alpha) * inv_n;

//Replace the H matrix with the G matrix
for (int i=0; i < n; i++) {
 for (int j=0; j < n; j++) {

  H[i][j] = alpha*H[i][j] + dNodes[i][j] + tNodes;
 }
}

// Iterate until convergence!
// If error is smaller than epsilon then we've found the PageRank values
while ( error >= epsilon) {

 // Make a copy of the PageRank vector before we update it
 for (int i=0; i < n; i++) {
  tmpPR[i] = pR[i];
 }

 double dummy =0;

 // Now we get the next point in the iteration
 for (int i=0; i < n; i++) {

  dummy =0;

  for (int j=0; j < n; j++) {

   dummy += pR[j]*H[j][i];
  }

  pR[i] = dummy;
 }

 // Get the error, so that we can check convergence
 error = norm(pR,tmpPR);

 //increase the value of the counter by one
 k++;
```

```
  }

  // Report the final values
  System.out.println(
➥  "\n______________ Calculation Results _______________\n");
  for (int i=0; i < n; i++) {
    System.out.println("Page URL: "+
➥  matrixH.getIndexMapping().getValue(i)+"  --> Rank: "+pR[i]);
  }
}
```

Given the importance of this method, we've gone to great lengths to make this as easy to read as possible. We've removed some Javadoc associated with a to-do topic, but otherwise this snippet is intact. So, we start by getting the values of the matrix *H* based on the links and then initialize the PageRank vector. Subsequently, we obtain the dangling node contribution and the teleportation contribution. Note that the dangling nodes require a full 2D array, whereas our teleportation contribution requires only a single double variable. Once we have all three components, we add them together. This is the most efficient way to prepare the data for the power method, but instead of full 2D arrays, you should use sparse matrices; we describe this enhancement in one of the to-do topics at the end of the chapter.

Once the new *H* matrix has been computed, we begin the power method—the code inside the `while` loop. We know that we've attained the PageRank values if our error is smaller than the arbitrarily small value `epsilon`. Of course, that makes you wonder: What if I change `epsilon`? Will the PageRank values change? If so, what should the value of `epsilon` be? Let's take these questions one by one. First, let's say that the error is calculated as the absolute value of the term by term difference between the new and the old PageRank vectors. Listing 2.8 shows the method `norm`, from the `iweb2.ch2.ranking.Rank` class, which evaluates the error.

Listing 2.8 Evaluation of the error between two consecutive PageRank vectors

```
private double norm(double[] a, double[] b) {

  double norm = 0;

  int n = a.length;

  for (int i=0; i < n; i++) {
     norm += Math.abs(a[i]-b[i]);
  }

  return norm;
}
```

If you run the code a few times, or observe figure 2.6 closely, you'll realize that the values of the PageRank at the time of convergence change at the digit that corresponds to the smallness of `epsilon`. So, the value of `epsilon` ought to be small enough to allow us to separate all web pages according to the PageRank values. If we have 100 pages then a value of `epsilon` equal to 0.001 should be sufficient. If we have the entire internet, about 10^{10} web pages, then we need a value of epsilon that is about 10^{-10} small.

2.3.5 Combining the index scores and the PageRank scores

Now that we've showed you how to implement the PageRank algorithm, we're ready to show you how to combine the Lucene search scores with the relevance of the pages as given by the PageRank algorithm. We'll use the same seven web pages that refer to business news, but this time we'll introduce three spam pages (called spam-biz-0*x*.html, where *x* stands for a numeral). The spam pages will fool the index-based search, but they won't fool PageRank.

Let's run this scenario and see what happens. Listing 2.9 shows you how to

- Load the business web pages, as we did before.
- Add the three spam pages, one for each subject.
- Index all the pages.
- Build the PageRank.
- Compute a hybrid ranking score that incorporates both the index relevance score (from Lucene) and the PageRank score.

Listing 2.9 Combining the Lucene and PageRank scores for ranking web pages

```
FetchAndProcessCrawler crawler =
➥ new FetchAndProcessCrawler("C:/iWeb2/data/ch02",5,200);

crawler.setUrls("biz");

crawler.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-01.html");   <-- Add spam pages
crawler.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-02.html");
crawler.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-03.html");
crawler.run();

LuceneIndexer luceneIndexer =
➥ new LuceneIndexer(crawler.getRootDir());

luceneIndexer.run();                          <-- Index all pages

PageRank pageRank = new PageRank(crawler.getCrawlData());
pageRank.setAlpha(0.99);
pageRank.setEpsilon(0.00000001);
pageRank.build();                          <-- Build PageRank

MySearcher oracle = new MySearcher(luceneIndexer.getLuceneDir());

oracle.search("nvidia",5, pageRank);     <-- Search using combined score
```

The results of our search for "nvidia" are shown in figure 2.7. First, we print the result set that's based on Lucene alone, then we print the resorted results where we took into account the PageRank values. As you can see, we have a talent for spamming! The deceptive page comes first in our result set when we use Lucene alone. But when we apply the hybrid ranking, the most relevant pages come up first. The spam page went down in the abyss of irrelevance where it belongs! You've just written your first Google-like search engine. Congratulations!

The code that combines the two scores can be found in the class `MySearcher` inside the overloaded method `search` that uses the `PageRank` class as an argument.

```
bsh % oracle.search("nvidia",5,pr);

Search results using Lucene index scores:
Query: nvidia

Document Title: NVIDIA shares plummet into cheap medicine for
you!
Document URL: file:/c:/iWeb2/data/ch02/spam-biz-02.html   -->
Relevance Score: 0.519243955612183
_______________________________________________________
Document Title: Nvidia shares up on PortalPlayer buy
Document URL: file:/c:/iWeb2/data/ch02/biz-05.html
Relevance Score: 0.254376530647278
_______________________________________________________
Document Title: NVidia Now a Supplier for MP3 Players
Document URL: file:/c:/iWeb2/data/ch02/biz-04.html   -->
Relevance Score: 0.190782397985458
_______________________________________________________
Document Title: Chips Snap: Nvidia, Altera Shares Jump
Document URL: file:/c:/iWeb2/data/ch02/biz-06.html   -->
Relevance Score: 0.181735381484032

Document Title: Economic stimulus plan helps stock prices
Document URL: file:/c:/iWeb2/data/ch02/biz-07.html   -->
Relevance Score: 0.084792181849480
_______________________________________________________

Search results using combined Lucene scores and page rank scores:
Query: nvidia

Document URL: file:/c:/iWeb2/data/ch02/biz-04.html   -->
Relevance Score: 0.087211910261991
Document URL: file:/c:/iWeb2/data/ch02/biz-06.html   -->

Document URL: file:/c:/iWeb2/data/ch02/biz-05.html   -->
Relevance Score: 0.062737066556678
Document URL: file:/c:/iWeb2/data/ch02/spam-biz-02.html   -->

Document URL: file:/c:/iWeb2/data/ch02/biz-07.html   -->
Relevance Score: 0.000359708275446
_______________________________________________________
```

Figure 2.7 Combining the Lucene scores and the PageRank scores allows you to eliminate spam.

The snippet of code in listing 2.10 is from that method and captures the combination of the two scores.

Listing 2.10 Combining the Lucene scores and the PageRank scores

