Chapter 13: Bot Programming and Neural Networks

- Creating a Simple Bot
- Analyzing Text
- Training a Neural Bot
- Using a Neural Bot

Bots are computer programs that are designed to use the Internet in much the same way as humans use it. Neural networks can be useful in developing bots. In this chapter you will see how a neural network can be used to assist a bot in finding desired information on the Internet.

A discussion exploring the creation of bots could easily fill a book. If you would like to learn more about bot programming, you may find the book HTTP Programming Recipes for Java Bots (ISBN: 0977320669) helpful. This book presents many algorithms commonly used to create bots. The bots created in this chapter will use some of the code from HTTP Programming Recipes for Java Bots.

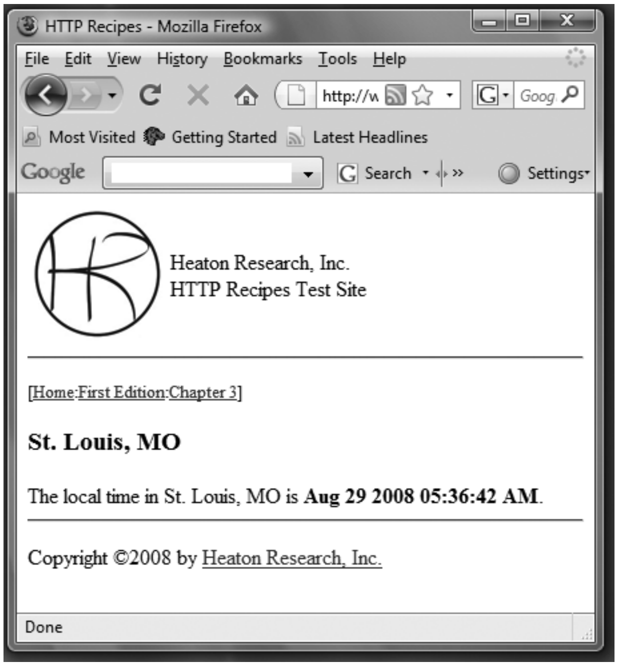
A Simple Bot

A bot is generally used to retrieve information from a website. Before we create a complex bot that makes use of neural networks, it will be useful to see how a simple bot is created. The simple bot will get the current time for the city of St. Louis, Missouri from a website. The site that has the data we are seeking is located at the following URL:

http://www.httprecipes.com/1/3/time.php

If you point a browser to the above URL, the page shown in Figure 13.1 will be displayed.

Figure 13.1: Local time in St. Louis, MO.


HTTP Recipes - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://w
Most Visited Getting Started Latest Headlines
Google Search Settings
Done

Heaton Research, Inc.
HTTP Recipes Test Site

[Home:First Edition:Chapter 3]

St. Louis, MO

The local time in St. Louis, MO is **Aug 29 2008 05:36:42 AM**.

Copyright ©2008 by Heaton Research, Inc.

The source code for the simple bot is shown in Listing 13.1.

Listing 13.1: A Simple Bot (SimpleBot.java)

```
package com.heatonresearch.book.introneuralnet.ch13;

import java.io.*;
import java.net.*;

public class SimpleBot
{
  /**
   * The size of the download buffer.
   */
  public static int BUFFER_SIZE = 8192;

  /**
   * This method downloads the specified URL into a Java
   * String. This is a very simple method, that you can
   * reused anytime you need to quickly grab all data from
   * a specific URL.
   *
   * @param url The URL to download.
```

```
 * @return The contents of the URL that was downloaded.
 * @throws IOException Thrown if any sort of error occurs.
 */
public String downloadPage(URL url) throws IOException
{
  StringBuilder result = new StringBuilder();
  byte buffer[] = new byte[BUFFER_SIZE];

  InputStream s = url.openStream();
  int size = 0;

  do
  {
    size = s.read(buffer);
    if (size != -1)
      result.append(new String(buffer, 0, size));
  } while (size != -1);

  return result.toString();
}

/**
 * Run the example.
 *
 * @param page The page to download.
 */
public void go(String page)
{
  try
  {
    URL u = new URL(page);
    String str = downloadPage(u);
    System.out.println(str);

  } catch (MalformedURLException e)
  {
    e.printStackTrace();
  } catch (IOException e)
  {
    e.printStackTrace();
  }
}

/**
 * Typical Java main method, create an object, and then
 * call that object's go method.
```

```
   *
   * @param args Website to access.
   */
  public static void main(String args[])
  {
    SimpleBot module = new SimpleBot();
    String page;
    if (args.length == 0)
      page = "http://www.httprecipes.com/1/3/time.php";
    else
      page = args[0];
    module.go(page);
  }
}
```

The bot begins by creating a new **URL** object for the desired web page.

```
URL u = new URL("http://www.httprecipes.com/1/3/time.php");
```

The contents of the page are then downloaded into a **String** object. The **downloadPage** method performs this operation.

```
String str = downloadPage(u);
```

Finally, the time is extracted from the HTML. This is done using the **extract** method. The beginning and ending tags that enclose the desired data are provided to the **extract** method. The third parameter specifies the instance of the **<b>** tag to be found. The number one means that we are seeking the first instance of the **<b>** tag.

```
System.out.println(extract(str, "<b>", "</b>", 1));
```

The HTML data from the target web page must be examined to see which HTML tags enclose the desired data. The HTML located at the website shown in Figure 13.1 is shown in Listing 13.2.

Listing 13.2: HTML Data Encountered by the Bot

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">

<HTML>
<HEAD>
      <TITLE>HTTP Recipes</TITLE>
      <meta http-equiv="Content-Type" content="text/html; 
charset=UTF-8">
      <meta http-equiv="Cache-Control" content="no-cache">
      <meta http-equiv="Content-Style-Type" content="text/css">
<link rel="alternate" type="application/rss+xml" title="RSS" 
href="http://www.httprecipes.com/1/12/rss2.xml">
</HEAD>
```

```
<BODY>

<table border="0"><tr><td>
<a href="http://www.httprecipes.com/"><img src="/images/logo.gif"
alt="Heaton Research Logo" border="0" width="100" height="100"></
a>
</td><td valign="middle">Heaton Research, Inc.<br>
HTTP Recipes Test Site
</td></tr>
</table>
<hr><p><small>[<a href="/">Home</a>:<a href="/1/">First Edition</
a>:
<a href="/1/3/">Chapter 3</a>]</small></p>

<h3>St. Louis, MO</h3>
The local time in St. Louis, MO is <b>Jun 09 2008 10:27:46 PM</b>.

<hr>
<p>Copyright &copy;2008 by <a href="http://www.heatonresearch.
com/">
Heaton Research, Inc.</a></p>
</BODY>
</HTML>
```

As you can see from the above listing, the data the bot seeks is located between a beginning **<b>** tag and an ending **</b>** tag. The bot made use of two methods to download and then extract the data. These two methods will be covered in the following two sections.

Downloading the Page

The contents of a web page are downloaded to a **String** object using the **downloadPage** method. The signature for the **downloadPage** method is shown here:

```
public String downloadPage(URL url) throws IOException
```

First, a new **StringBuilder** object is created to hold the contents of the web page after it has been downloaded. Additionally, a **buffer** is created to hold the data as it is read.

```
StringBuilder result = new StringBuilder();
byte buffer[] = new byte[BUFFER_SIZE];
```

Next, an **InputStream** object is created to read data from the **URL** object.

```
InputStream s = url.openStream();
int size = 0;
```

The data is then read until the end of the stream is reached.

```
do {
  size = s.read(buffer);
```

When the end of the stream is reached, the **read** method will return -1. If we have not reached the end of the stream, then the **buffer** is appended to the **result**.

```
  if (size != -1)
    result.append(new String(buffer, 0, size));
} while (size != -1);
```

Finally, the **result** is returned as a **String** object.

```
return result.toString();
```

The **downloadPage** method is very useful for obtaining a web page as a single **String** object. Once the page has been downloaded, the **extract** method is used to extract data from the page. The **extract** method is covered in the next section.

Extracting the Data

Web pages can be thought of as one large **String** object. The process of breaking the string up into useable data is called string parsing. The signature for the **extract** method is shown below. The first parameter contains the string to be parsed; the second and third parameters specify the beginning and end tokens being sought. Finally, the fourth parameter specifies for which instance of the first token you are looking.

```
public String extract(String str, String token1, String token2,
int count)
```

Two variables are created to hold the locations of the two tokens. They are both initialized to zero.