```
double m = 1 - (double) 1/pR.getH().getSize();     <-- Calculate scaling factor

for (int i = 0; i < numberOfMatches; i++) {

 url = docResults[i].getUrl();

 double hScore =                                          Calculate
 ➥ docResults[i].getScore() *Math.pow(pR.getPageRank(url),m);   <-- hybrid score

 docResults[i].setScore(hScore);

 urlScores.put(hScore, url);    <-- Create map between scores and URLs
}
```

Now, a number of reasonable questions may come to your mind. Why did we introduce the variable `m`? Why didn't we take the average of the two scores? Why didn't we use a more complicated formula for combining the indexing score and the PageRank score? These are good questions to ask, and the answers may surprise you. Apart from the fact that our formula retains the value of the score between 0 and 1, our selections have been arbitrary. We may as well have taken the product of the two scores in order to combine them.

The rationale for raising the PageRank value to power `m` is that the small number of pages that we've indexed may cause the relevance score of indexing to be too high for the spam pages, thus artificially diluting the effectiveness of the PageRank. As the number of pages increases, the value of the scaled PageRank (the second term of the hybrid score) tends to the original PageRank value, because `m` quickly becomes approximately equal to 1. We believe that in small networks, such a power-law scaling can help you increase the importance of the link structure over that of the index. This formula should work well for small as well as large sets of documents. There's a deep mathematical connection between power laws and graphs similar to the internet, but we won't discuss it here (see Adamic et al.). The corollary is that when you deal with a small number of pages, and if the search term appears in the document a large number of times (as it happens with spam pages), the index page score (the number that Lucene returns as the score of a search result) will be close to 1; therefore a rescaling is required to balance that effect.

2.4 Improving search results based on user clicks

In the previous section, we showed that link analysis allows us to take advantage of the structural aspects of the internet. In this section, we'll talk about a different way of leveraging the nature of the internet: user clicks. As you know, every time a user executes a query, he'll either click one of the results or click the link that shows the next page of results, if applicable. In the first case, the user has identified something of interest and clicks the link either because that's what he was looking for or because the result is interesting and he wants to explore the related information, in order to decide if it is indeed what he was looking for. In the second case, the best results weren't what the user wanted to see and he wants to look at the next page just in case the search engine is worth a dime!

Kidding aside, one reason why evaluating relevance is a difficult task is because relevance is subjective. If you and I are looking results for the query "elections," you may be interested in the U.S. elections, while I may be interested in the UK elections, or even in my own town's elections. It's impossible for a search engine to know the intention (or the context) of your search without further information. So, the most relevant results for one person can be, and quite often are, different from the most relevant results for another person, even though the query terms may be identical!

We're going to introduce user clicks as a way of improving the search results for each user. This improvement is possible due to an algorithm that we'll study in great detail later in the book—the `NaiveBayes` classifier. We'll demonstrate the combination of index scores, PageRank scores, and the scores from the user clicks for improving our search results.

2.4.1 A first look at user clicks

User clicks allow us to take as input the interaction of each user with the search engine. Aristotle said, "We are what we repeatedly do," and that's the premise of user clicks analysis: your interaction with the search engine defines your own areas of interest and your own subjectivity. This is the first time that we describe an intelligent technique responsible for the *personalization* of a web application. Of course, a necessary condition for this is that the search engine can identify which queries come from a particular user. In other words, the user must be logged in to your application or must have otherwise established a session with the application. It should be clear that our approach for user-click analysis is applicable to every application that can record the user's clicks, and it's not specific to search applications.

Now, let's assume that you've collected the clicks of the users as indicated in the file user-clicks.csv, which you can find in the `data/ch02` directory together with the rest of the files that we've been using in this chapter. Our goal is to write code that can help us leverage that information, much like the PageRank algorithm helped us to leverage the information about our network. That is, we want to use this data to *personalize* the results of the search by appropriately modifying the ranking, depending on who submits the query. The comma separated file contains values in three fields:

- A string that identifies the user
- A string that represents the search query
- A string that contains the URL that the user has selected in the past, after reviewing the results for that query

If you don't know the user (no login/no session of any kind), you can use some default value such as "anonymous"—of course, you should ensure that anonymous isn't actually a valid username in your application! If your data has some other format, it's okay. You shouldn't have any problems adopting our code for your specific data. In order to personalize our results, we need to know the user, her question, and her past selections of links for that question. If you have that information available then you should be ready to get in action!

You may notice that, in our data, for the same user and the same query there is more than one entry. That's normal and you should notice it in your data as well. The number of times that a click appears in that file makes its URL a better or worse candidate for our search results. Typically, the same user will click a number of different links for the same query because his interest at the time may be different or because he may be looking for additional information on a topic. An interesting attribute that you should consider is a *timestamp*. Time-related information can help you identify temporal structure in your data. Some user clicks follow periodic patterns; some are event-driven; others are completely random. A timestamp can help you identify the patterns or the correlations with other events.

First let's see how we can obtain personalized results for our queries. Listing 2.11 shows our script, which is similar to listing 2.9, but this time we load the information about the user clicks and we run the same query "google ads" twice, once for user dmitry and once for user babis.

Listing 2.11 Accounting for user clicks in the search results

```
FetchAndProcessCrawler crawler =
➥ new FetchAndProcessCrawler("C:/iWeb2/data/ch02",5,200);

crawler.setUrls("biz");
crawler.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-01.html");
crawler.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-02.html");
crawler.addUrl("file:///c:/iWeb2/data/ch02/spam-biz-03.html");
crawler.run();

LuceneIndexer luceneIndexer =
➥ new LuceneIndexer(crawler.getRootDir());

luceneIndexer.run();
MySearcher oracle = new MySearcher(luceneIndexer.getLuceneDir());

PageRank pageRank = new PageRank(crawler.getCrawlData());
pageRank.setAlpha(0.9);
pageRank.setEpsilon(0.00000001);
pageRank.build();
                                                        Load user clicks
UserClick aux = new UserClick();
UserClick[] clicks =aux.load("C:/iWeb2/data/ch02/user-clicks.csv");

TrainingSet tSet = new TrainingSet(clicks);     Create training set

NaiveBayes naiveBayes = new NaiveBayes("Naïve Bayes", tSet);     Define
                                                                 classifier
naiveBayes.trainOnAttribute("UserName");        Select
naiveBayes.trainOnAttribute("QueryTerm_1");     attributes
naiveBayes.trainOnAttribute("QueryTerm_2");

naiveBayes.train();                       Train classifier

oracle.setUserLearner(naiveBayes);

UserQuery dmitryQuery = new UserQuery("dmitry","google ads");
oracle.search(dmitryQuery,5, pageRank);

UserQuery babisQuery = new UserQuery("babis","google ads");
oracle.search(babisQuery,5, pageRank);
```

You've seen the first part of this script in listing 2.9. First, we load the pages that we want to search. After that, we index them with Lucene and build the PageRank that corresponds to their structure. The part that involves new code comes with the class `UserClick`, which represents the click of a specific user on a particular URL. We also defined the class `TrainingSet`, which holds all the user clicks. Of course, you may wonder, what's wrong with the array of `UserClicks`? Why can't we just use these objects? The answer lies in the following: in order to determine the links that are more likely to be desirable for a particular user and query, we're going to load the user clicks onto a *classifier*—in particular, the `NaiveBayes` classifier.

2.4.2 Using the NaiveBayes classifier

We'll address classification extensively in chapters 5 and 6, but we'll describe fundamentals here for clarity. Classification relies on reference structures that divide the space of all possible data points into a set of classes (also known as *categories* or *concepts*) that are (usually) non-overlapping. We encounter classification on a daily basis. From our everyday experience, we know that we can list food items according to a restaurant's menu, for example salads, appetizers, specialties, pastas, seafood, and so on. Similarly, the articles in a newspaper, or in a newsgroup on the internet, are classified based on their subject—politics, sports, business, world, entertainment, and so on. In short, we can say that classification algorithms allow us to automatically identify objects as part of this or that class.

In this section, we'll use a probabilistic classifier that implements what's known as the *naïve Bayes algorithm*; our implementation is provided by the `NaiveBayes` class. Classifiers are agnostic to `UserClicks`, they're only concerned with `Concepts`, `Instances`, and `Attributes`. Think of `Concepts`, `Instances`, and `Attributes` as the analogues of directories, files, and file attributes on your filesystem.

A classifier's job is to assign a `Concept` to an `Instance`; that's all a classifier does. In order to know what `Concept` should be assigned to a particular `Instance`, a classifier reads a `TrainingSet`—a set of `Instances` that already have a `Concept` assigned to them. Upon loading those `Instances`, the classifier *trains* itself, or *learns,* how to map a `Concept` to an `Instance` based on the assignments in the `TrainingSet`. The way that each classifier trains depends on the classifier.

Our intention is to use the `NaiveBayes` classifier as a means of obtaining a relevance score for a particular URL based on the user and submitted query. The good thing about the `NaiveBayes` classifier is that it provides something called the *conditional probability* of X given Y—a probability that tells us how likely is it to observe event X provided that we've already observed event Y. In particular, this classifier uses as input the following:

- The probability of observing concept X, in general, also known as the *prior* probability and denoted by $p(X)$.
- The probability of observing instance Y if we randomly select an instance from concept X, also known as the *likelihood* and denoted by $p(Y|X)$.
- The probability of observing instance Y in general, also known as the *evidence* and denoted by $p(Y)$.

The essential part of the classifier is the calculation of the probability that an observed instance *Y* belongs in concept *X*, which is also known as the *posterior probability* and denoted by $p(X|Y)$. The calculation is performed based on the following formula (known as Bayes theorem):

$$p(X|Y) = p(Y|X)\, p(X) \,/\, p(Y)$$

The `NaiveBayes` classifier can provide a measure of how likely it is that user *A* wants to see URL *X* provided that she submitted query *Q*; in our case, $Y = A + Q$. In other words, we won't use the `NaiveBayes` classifier to classify anything. We'll only use its capacity to produce a measure of relevance, which exactly fits our purposes. Listing 2.12 shows the relevant code from the class `NaiveBayes`; for a complete description, see section 5.3.

Listing 2.12 Evaluating the relevance of a URL with the `NaiveBayes` classifier

```
public class NaiveBayes implements Classifier {
  private String name;                                    ❶
  private TrainingSet tSet;                                 ❷

  private HashMap<Concept,Double> conceptPriors;            ❸

  protected Map<Concept,Map<Attribute, AttributeValue>> p;       ❹

  private ArrayList<String> attributeList;        ❺

  public double getProbability(Concept c, Instance i) {
    double cP=0;
    if (tSet.getConceptSet().contains(c)) {

     cP = (getProbability(i,c)*getProbability(c))/getProbability(i);     ❻
    } else {

     cP = 1/(tSet.getNumberOfConcepts()+1);     ❼
    }
    return cP;
  }

  public double getProbability(Instance i) {
    double cP=0;

    for (Concept c : getTset().getConceptSet()) {

     cP += getProbability(i,c)*getProbability(c);
    }
    return (cP == 0) ? (double)1/tSet.getSize() : cP;     ❽
  }

  public double getProbability(Concept c) {
    Double trInstanceCount = conceptPriors.get(c);
    if( trInstanceCount == null ) {
       trInstanceCount = 0.0;
    }
    return trInstanceCount/tSet.getSize();     ❾
  }

  public double getProbability(Instance i, Concept c) {
    double cP=1;
    for (Attribute a : i.getAtrributes()) {
```

```
      if ( a != null && attributeList.contains(a.getName()) ) {

       Map<Attribute, AttributeValue> aMap = p.get(c);
       AttributeValue aV = aMap.get(a);
       if ( aV == null) {                                    ❿
            cP *= ((double) 1 / (tSet.getSize()+1));
       } else {
         cP *= (double)(aV.getCount()/conceptPriors.get(c));
       }
      }
     }
     return (cP == 1) ? (double)1/tSet.getNumberOfConcepts() : cP;
   }
 }
```

First, let's examine the main points of the listing:

❶ This is a name for this instance of the `NaiveBayes` classifier.

❷ Every classifier needs a training set. The name of the classifier and its training set are intentionally set during the Construction phase. Once you've created an instance of the `NaiveBayes` classifier, you can't set its `TrainingSet`, but you can always get the reference to it and add instances.

❸ The `conceptPriors` map stores the counts for each of the concepts that we have in our training set. We could've used it to store the *prior* probabilities, not just the counts. But we want to reuse these counts, so in the name of computational efficiency, we store the counts; the priors can be obtained by a simple division.

❹ The variable `p` stores the conditional probabilities—the probability of observing concept *X* given that we observed instance *Y*, or in the case of the user clicks, the probability that a user *A* wants to see URL *X* provided that he submitted query *Q*.

❺ This is the list of attributes that should be considered by the classifier for training. The instances of a training set may have many attributes and it's possible that only a few of these attributes are relevant (see chapter 5), so we keep track of what attributes should be used.

❻ If we've encountered the concept in our training set, use the formula that we mentioned earlier and calculate the posterior probability.

❼ It's possible that we haven't encountered a particular instance before, so the `getProbability(i)` method call wouldn't be meaningful. In that case, we assign something reasonable as a *posterior* probability. Setting that value equal to one over the number of all known concepts is reasonable, in the absence of information for assigning higher probability to any one concept. We've also added unity to that number. That's an arbitrary modification, intended to lower the probability assigned to each concept, especially for a small number of observed concepts. Think about why, and under what conditions, this can be useful.

❽ This method of the `NaiveBayes` class isn't essential for the pure classification problem because its value is the same for all concepts. In the context of this example, we decided to keep it. Feel free to modify the code so that you get back only the numerator of the Bayes theorem; what do your results look like?

❾ The prior probability for a given concept *c* is evaluated based on the number of times that we encountered this concept in the training set. Note that we arbitrarily assign probability zero to unseen concepts. This can be good and bad. If you're pretty confident that you have all related concepts in your training set then this ad hoc choice helps you eliminate flukes in your data. In a more general case, where you might not have seen a lot of concepts, you should replace the zero value with something more reasonable—one over the total number of known concepts. What other choices do you think are reasonable? Is it important to have a sharp estimate of that quantity? Regardless of your answer, try to rationalize your decision and justify it as best as you can.

❿ We arrive at the heart of the `NaiveBayes` class. The "naïve" part of the Bayes theorem is the fact that we evaluate the likelihood of observing `Instance i`, as the product of the probabilities of observing each of the attribute values. That assumption implies that the attributes are statistically independent. We used quotes around the word *naïve* because the naïve Bayes algorithm is very robust and widely applicable, even in problems where the attribute independence assumption is clearly violated. It can be shown that the naïve Bayes algorithm is optimal in the exact opposite case—cases in which there's a completely deterministic dependency among the attributes (see Rish).

If you recall the script in listing 2.11, we've created a training set and an instance of the classifier with that training set, and before we assign the classifier to the `MySearcher` instance, we do the following two things:

- We tell the classifier what attributes should be taken into account for training purposes.
- We tell the classifier to train itself on the set of user clicks that we just loaded and for the attributes that we specified.

The attribute with label `UserName` corresponds to the user. The attributes `QueryTerm_1` and `QueryTerm_2` correspond to the first and second term of the query, respectively. These terms are obtained by using Lucene's `StandardAnalyzer` class. During training, we're assigning probabilities based on the frequency of occurrence for each instance. The important method, in our context, is `getProbability(Concept c, Instance i)`, which we'll use to obtain the relevance of a particular URL (`Concept`) when a specific user executes a specific query (`Instance`).

2.4.3 Combining Lucene indexing, PageRank, and user clicks

Armed with the probability of a user preferring a particular URL for a given query, we can proceed and combine all three techniques to obtain our enhanced search results. The relevant code is shown in listing 2.13.

Listing 2.13 Lucene indexing, PageRank values, and user click probabilities

```
public SearchResult[] search(UserQuery uQuery,
➥ int numberOfMatches, Rank pR) {

 SearchResult[] docResults =
➥ search(uQuery.getQuery(), numberOfMatches);      <— Results based on index
```

```
String url;

StringBuilder strB = new StringBuilder();

int docN = docResults.length;
                                                    Collect at most
if (docN > 0) {                          numberOfMatches documents

  int loop = (docN<numberOfMatches) ? docN : numberOfMatches;

   for (int i = 0; i < loop; i++) {

     url = docResults[i].getUrl();                  Collect all user
                                                    click scores
     UserClick uClick = new UserClick(uQuery,url);

       double indexScore = docResults[i].getScore();

     double pageRankScore  = pR.getPageRank(url);