```
int location1, location2;
location1 = location2 = 0;
```

The location of the first token is then found.

```
do
{
  location1 = str.indexOf(token1, location1);
```

If the first token cannot be found, then **null** is returned to indicate that we failed to find anything.

```
  if (location1 == -1)
    return null;
```

If the first token is found, then the **count** variable is decreased and we continue to search as long as there are more instances to find.

```
  count--;
```

```
} while (count > 0);
```

We then attempt to locate the second token. If the second token is located, then its index is stored in the **`location2`** variable. If the **`extract`** method fails to find the second token, then null is returned to indicate failure.

```
location2 = str.indexOf(token2, location1 + 1);
if (location2 == -1)
  return null;
```

Finally, the characters located between the two tokens are returned.

```
return str.substring(location1 + token1.length(), location2);
```

While this bot is very simple, it demonstrates the principles of bot programming. The remaining sections of this chapter will show how to create a much more complex, neural network-based bot.

Introducing the Neural Bot

The neural network-based bot detailed below is provided with the name of a famous person. It uses this information to perform a Yahoo search and obtain information on the person. The bot "reads" all of the information found on the person and attempts to determine the individual's correct birth year.

There are three distinct modes in which this program runs. These modes are summarized in Table 13.1.

Table 13.1: When Born Neural Bot Modes

Mode	Purpose
Gather	In this mode, the bot gathers articles on famous people. It performs a Yahoo search to obtain a number of articles on each. Depending on your Internet speed, this process can take from fifteen minutes to several hours.
Train	Using the data gathered in the first mode, a neural network is built and trained. Once the acceptable error level is reached, this neural network is saved.
Born	After the neural network has been trained, new famous people can be entered. The bot will then attempt to discover their birth year.

The program must run in the **`Gather`** mode before it can be trained. Likewise, the program must be trained before it can perform the **`Born`** operation.

There are several configuration constants that can be modified for this bot. These items are shown in Listing 13.3.

Listing 13.3: Configuring the Neural Bot (Config.java)

```
package com.heatonresearch.book.introneuralnet.ch13;
/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * Config: Holds configuration data for the bot.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class Config {

      public final static int INPUT_SIZE = 10;
      public final static int OUTPUT_SIZE = 1;
      public final static int NEURONS_HIDDEN_1 = 20;
      public final static int NEURONS_HIDDEN_2 = 0;
      public final static double ACCEPTABLE_ERROR = 0.01;
      public final static int MINIMUM_WORDS_PRESENT = 3;

      public static final String FILENAME_GOOD_TRAINING_TEXT =
"bornTrainingGood.txt";
      public static final String FILENAME_BAD_TRAINING_TEXT =
"bornTrainingBad.txt";
      public static final String FILENAME_COMMON_WORDS = "common.
csv";
      public static final String FILENAME_WHENBORN_NET = "whenborn.
net";
      public static final String FILENAME_HISTOGRAM = "whenborn.
hst";
}
```

The configuration constants are defined in Table 13.2.

Table 13.2: Configuring the Neural Bot

Constant	Default	Purpose
INPUT_SIZE	10	Number of input neurons.
OUTPUT_SIZE	1	Number of output neurons.
NEURONS_HIDDEN_1	20	Number of neurons in the first hidden layer.
NEURONS_HIDDEN_2	0	Number of neurons in the second hidden layer.
ACCEPTABLE_ERROR	0.01	The maximum acceptable error level.
MINIMUM_WORDS_PRESENT	3	Minimum number of words for a sentence.
FILENAME_GOOD_TRAINING_TEXT	bornTrainingGood.txt	Good sentences gathered.
FILENAME_BAD_TRAINING_TEXT	bornTrainingBad.txt	Bad sentences gathered.
FILENAME_COMMON_WORDS	common.csv	Common English words.
FILENAME_WHENBORN_NET	whenborn.net	The saved trained neural network.
FILENAME_HISTOGRAM	whenborn.hst	The saved histogram of common words.

We will now examine each mode of operation for the neural bot.

Gathering Training Data for the Neural Bot

The neural-based bot works by performing a Yahoo search on a person of interest. All articles returned from Yahoo are decomposed into sentences and scanned for the correct birth year. A list containing famous people and their birth years is used as training data. This list is stored in a file named “famous.csv”. A sampling of the data is shown in Listing 13.4.

Listing 13.4: Famous People