     BaseConcept bC = new BaseConcept(url);

     double userClickScore = learner.getProbability(bC, uClick);

     double hScore;                      Evaluate final
                                         (hybrid) score
     if (userClickScore == 0) {

       hScore = indexScore * pageRankScore * EPSILON;

     } else {

         hScore = indexScore * pageRankScore * userClickScore;
        }

        docResults[i].setScore(hScore);

   strB.append("Document URL   : ")
➥ .append(docResults[i].getUrl()).append("  -->  ");
   strB.append("Relevance Score: ")
➥ .append(docResults[i].getScore()).append("\n");
  }
 }
 strB.append(PRETTY_LINE);
 System.out.println(strB.toString());

 return docResults;
}
```

Figure 2.8 shows the results for user dmitry. As you can see, due to the fact that dmitry clicked several times on the page biz-03.html in the past, the relevance score for that page is the highest. The second best hit is page biz-01.html, which is also in the user clicks file. The spam page appears third, but that's a side effect of the small number of pages; we intentionally didn't include our scaling `m` factor to demonstrate its impact on the results.

In figure 2.9, we execute the same query—"google ads"—but this time we do it as user babis. We've reversed the order of dmitry's clicks to create the clicks for the user babis. The results show that the first hit is page biz-01.html; page biz-03.html is second. Everything else is the same. The only difference in the result set comes from the fact that the query was executed by different users, and that difference reflects exactly what the application *learned* from the file user-clicks.csv.

```
bsh % UserQuery dQ = new UserQuery("dmitry", "google ads");
bsh % oracle.search(dQ,5,pr);

Search results using Lucene index scores:
Query: google ads

Document Title: Google Ads and the best drugs
Document URL: file:/c:/iWeb2/data/ch02/spam-biz-01.html -->
Relevance Score: 0.788674294948578
_______________________________________________________
Document Title: Google Expands into Newspaper Ads
Document URL: file:/c:/iWeb2/data/ch02/biz-01.html -->
Relevance Score: 0.382
_______________________________________________________
Document Title: Google sells newspaper ads
Document URL: file:/c:/iWeb2/data/ch02/biz-03.html -->
Relevance Score: 0.317
_______________________________________________________
Document Title:Google's sales pitch to newspapers
Document URL: file:/c:/iWeb2/data/ch02/biz-02.html -->
Relevance Score: 0.291
_______________________________________________________
Document Title: Economic stimulus plan helps stock prices
Document URL: file:/c:/iWeb2/data/ch02/biz-07.html -->
Relevance Score: 0.031
_______________________________________________________

Search results using combined Lucene scores, page rank scores and
user clicks:
Query: user=dmitry, query text=google ads

Document URL: file:/c:/iWeb2/data/ch02/biz-03.html   -->
Relevance Score: 0.0057

Document URL: file:/c:/iWeb2/data/ch02/biz-01.html   -->
Relevance Score: 0.0044

Document URL: file:/c:/iWeb2/data/ch02/spam-biz-01.html-->
Relevance Score: 0.0040

Document URL: file:/c:/iWeb2/data/ch02/biz-02.html   -->
Relevance Score: 0.0012

Document URL: file:/c:/iWeb2/data/ch02/biz-07.html   -->
Relevance Score: 0.0002
_______________________________________________________
```

Figure 2.8 Combining Lucene, PageRank, and user clicks to produce high-relevance search results for dmitry.

```
bsh % UserQuery bQ = new UserQuery("babis", "google ads");
bsh % oracle.search(bQ,5,pr);

Search results using Lucene index scores:
Query: google ads

Document Title: Google Ads and the best drugs
Document URL: file:/c:/iWeb2/data/ch02/spam-biz-01.html -->
Relevance Score: 0.788674294948578
___________________________________________________________
Document Title: Google Expands into Newspaper Ads
Document URL: file:/c:/iWeb2/data/ch02/biz-01.html -->
Relevance Score: 0.382
___________________________________________________________
Documen t Title: Google sells newspaper ads
Document URL: file:/c:/iWeb2/data/ch02/biz-03.html -->
Relevance Score: 0.317
___________________________________________________________
Document Title: Google's sales pitch to newspapers
Document URL: file:/c:/iWeb2/data/ch02/biz-02.html -->
Relevance Score: 0.291
___________________________________________________________
Document Title: Economic stimulus plan helps stock prices
Document URL: file:/c:/iWeb2/data/ch02/biz-07.html -->
Relevance Score: 0.0314
___________________________________________________________

Search results using combined Lucene scores, page rank scores
and user clicks:
Query: user=babis, query text=google ads

Document URL: file:/c:/iWeb2/data/ch02/biz-01.html    -->
Relevance Score: 0.00616

Document URL: file:/c:/iWeb2/data/ch02/biz-03.html    -->
Relevance Score: 0.00407

Document URL: file:/c:/iWeb2/data/ch02/spam-biz-01.html -->
Relevance Score: 0.00393

Document URL: file:/c:/iWeb2/data/ch02/biz-02.html    -->
Re levance Score: 0.00117
```

Figure 2.9 Lucene, PageRank, and user clicks together produce high-relevance search results for Babis.

That's great! We now have a powerful improvement over the pure index-based search that accounts for the structure of the hyperlinked documents and the preferences of the users based on their clicks. But a large number of applications must search among documents that aren't explicitly linked to each other. Is there anything that we can do to improve our search results in that case? Let's examine exactly that case in what follows.

2.5 Ranking Word, PDF, and other documents without links

Let's say that you have hundreds of thousands of Word or PDF documents, or any other type of document that you want to search through. At first, it may seem that indexing is your only option and, at best, you may be able to do some user-click analysis too. But we'll show you that it's possible to extend the same ideas of link analysis that we applied to the Web. Hopefully, we'll get you thinking and develop an even better method. By the way, to the best of our knowledge, the technique that we describe here has never been published before.

To demonstrate that it's possible to introduce ranking in documents without links, we'll take the HTML documents and create Word documents with identical content. This willl allow us to compare our results with those in section 2.3 and identify any similarities or differences in the two approaches. Parsing Word documents can be done easily using the open source library *TextMining*; note that the name has changed to *tm-extractor*. The license of this library starting with the 1.0 version is LGPL, which makes it business friendly. You can obtain the source code from http://code.google.com/p/text-mining/source/checkout. We've written a class called `MSWordDocumentParser` that encapsulates the parsing of a Word document in that way.

2.5.1 An introduction to DocRank

In listing 2.14 we use the same classes to read the Word documents as we did to read the HTML documents (the `FetchAndProcessCrawler` class) and we use Lucene to index the content of these documents.

Listing 2.14 Ranking documents based on content

```
FetchAndProcessCrawler crawler =
➥ new FetchAndProcessCrawler("C:/iWeb2/data/ch02",5,200);

crawler.setUrls("biz-docs");

crawler.addDocSpam();
crawler.run();

LuceneIndexer luceneIndexer =
➥ new LuceneIndexer(crawler.getRootDir());

luceneIndexer.run();

MySearcher oracle = new MySearcher(luceneIndexer.getLuceneDir());
oracle.search("nvidia",5);

DocRank docRank = new DocRank(luceneIndexer.getLuceneDir(),7);
```

Load business Word documents

Build Lucene index

Create plain search engine

```
docRank.setAlpha(0.9);
docRank.setEpsilon(0.00000001);
docRank.build();                      Create DocRank engine

oracle.search("nvidia",5, docRank);
```

Figure 2.10 shows that a search for "nvidia" returns as the highest ranked result the undesirable spam-biz-02.doc file—a result similar to the case of the HTML documents. Of course, in the case of Word, PDF, and other text documents, the chance of having spam documents is fairly low, but you could have documents with unimportant repetitions of terms in them.

So far, everything has been the same as in listing 2.9. The new code is invoked by the class `DocRank`. That class is responsible for creating a measure of relevance between documents that's equivalent to the relevance which PageRank assigns between web pages. Unlike the `PageRank` class, it takes an additional argument whose role we'll explain later on. Similar to the previous sections, we want to have a matrix that represents the importance of page *Y* based on page *X*. Our problem is that, unlike with web pages, we don't have an explicit linkage between our documents. Those web links were only used to create a matrix whose values told us how important page *Y* is according to page *X*. If we could find a way to assign a measure of importance for document *Y* according to document *X* we could use the same mathematical theory that underpins the PageRank algorithm. Our code provides such a matrix.

```
bsh % oracle.search("nvidia", 5);

Search results using Lucene index sco
Query: nvidia

Document Title: NVIDIA shares plummet into cheap medicine for
you!
Document URL: file:/c:/iWeb2/data/ch02/spam-biz-02.doc -->
Relevance Score: 0.458221405744553
_____________________________________________________________
Document Title: Nvidia shares up on PortalPlayer buy
Document URL: file:/c:/iWeb2/data/ch02/biz-05.doc   -->
Relevance Score: 0.324011474847794
_____________________________________________________________
Document Title: NVidia Now a Supplier for MP3 Players
Document URL: file:/c:/iWeb2/data/ch02/biz-04.doc  -->
Relevance Score: 0.194406896829605
_____________________________________________________________
Document Title: Nov. 6, 2006, 2:38PM?Chips Snap: Nvidia, Altera
Shares Jump
Document URL: file:/c:/iWeb2/data/ch02/biz-06.doc  -->
Relevance Score: 0.185187965631485
_____________________________________________________________
```

Figure 2.10 Index based searching for "nvidia" in the Word documents that contain business news and spam

2.5.2 The inner workings of DocRank

Our measure of importance is to a large degree arbitrary, and its viability depends crucially on two properties that are related to the elements of our new *H* matrix. The elements of that matrix should be such that:

- They are all positive numbers.
- The sum of the values in any row is equal to 1.

Whether our measure will be successful depends on the kind of documents that we're processing. Listing 2.15 shows the code from class `DocRankMatrixBuilder` that builds matrix *H* in the case of our Word documents.

Listing 2.15 DocRankMatrixBuilder: Ranking text documents based on content

```
public class DocRankMatrixBuilder implements CrawlDataProcessor {
   private final int TERMS_TO_KEEP = 3;

   private int termsToKeep=0;
   private String indexDir;
   private PageRankMatrixH matrixH;

   public void run() {
      try {
         IndexReader idxR =
➥  IndexReader.open(FSDirectory.getDirectory(indexDir));
         matrixH = buildMatrixH(idxR);
      }
      catch(Exception e) {
         throw new RuntimeException("Error: ", e);
      }
   }

   // Collects doc ids from the index for documents with matching doc type
   private List<Integer> getProcessedDocs(IndexReader idxR)
      throws IOException {
      List<Integer> docs = new ArrayList<Integer>();
      for(int i = 0, n = idxR.maxDoc(); i < n; i++) {
         if( idxR.isDeleted(i) == false ) {
            Document doc = idxR.document(i);
            if( eligibleForDocRank(doc.get("doctype") ) ) {
               docs.add(i);
            }
         }
      }
      return docs;
   }

// Is the index entry eligible?

   private boolean eligibleForDocRank(String doctype) {
      return ProcessedDocument.DOCUMENT_TYPE_MSWORD
➥     .equalsIgnoreCase(doctype);
   }

   private PageRankMatrixH buildMatrixH(IndexReader idxR)
```

```
    throws IOException {

  // consider only URLs with fetched and parsed content
    List<Integer> allDocs = getProcessedDocs(idxR);

    PageRankMatrixH docMatrix =
  new PageRankMatrixH( allDocs.size() );

    for(int i = 0, n = allDocs.size(); i < n; i++) {

        for(int j = 0, k = allDocs.size(); j < k; j++) {

             double similarity = 0.0d;

          Document docX = idxR.document(i);
             String xURL= docX.get("url");

             if ( i == j ) {

              // Avoid shameless self-promotion ;-)
              docMatrix.addLink(xURL, xURL, similarity);

             } else {

              TermFreqVector x =
  idxR.getTermFreqVector(i, "content");
              TermFreqVector y =
  idxR.getTermFreqVector(j, "content");

              similarity = getImportance(x.getTerms(),
  x.getTermFrequencies(), y.getTerms(), y.getTermFrequencies());

              // add link from docX to docY
              Document docY = idxR.document(j);
              String yURL = docY.get("url");

              docMatrix.addLink(xURL, yURL, similarity);
          }
        }
    }
    docMatrix.calculate();

    return docMatrix;
}

// Calculates importance of document Y in the context of document X
private double getImportance(String[] xTerms, int[] xTermFreq,
                       String[] yTerms, int[] yTermFreq){

  // xTerms is an array of the most frequent terms for first document
    Map<String, Integer> xFreqMap =
  buildFreqMap(xTerms, xTermFreq);

   // yTerms is an array of the most frequent terms for second document
    Map<String, Integer> yFreqMap =
  buildFreqMap(yTerms, yTermFreq);

    // sharedTerms is the intersection of the two sets
    Set<String> sharedTerms =
  new HashSet<String>(xFreqMap.keySet());
    sharedTerms.retainAll(yFreqMap.keySet());
```

```
        double sharedTermsSum = 0.0;

        // Note that this isn't symmetrical.
        // If you swap X with Y then you get a different value;
        // unless the frequencies are equal, of course!

        double xF, yF;
        for(String term : sharedTerms) {

            xF = xFreqMap.get(term).doubleValue();
            yF = yFreqMap.get(term).doubleValue();

            sharedTermsSum += Math.round(Math.tanh(yF/xF));
        }

        return sharedTermsSum;
    }

    private Map<String, Integer> buildFreqMap(String[] terms, int[] freq) {

int topNTermsToKeep = (termsToKeep == 0)? TERMS_TO_KEEP: termsToKeep;

Map<String, Integer> freqMap =
➥ TermFreqMapUtils.getTopNTermFreqMap(terms, freq, topNTermsToKeep);

        return freqMap;
    }
}
```

There are two essential ingredients in our solution. First, note that we use the Lucene *term vectors*, which are pairs of terms and their frequencies. If you recall our discussion about indexing documents with Lucene, we mentioned that the text of a document is first parsed, then analyzed before it's indexed. During the analysis phase, the text is dissected into *tokens* (terms); the way that the text is tokenized depends on the analyzer that's used. The beautiful thing with Lucene is that we can retrieve that information later on and use it. In addition to the terms of the text, Lucene also provides us with the number of times that each term appears in a document. That's all we need from Lucene: a set of terms and their frequency of occurrence in each document.

The second ingredient of our solution is the choice of assigning importance to each document. The method `getImportance` in listing 2.15 shows that, for each document *X*, we calculate the importance of document *Y* by following two steps: (1) we find the intersection between the most frequent terms of document *X* and the most frequent terms of document *Y* and (2) for each term in the set of shared terms (intersection), we calculate the ratio of the number of times the term appears in document *Y* (Y-frequency of occurrence) over the number of times the term appears in document *X* (X-frequency of occurrence). The importance of document *Y* in the context of document *X* is given as the sum of all these ratios and filtered by the hyperbolic tangent function (`Math.tanh`) as well as the rounding function (`Math.round`). The end result of these operations will be the entry in the *H* matrix for row *X* and column *Y*.

We use the hyperbolic tangent function because we want to gauge whether a particular term between the two documents should be considered a good indicator for assigning importance. We aren't interested in the exact value; we're interested only in keeping the importance factor within reasonable limits. The hyperbolic tangent takes values between 0 and 1, so the final rounding will ensure that each term can either be neglected or count for one unit of importance. That's the rationale behind building the formula by using these functions.

Figure 2.11 shows that a search for "nvidia" returns the file biz-05.doc as the highest-ranked result; that's a legitimate file (not spam) and related to nvidia! The spam

```
bsh % oracle.search("nvidia",5,dr);

Search results using Lucene index scores:
Query: nvidia

Document Title: NVIDIA shares plummet into cheap medicine for
you!