```
Person,Year
Abdullah bin Abdul Aziz Al Saud,1924
Al Gore,1948
Alber Elbaz,1961
America Ferrera,1984
Amr Khaled,1967
Angela Merkel,1954
Anna Netrebko,1971
Arnold Schwarzenegger,1947
Ayatullah Ali Khamenei,1939
Barack Obama,1961
Bernard Arnault,1949
Bill Gates,1955
Brad Pitt,1963
...
Warren Buffett,1930
Wesley Autrey,1956
William Jefferson Clinton,1946
Youssou N'Dour,1959
Zeng Jinyan,1983
```

Using this list, a Yahoo search will be performed to gather information for each of these people.

Gathering the Data

To gather data for the training, the **GatherForTrain** class is executed. The **GatherForTrain** class begins performing Yahoo searches and gathering data for all of the famous people. The **GatherForTrain** class is built to be multithreaded. This speeds the processing, even on a single-processor computer. Because the program must often wait for the many websites it visits to respond, it improves efficiency to access several websites at once. The source code for the **GatherForTrain** class is shown in Listing 13.5.

Listing 13.5: Gathering Training Data (GatherForTrain.java)

```java
package com.heatonresearch.book.introneuralnet.ch13.gather;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Date;
import java.util.Vector;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import com.heatonresearch.book.introneuralnet.ch13.Config;
import com.heatonresearch.book.introneuralnet.ch13.ScanReportable;
import com.heatonresearch.book.introneuralnet.common.ReadCSV;

/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * GatherForTrain: Gather training data for the neural network.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class GatherForTrain implements ScanReportable {
      public static final boolean LOG = true;
      public static final int THREAD_POOL_SIZE = 20;

      /**
       * The main method processes the command line arguments
       * and then calls
       * process method to determine the birth year.
       *
       * @param args
       *           Holds the Yahoo key and the site to search.
       */
      public static void main(final String args[]) {
            try {
                  final GatherForTrain when = new GatherForTrain();
                  when.process();
            } catch (final Exception e) {
                  e.printStackTrace();
            }
      }

      private ExecutorService pool;
      private final Collection<String> trainingDataGood = new 
Vector<String>();
      private final Collection<String> trainingDataBad = new 
Vector<String>();
      private int currentTask;

      private int totalTasks;
	/**
	 * This method is called to determine the birth year for a
	 * person. It
	 * obtains 100 web pages that Yahoo returns for that person.
	 * Each of these
	 * pages is then searched for the birth year of that person.
	 * Which ever year
	 * is selected the largest number of times is selected as
	 * the birth year.
	 *
	 * @param name
	 *           The name of the person you are seeing the
	 * birth year for.
	 * @throws IOException
	 *            Thrown if a communication error occurs.
	 */
	public void process() throws IOException {
		this.pool = Executors
				.newFixedThreadPool(
			GatherForTrain.THREAD_POOL_SIZE);
		final Collection<Callable<Integer>> tasks =
			new ArrayList<Callable<Integer>>();
		final ReadCSV famous = new ReadCSV("famous.csv");
		report(
"Building training data from list of famous people.");
		final Date started = new Date();

		while (famous.next()) {
			final String name = famous.get("Person");
			final int year = famous.getInt("Year");

			final CollectionWorker worker =
				new CollectionWorker(this, name,
					year);
			tasks.add(worker);
		}

		try {
			this.totalTasks = tasks.size();
			this.currentTask = 1;
			this.pool.invokeAll(tasks);
			this.pool.shutdownNow();
			report("Done collecting Internet data.");
		} catch (final InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
```

```
            }
            long length = (new Date()).getTime() -
                  started.getTime();
            length /= 1000L;
            length /= 60;
            System.out.println("Took " + length
+ " minutes to collect training data from the Internet.");
            System.out.println("Writing training file");
            writeTrainingFile();

      }

      public void receiveBadSentence(final String sentence) {
            this.trainingDataBad.add(sentence);

      }

      public void receiveGoodSentence(final String sentence) {
            this.trainingDataGood.add(sentence);

      }

      public void report(final String str) {
            synchronized (this) {
                  System.out.println(str);
            }
      }

      public void reportDone(final String string) {
            report(this.currentTask + "/" + this.totalTasks
                  + ":" + string);
            this.currentTask++;
      }

      private void writeTrainingFile() throws IOException {
            OutputStream fs = new FileOutputStream(
                        Config.FILENAME_GOOD_TRAINING_TEXT);
            PrintStream ps = new PrintStream(fs);

            for (final String str : this.trainingDataGood) {
                  ps.println(str.trim());
            }

            ps.close();
            fs.close();
            fs = new FileOutputStream(
                  Config.FILENAME_BAD_TRAINING_TEXT);
            ps = new PrintStream(fs);

            for (final String str : this.trainingDataBad) {
                  ps.println(str.trim());
            }

            ps.close();
            fs.close();
      }

}
```

The **process** method performs almost all of the work in the **GatherForTrain** class. The signature for the **process** method is shown here:

```
public void process() throws IOException
```

First, a new thread pool is created.

```
this.pool = Executors
      .newFixedThreadPool(GatherForTrain.THREAD_POOL_SIZE);
final Collection<Callable<Integer>> tasks = new
ArrayList<Callable<Integer>>();
```

Next, the **ReadCSV** class is used to read the comma separated value list of famous people. Comma separated files were introduced in chapter 10.

```
final ReadCSV famous = new ReadCSV("famous.csv");
report("Building training data from list of famous people.");
final Date started = new Date();
```

While there is still data to be read, we continue to loop through the list of famous people.

```
while (famous.next()) {
```

The famous person and their birth year are obtained from the CSV file.

```
  final String name = famous.get("Person");
  final int year = famous.getInt("Year");
```

A **CollectionWorker** object is created to actually process this person, and it is added to the thread pool.

```
  final CollectionWorker worker = new CollectionWorker(this, name,
        year);
  tasks.add(worker);
}
```

The thread pool is then executed.

```
try {
  this.totalTasks = tasks.size();
  this.currentTask = 1;
  this.pool.invokeAll(tasks);
  this.pool.shutdownNow();
  report("Done collecting Internet data.");
} catch (final InterruptedException e) {
  e.printStackTrace();
}
```

Once the thread pool terminates, the processing is complete and the amount of time the processing took is displayed.

```
long length = (new Date()).getTime() - started.getTime();
length /= 1000L;
length /= 60;
System.out.println("Took " + length
      + " minutes to collect training data from the Internet.");
System.out.println("Writing training file");
writeTrainingFile();
```

Finally, the data is written to the training file.

Support for Multithreading

Multithreading for the gathering mode is provided by the **CollectionWorker** class. The **CollectionWorker** class is shown in Listing 13.6.

Listing 13.6: Collection Worker (CollectionWorker.java)

```
package com.heatonresearch.book.introneuralnet.ch13.gather;

import java.io.IOException;
import java.net.URL;
import java.util.Collection;
import java.util.concurrent.Callable;

import com.heatonresearch.book.introneuralnet.ch13.Text;
import com.heatonresearch.book.introneuralnet.common.YahooSearch;

/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * CollectionWorker: Worker class in a thread pool used to
 * gather sentences from the web.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class CollectionWorker implements Callable<Integer> {

      private final GatherForTrain bot;

      /*
       * The search object to use.
       */
      private final YahooSearch search;

      private final String name;
      private final int year;

      public CollectionWorker(final GatherForTrain bot,
            final String name,
                  final int year) {
            this.bot = bot;
            this.name = name;
            this.year = year;
            this.search = new YahooSearch();
      }

      public Integer call() throws Exception {
            try {
                  scanPerson(this.name, this.year);
                  this.bot.reportDone(this.name +
                  ", done scanning.");
            } catch (final Exception e) {
                  this.bot.reportDone(this.name +
                  ", error encountered.");
                  e.printStackTrace();
                  throw e;
            }
            return null;
      }

      private void scanPerson(final String name, final int year)
                  throws IOException {
            final Collection<URL> c = this.search.search(name);
            int i = 0;

            for (final URL u : c) {
                  try {
                        i++;
                        Text.checkURL(this.bot, u, year);
                  } catch (final IOException e) {
                    }
                }
            }

    }
```

The thread pool initiates the worker by calling the **call** method. As you can see from the listing of the **call** method above, it does little more than call the **scanPerson** method. The signature for the **scanPerson** method is shown here:

```
private void scanPerson(final String name, final int year)
  throws IOException
```

First, a collection of URLs is obtained from Yahoo for the specified famous person.