Document URL: file:/c:/iWeb2/data/ch02/spam-biz-02.doc  -->
Relevance Score: 0.4582
____________________________________________________________
Document Title: Nvidia shares up on PortalPlayer buy

Document URL: file:/c:/iWeb2/data/ch02/biz-05.doc  -->
Relevance Score: 0.3240
____________________________________________________________
Document Title: NVidia Now a Supplier for MP3 Players

Document URL: file:/c:/iWeb2/data/ch02/biz-04.doc  -->
Relevance Score: 0.1944
____________________________________________________________
Document Title: Chips Snap: Nvidia, Altera Shares Jump

Document URL: file:/c:/iWeb2/data/ch02/biz-06.doc  -->
Relevance Score: 0.1852
________________________________________________________

Search results using combined Lucene scores and page rank scores:
Query: nvidia

Document URL: file:/c:/iWeb2/data/ch02/biz-05.doc  -->
Relevance Score: 0.03858

Document URL: file:/c:/iWeb2/data/ch02/spam-biz-02.doc  -->
Relevance Score: 0.03515

Document URL: file:/c:/iWeb2/data/ch02/biz-04.doc  -->
Relevance Score: 0.02925

Document URL: file:/c:/iWeb2/data/ch02/biz-06.doc  -->
Relevance Score: 0.02233
____________________________________________________________
```

Figure 2.11 Index and ranking based search for "nvidia" on the Word documents

page survived because the number of our documents is small, but we did get additional value. The Lucene index had the exact same information all along, but its metric of relevance has been skewed by the ersatz news document. DocRank helped us to increase the relevance of the biz-05.doc document, and in more realistic situations it can help you identify the most pertinent documents in a collection. The DocRank values, like the PageRank values, need to be calculated only once, but can be reused for all queries.

There are other means of enhancing the search of plain documents, and we provide the related references at the end of this chapter. DocRank is a more powerful algorithm when applied to data from a relational database. To see this, let's say that we have two tables—table *A* and table *B*—that are related to each other through table *C*; this is a common case. For example, you may have a table that stores users, another table that stores groups, and another that stores the relationship between users and groups by relating the IDs of each entry. In effect, you have one graph that connects the users based on their groups and another graph that connects the groups based on their users. Every time you have a linked representation of entities, it's worthwhile to try the DocRank algorithm or a similar variant. Don't be afraid to experiment! There's no single answer to this kind of problem, and sometimes the answer may surprise you.

2.6 Large-scale implementation issues

Everything that we've discussed so far can be used across the functional areas and the various domains of web applications. But if you're planning to process vast amounts of data, and you have the computational resources to do it, you're going to face issues that fall largely into two categories. The first category is related to the mathematical properties of the algorithms; the second is related to the software engineering aspects of manipulating data on the scale of terabytes or even petabytes!

The first symptom of large-scale computing constraints is the lack of addressable memory. In other words, your data is so large that the data structures don't fit in memory anymore; that would be particularly true for an interpreted language, like Java, because even if you manage to fit the data, you'd probably have to worry about garbage collection. In large-scale computing, there are two basic strategies for dealing with that problem. The first is the construction of more efficient data structures, so that the data does fit in memory; the second is the construction of efficient, distributed, I/O infrastructure for accessing and manipulating the data *in situ*. For very large datasets, with sizes similar to what Google handles, you should implement both strategies because you want to squeeze every bit of efficiency out of your system.

In terms of representing data more efficiently, consider the structures that we used for storing the *H* matrix. The part of the original link structure required a `double[n][n]` and the part of the dangling node matrix required another `double[n][n]`, where `n` is the number of pages (or documents for DocRank). If you think about it, that's a huge waste of resources when `n` is very large, because most of these `double` values are zero. A more efficient way to store that information would be by means of an *adjacency list.* In Java, you can easily implement an adjacency list using a `Hashtable` that will contain `HashSets`. So, the definition of the variable `matrix` in the class `PageRankMatrixH` would look as follows:

```
Hashtable<Integer, HashSet<Integer,Double>> matrix;
```

One of the exercises that we propose is to rewrite our algorithmic implementation using these efficient structures. You could even compress the data in the adjacency list by *reference encoding* or other techniques (see Boldi and Vigna). Reference encoding relies on the similarity of web pages and sacrifices simplicity of implementation for memory efficiency.

Another implementation aspect for large-scale searching is the accuracy that you're going to have for the PageRank values (or any other implementation of the `Rank` base class). To differentiate between values of the PageRank for any two web pages among N, you'll need a minimum of $1/N$ accuracy in your numerical calculation. So, if you deal with $N = 1000$ pages then even 10^{-4} accuracy should suffice. If you want to get the rankings of billions of pages, the accuracy should be on the order of 10^{-10} for the PageRank values.

Consider a situation where the dangling nodes make up a large portion of your fetched web pages. This could happen if you want to build a dedicated search engine for a central site such as the Apache set of projects, or something less ambitious such as the Jakarta project alone. Brin and Page realized that handling a large number of nodes that are, in essence, artificial—because their entries in the H matrix don't reflect the link structure of the web but rather help the matrix to conform with certain nice mathematical properties—isn't going to be very efficient. They suggested you could remove the dangling nodes during the computation of the PageRank, and add them back after the values of the remaining PageRanks have converged sufficiently.

We don't know, of course, the actual implementation of the Google search engine—such secrets are closely guarded—but we can say with certainty that an equitable treatment of all pages will require inclusion of the dangling nodes from the beginning to the end of the calculation of PageRank. In an effort to be both fair and efficient, we can use methods that rely on the symmetric reordering of the H matrix. These techniques appear to converge at the same rate as the original PageRank algorithm while acting on a smaller problem, which means that you can have significant gains in computational time; for more details see *Google's PageRank and Beyond: The Science of Search Engine Rankings.*

Implicit in all discussions with respect to large-scale computations of search are concerns about memory and speed. One speed factor is the number of iterations for the power method, which as we've seen depends on the value of `alpha` as well as the number of the linked pages. Unfortunately, in practitioner's books similar to ours, we found statements asserting that the initial value of the PageRank vector doesn't matter and that you could set all the values equal to 1. Strictly speaking, that's not true and it can have dramatic implications when you work with large datasets whose composition changes periodically. The closer the initial vector is to the unique PageRank values, the fewer the number of iterations required. A number of techniques, known collectively as *approximate aggregation techniques,* to compute the PageRank vector of a smaller matrix in order to generate an estimate of the true updated distribution of the PageRank vector. That estimate, in turn, will be used as the initial vector for the final computation. The mathematical underpinnings of these methods won't be covered in this book. For more information on these techniques, see the references at the end of this chapter.

While we're discussing acceleration techniques for the computation of the PageRank vector, we should mention the Aitken extrapolation, a quadratic extrapolation technique by Kamvar et al., as well as more advanced techniques such as the application of spectral methods (such as Chebyshev polynomial spectral methods). These techniques aim at obtaining a better approximation of the PageRank vector between iterations. They may be applicable in the calculation of your ranking, and it may be desirable to implement them; see the references for more details.

With regard to the software aspects of an implementation for large-scale computations, we should mention Hadoop (http://hadoop.apache.org/). Hadoop is a full-blown, top-level project of the Apache Software Foundation and it offers an open source software platform that's scalable, economical, efficient, and reliable. Hadoop implements MapReduce (see Dean and Ghemawat), by using its own distributed filesystem (HDFS). MapReduce divides applications into many small blocks of work. HDFS creates multiple copies of data blocks for reliability, and places them on computational nodes around a computational cluster (see figure 2.12). MapReduce can then process the data where it's located. Hadoop has been demonstrated on clusters with 2,000 nodes. The current design target for the Hadoop platform is 10,000 node clusters.

The ability to handle large datasets is certainly of great importance in real-world production systems. We gave you a glimpse of the issues that can arise and pointed you to some appropriate projects and the relevant literature on that subject. When you design a search engine, you need to consider not just your ability to scale and handle a larger volume of data, but the quality of your search results. At the end of the day, your users want your results to be fast and accurate. So, let's see a few quantitative ways of measuring whether what we have is what we want.

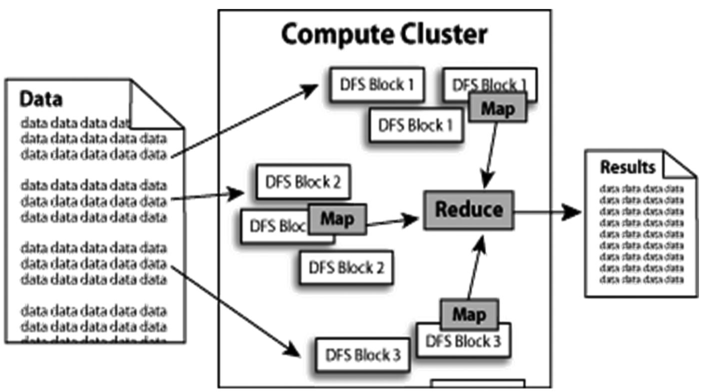

Figure 2.12 The MapReduce implementation of Hadoop using a distributed file system

2.7 Is what you got what you want? Precision and recall

Google and Yahoo! spend a considerable amount of time studying the quality of their search engines. Similar to the process of validation and verification (QA) of software systems, search quality is crucial to the success of a search engine. If you submit a query to a search engine, you may or may not find what you want. There are various metrics that quantify the degree of success for a search engine. The two most common metrics—*precision* and *recall*—are easy to implement and understand qualitatively.

Figure 2.13 shows the possibilities of results from a typical query. That is, provided a set of documents, a subset of these documents will be relevant to your query and another subset will be retrieved. Clearly the goal is to retrieve all the relevant documents, but that's rarely the case. So, our attention turns quickly to the intersection between these two sets, as indicated in figure 2.13.

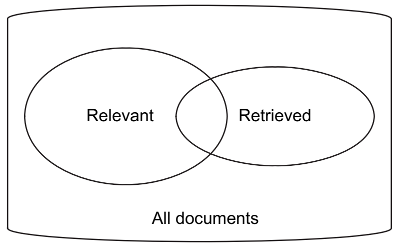

Figure 2.13 This diagram shows the set of relevant documents and the set of retrieved documents; their intersection is used to define the search metrics precision and recall.

In information retrieval, precision is the ratio of the number of relevant documents that are retrieved (RR) divided by the total number of retrieved documents (Rd)—*precision = RR/Rd*. In figure 2.13, precision would be about 1/5 or 0.2. That's measured with the "eye norm"; it's not exact, we're engineers after all! On the other hand, recall is the ratio of the number of relevant documents that are retrieved divided by the total number of relevant documents (Rt)—*recall = RR/Rt*.

Qualitatively, these two measures answer different questions. Precision answers, "To what extent do I get what I want?" Recall answers, "Does what I got include everything that I can get?" Clearly it's easier to find precision than it is to find recall, because finding recall implies that we already know the set of all relevant documents for a given query. In reality, that's hardly ever the case. We plot these two measures together so that we can assess to what extent the good results blend with bad results. If what I get is the truth, the whole truth, and nothing but the truth, then the precision and recall values for my queries will both be close to one.

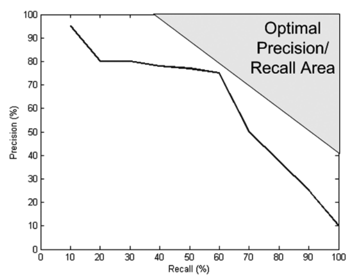

Figure 2.14 A typical precision/recall plot for a search engine

During the evaluation of the algorithms and tweaks involved in tuning a search engine, you should employ plots of these two quantities for representative queries that span the range of questions that your users are trying to answer. Figure 2.14 shows a typical plot of these quantities. For each query, we enter a point that corresponds to the precision and recall values of that query. If you execute many queries and plot these points, you'll get a line that looks like the one shown in figure 2.14. Be aware that interpolating the values, if you have a small number of queries, may not be a good idea. It would be better to leave the values as points without connecting them.

Good precision-recall points are located in the upper-right corner of the graph because we want to have high precision and high recall. These plots can help you establish, objectively, the need for a particular tweak in an algorithm or the superiority of one approach versus another. It could help you convince your ever-skeptical upper management team to use the algorithms of this book! You can practice by using the three approaches that we presented in this chapter (search with Lucene; Lucene and PageRank; Lucene, PageRank, and user clicks). You can apply them on the dataset that we provided you or another dataset that you can create yourself, and you can create a precision/recall plot that includes the results of 10–20 queries.

In section 5.5, we'll discuss many aspects of credibility that can be evaluated for a particular algorithm and how to compare two algorithms. We'll also talk about the way that the validation experiments must be carried out in order to enhance the confidence that we have in our results. Precision and recall are the tip of the iceberg when we consider the quality of our search results. We'll postpone a more detailed analysis of credibility until after we cover all the basic intelligent algorithms that we want to present. This approach will allow us to use a general framework for assessing the quality of intelligence.

2.8 Summary

Since early 2000, a lot of online news article have proclaimed: "Search is king!" This kind of statement could've been insightful, and perhaps prophetic, in the last millennium, but it's a globally accepted truth today. If you don't believe us, Google it!

This chapter has shown that intelligently answering user queries on content-rich material that's spread across the globe deserves attention and effort beyond indexing. We've demonstrated a searching strategy that starts with building on traditional information retrieval techniques provided by the Lucene library. We talked about collecting content from the Web (web crawling) and provided our own crawler implementation. We used a number of document parsers such as NekoHTML and the TextMining library (tm-extractor), and passed the content to the Lucene analyzers. The standard Lucene analyzers are powerful and flexible, and should be adequate for most purposes. If they're not suitable for you, we've discussed a number of potential extensions and modifications that are possible. We also hinted at the power of the Lucene querying framework and its own extensibility and flexibility.