```
final Collection<URL> c = this.search.search(name);
int i = 0;
```

For each **URL** that is obtained, the **checkURL** method of the **Text** class is called. The **Text** class will be covered in the next section.

```
for (final URL u : c) {
  try {
      i++;
      Text.checkURL(this.bot, u, year);
  } catch (final IOException e) {

  }
}
```

If an error is encountered, it is likely an invalid URL was obtained from Yahoo. This is not an unusual occurrence; such URLs are simply ignored.

Parsing Websites

As you saw in the last section, websites are parsed using the class named **Text**. The **Text** class is shown in Listing 13.7.

Listing 13.7: Parsing Websites (Text.java)

```
package com.heatonresearch.book.introneuralnet.ch13;

import java.io.IOException;
import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;
import java.util.StringTokenizer;
import com.heatonresearch.httprecipes.html.ParseHTML;

/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * Text: Text parsing utilities.
 *
 * @version 2.1
 */
public class Text {

      /**
       * Check the specified URL for a birth year. This will
       * occur if one sentence is found that has the word
       * born, and a numeric value less than 3000.
       *
       * @param url
       *            The URL to check.
       * @throws IOException
       *             Thrown if a communication error occurs.
       */
      public static void checkURL(final ScanReportable report,
                  final URL url,
                  final Integer desiredYear) throws IOException {
            int ch;
            final StringBuilder sentence = new StringBuilder();
            String ignoreUntil = null;

            final URLConnection http = url.openConnection();
            http.setConnectTimeout(1000);
            http.setReadTimeout(1000);
            final InputStream is = http.getInputStream();
            final ParseHTML html = new ParseHTML(is);
            do {
                  ch = html.read();
                  if ((ch != -1) && (ch != 0)
                        && (ignoreUntil == null)) {
                        if (".?!".indexOf(ch) != -1) {
                              final String str =
                                    sentence.toString();
                              final int year =
                                    Text.extractYear(str);

                              if (desiredYear == null) {
                                    // looking for any year
                                    if (year != -1) {
                                          report.
                              receiveGoodSentence(str);
                                    }
                              } else {
                                    // looking for a specific year
                                    if (year == desiredYear) {
                                          report.
                              receiveGoodSentence(str);
                                    } else if (year != -1) {
                                          report.
                              receiveBadSentence(str);
                                    }
                              }
                              sentence.setLength(0);
                        } else if (ch == ' ') {
                              if ((sentence.length() > 0)
                                          && (sentence.
                        charAt(sentence.length() - 1) != ' ')) {
                                    sentence.append(' ');
                              }
                        } else if ((ch != '\n') && (ch != '\t')
                              && (ch != '\r')) {
                              if ((ch) < 128) {
                                    sentence.append((char) ch);
                              }
                        }
                  } else if (ch == 0) {
                        // clear anything before a body tag
                        if (html.getTag().getName().
equalsIgnoreCase("body")
      || html.getTag().getName().equalsIgnoreCase("br")
      || html.getTag().getName().equalsIgnoreCase("li")
      || html.getTag().getName().equalsIgnoreCase("p")
      || html.getTag().getName().equalsIgnoreCase("h1")
      || html.getTag().getName().equalsIgnoreCase("h2")
      || html.getTag().getName().equalsIgnoreCase("h3")
      || html.getTag().getName().equalsIgnoreCase("td")
      || html.getTag().getName().equalsIgnoreCase("th")) {
                              sentence.setLength(0);
                        }
                        // ignore everything between script and
style tags
                        if (ignoreUntil == null) {
                              if (html.getTag().getName().
                  equalsIgnoreCase("script")) {
                                    ignoreUntil = "/script";
                              } else if (html.getTag().getName()

                  .equalsIgnoreCase("style")) {
                                    ignoreUntil = "/style";
                              }
                        } else {
                              if (html.getTag().getName()
                  .equalsIgnoreCase(ignoreUntil)) {
                                    ignoreUntil = null;
                              }
                        }

                        // add a space after the tag
                        if (sentence.length() > 0) {
                              if (sentence.charAt(
                  sentence.length() - 1) != ' ') {
                                    sentence.append(' ');
                              }
                        }
                  }
            } while (ch != -1);