More importantly, we've described in great detail the most celebrated link analysis algorithm—PageRank. We provided a full implementation that doesn't have any dependencies and adopts the formulation of the *G*(oogle) matrix that's amenable to the large-scale implementation of sparse matrices. We also provided hints that'll allow you to complete this step and feel the pride of that great accomplishment yourself! We've touched upon a number of intricacies of that algorithm and explained its key characteristics, such as the teleportation component and the power method, in detail.

We also presented user-click analysis, which introduced you to intelligent probabilistic techniques such as our `NaiveBayes` classifier implementation. We've provided wrapper classes that expose all the important steps involved, but we've also analyzed the code under the hood to a great extent. This kind of technique allows us to learn the preferences of a user toward a particular site or topic, and it can be greatly enhanced and extended to include additional features.

Since one size doesn't fit all, we've provided material that'll help you deal with documents that aren't web pages, by employing a new algorithm that we called DocRank. This algorithm has shown some promise, but more importantly it demonstrates that the underlying mathematical theory of PageRank can be readily extended and studied in other contexts by careful modifications. Lastly, we talked about some of the challenges that may arise in dealing with very large networks, and we provided a simple yet robust way of qualifying your search results and add credibility to your search engine.

The statement "search is king" might be true, but recommendation systems also have royal blood! The next chapter covers exclusively the creation of suggestions and recommendations. Adding both to your application can make a big difference in the user experience of your application. But before you move on, make sure that you read the To do items for search, if you haven't done so already. They're full of interesting and valuable information.

2.9 To do

The last section of every chapter in the rest of this book will contain a number of to-do items that will guide you in the exploration of various topics. Whenever appropriate, our code has been annotated with "TODO" tags that you should be able to view in the Eclipse IDE in the Tasks panel. By clicking on any of the tasks, the task link will show the portion of the code associated with it. If you don't use Eclipse then simply search the code for the term "TODO".

Some of these to-do items aim at providing greater depth on a topic that's been covered in the main chapter, while others present a starting point for exploration on topics that are peripheral to what we've already discussed. The completion of these tasks will provide you with greater depth and breadth on intelligent algorithms. We highly encourage you to peruse them.

With that in mind, here is our to do list for chapter 2.

1. *Build your own web search engine.* Use the crawler of your choice and crawl your favorite site, such as http://jakarta.apache.org/, then use our crawler to process the retrieved data, build an index for it, and search through its pages.

   How do the results vary if you add PageRank to them?

   How about user clicks?

   You could write your own small web search engine by applying the material of this chapter. Try it and let us know!

2. *Experiment with boosting.* Uncomment the code between lines 83 to 85 in the class `LuceneIndexBuilder` and see how the results of the Lucene ranking

change. Depending on your application, you can devise a unique strategy of boosting your documents that depends on factors that are specific to the domain of your application.

3 *Scaling the PageRank values.* In our example of a combined Lucene (index) and PageRank (ranking) search, we use a scaling factor that boosted the value of the PageRank. Our choice of function for the exponent had only one parameter—`m = (1 - 1/n)`, where *n* is the size of the *H* matrix—and its behavior was such that for large networks our scaling factor is approaching the value 1, while for small networks the value is between 0 and 1. In reality, you get zero only in the degenerate case where you have a single page, but that's not a very interesting network anyway!

Experiment with such scaling factors and observe the impact on the rankings. You may want to change that value to a higher power of *n*—another valid formula would be `m = (1 - 1 / Math.pow(n,k) )`, because as *k* takes on values greater than 1, the PageRank value approaches its calculated value faster.

4 *Altering the* G *matrix: Dangling nodes.* We've assigned a value of 1/n to all the nodes for each entry in a dangling node row. In the absence of additional information about the browsing habits of our users, or under the assumption that there's a sufficient number of users that covers all browsing habits, that's a reasonable assignment. But what if we make different kind of assumptions that are equally reasonable would the whole mechanism work?

Let's assume that a user encounters a dangling node. Upon arriving at the dangling node, it seems natural to assume that the user is more likely to select a search engine as his next destination, or a website similar to the dangling node, rather than a website that's dissimilar to the content of the dangling node. That kind of assumption would result in an adjustment of the dangling node values: higher values for search engines and similar content pages, and lower values for everybody else. How does that change affect the PageRank values? How about the results of the queries? Did your precision recall graph change in that case?

5 *Altering the* G *matrix: Teleportation.* In our original implementation, the teleportation contribution has been assigned in an egalitarian manner—all pages are assigned a contribution equal to *(1-alpha)/n*, where *n* is the number of the pages. But the potential of that component is enormous. If chosen appropriately, it can create an online bourgeois, and if it's chosen at a user level, it can target the preferences of each user much like the technique of user clicks allowed us to do. The latter reason is why the teleportation contribution is also known as the *personalization vector.*

Try to modify it so that certain pages get more weight than others. Does it work? Are your PageRank values higher for these pages? What issues do you see with such an implementation? If we assume that we assign a personalization vector to each user, what does this imply in terms of computational effort? Is it worth it? Is it feasible? The papers by Haveliwala, Jeh & Widom, and Richardson

& Domingos are related to this and can provide you with more information and insight on this important topic.

6 *Combining different scores.* In section 2.4.3, we showed one way to combine the three different scores, in order to provide the final ranking for the results of a particular query. That's not the only way. This is a case where you can devise a balancing of these three terms in a way that best fits your needs. Here's an idea: introduce weighing terms for each of the three scores and experiment with different allocations of weight to each one of them.

Provided that you consider a fixed network of pages or documents, how do the results change based on different values of these weight coefficients? Plot 20 precision/recall values that correspond to 20 different queries, and do that for three different weight combinations, for example (0.6, 0.2, 0.2), (0.2, 0.6, 0.2), (0.2, 0.2, 0.6). What do you see? How do these points compare to the equal weight distribution (1,1,1)? Can you come up with different formulas for balancing the various contributions?

2.10 References

Adamic, L.A., R.M. Lukose, A.R. Puniyani, and B.A. Huberman. "Search in power-law networks." *Physical Review E*, vol. 64, 046135. 2001.

Boldi, P., and S. Vigna. "The WebGraph Framework I: Compression Techniques." *WWW 2004*, New York.

Dean, J. and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004. http://labs.google.com/papers/mapreduce-osdi04.pdf.

Haveliwala, T.H. "Topic-sensitive PageRank: A context-sensitive ranking algorithm for web search." *IEEE transactions on Knowledge and Data Engineering*, 15 (4): 784. 2004. http://www-cs-students.stanford.edu/~taherh/papers/topic-sensitive-pagerank-tkde.pdf.

Jeh, G. and J. Widom. "Scaling personalized web search." Technical report, Stanford University, 2002. http://infolab.stanford.edu/~glenj/spws.pdf.

Kamvar, S.D., T.H. Haveliwala, Christopher D. Manning, and Gene H. Golub. Extrapolation Methods for Accelerating PageRank Computations. WWW 2003. http://www.kamvar.org/code/paper-server.php?filename=extrapolation.pdf.

Langville, A.N. and C.D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, 2006.

Richardson, M. and P. Domingos. The intelligent surfer: Probabilistic combination of link and content information in PageRank. *Advances in Neural Information Processing Systems*, 14:1441, 2002. http://research.microsoft.com/users/mattri/papers/nips2002/qd-pagerank.pdf.

Rish, I. An empirical study of the naïve Bayes classifier." *IBM Research Report*, RC22230 (W0111-014), 2001. http://www.cc.gatech.edu/~isbell/classes/reading/papers/Rish.pdf.

JAVA/WEB

Algorithms *of the* Intelligent Web

Haralambos Marmanis • Dmitry Babenko

Free ebook
SEE INSERT

An algorithm is a sequence of steps that solves a problem. *Algorithms of the Intelligent Web* provides exactly that—explicit, clearly organized patterns to implement valuable web application features like recommendation engines, smart searching, content organizers, and much more. With these techniques you'll capture vital raw information about your users and profitably transform it into action.

Algorithms of the Intelligent Web is a handbook for web developers who want to exploit relationships in user data that can't be discovered manually. The book presents crystal-clear explanations of techniques you can apply immediately. It is based on the authors' practical experience as web developers and their deep expertise in the science of machine learning. With a wealth of detailed, Java-based examples this book shows you how to build applications that behave intelligently and learn from your users' actions.

What's Inside

- Create recommendations like Netflix or Amazon
- Implement Google's PageRank algorithm
- Discover matches on social-networking sites
- Organize your news group discussions
- Select topics of interest from shared bookmarks
- Filter spam and categorize emails based on content

Dr. Haralambos (Babis) Marmanis is a pioneer in the adoption of machine learning techniques for industrial solutions, and also a world expert in supply management. **Dmitry Babenko** has designed applications and infrastructure for banking, insurance, supply-chain management, and business intelligence companies.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/AlgorithmsoftheIntelligentWeb

"Unequivocally outstanding—this is the best technical book I have read all year."
—Robert Hanson
Quality Technology Services

"You don't need a PhD to build an intelligent website—pick up this book instead."
—Ajay Bhandari, FoodieBytes.com

"Very useful ... will bring you up to speed quickly."
—Sumit Pal, LeapFrogrx

"Excellent ... a perfect blend of theory and practice."
—Carlton Gibson
Noumenal Software

"Unlock the future of the web by analyzing what we know today!"
—Eric Swanson, AAA

MANNING $44.99 / Can $56.99 [INCLUDING eBOOK]

ISBN 13: 978-1-933988-66-5
ISBN 10: 1-933988-66-5
54499
9781933988665