      }

      /**
       * Examine a sentence and see if it contains the word
       * born and a number.
       *
       * @param sentence
       *            The sentence to search.
       * @return The number that was found.
       */
      public static int extractYear(final String sentence) {
            int result = -1;

            final StringTokenizer tok =
                  new StringTokenizer(sentence);
            while (tok.hasMoreTokens()) {
                  final String word = tok.nextToken();

                  try {
                        result = Integer.parseInt(word);

                        if ((result < 1600) || (result > 2100)) {
                              result = -1;
                        }
```

```
                } catch (final NumberFormatException e) {
                }
            }

            return result;
        }

}
```

The **checkURL** method of the **Text** class is called to scan a web page and look for sentences. These sentences are then divided into two categories: good and bad. If a sentence contains a year, and it is the year of the famous person's birth, then this sentence is a good sentence; otherwise, it is a bad sentence. The two lists created as a result of this sorting are then used to train the neural network. The signature for the **checkURL** method is shown here:

```
public static void checkURL(
final ScanReportable report,
final URL url,
final Integer desiredYear) throws IOException
```

As you can see, three parameters are passed to this method. The first, named **report**, specifies the object that will receive the good and bad sentence lists that this method will produce. The second, named **url**, specifies the **URL** of the website to be scanned. The third, named **desiredYear**, specifies the year that the person was actually born.

First, variable **ch** is declared to hold the current character. Variable **sentence** is declared to hold the current sentence, and **ignoreUntil** is declared to hold an HTML tag that we will rely on to indicate when we should begin processing data again. For example, if we encounter the HTML tag **<script>**, we will set **ignoreUntil** to **</script>** and will not process any of the code between **<script>** and the **</script>** tag. The data between these tags is Javascript and will not be useful in determining a birth year.

```
int ch;
final StringBuilder sentence = new StringBuilder();
String ignoreUntil = null;
```

An HTTP connection is opened to the specified URL. Timeouts of 1,000 milliseconds, or one second, are specified. We will be processing a large number of pages and we do not want to wait too long for one to return an error. If data is not ready in a second, we move on to the next page.

```
final URLConnection http = url.openConnection();
http.setConnectTimeout(1000);
http.setReadTimeout(1000);
final InputStream is = http.getInputStream();
final ParseHTML html = new ParseHTML(is);
```

With the connection established, we begin reading from the HTML page. This method uses the HTML parser provided in the book HTTP Programming Recipes for Java Bots (ISBN: 0977320669).

```
do {
  ch = html.read();
```

If the value of **ch** is -1, then an HTML tag has been encountered. Otherwise, **ch** will be the next character of text. If the character is a period, question mark, or exclamation point, then we know we have reached the end of a sentence, unless we are ignoring data until we reach a tag, in which case **ignoreUntil** will not be **null**.

```
  if ((ch != -1) && (ch != 0) && (ignoreUntil == null)) {
      if (".?!".indexOf(ch) != -1) {
```

When we have accumulated a complete sentence, we convert the **sentence StringBuffer** into a **String** and determine whether or not the sentence contains a year. The **Text** class is used to accomplish this. The **Text** class does nothing more than use a **StringTokenizer** to split the sentence at each blank space and search for a year.

```
        final String str = sentence.toString();
        final int year = Text.extractYear(str);
```

We then determine if a particular year has been specified.

```
        if (desiredYear == null) {
```

If not, then the year found is reported and the user is told this a “good sentence.” If no year was found, then the **year** variable is -1.

```
            if (year != -1) {
              report.receiveGoodSentence(str);
            }
        } else {
```

If we are looking for a specific year, then we determine if the year found matches the **desiredYear** variable and report it as either a “good” or “bad” sentence. If no year was found, then we do not report anything.

```
            if (year == desiredYear) {
              report.receiveGoodSentence(str);
            } else if (year != -1) {
              report.receiveBadSentence(str);
            }
        }
```

The **sentence StringBuffer** is then cleared to make way for the next sentence.

```
        sentence.setLength(0);
```

If multiple spaces occur together, they are trimmed to a single space.

```
      } else if (ch == ' ') {
        if ((sentence.length() > 0)
              && (sentence.charAt(sentence.length() - 1) != ' '))
{
            sentence.append(' ');
        }
```

The following formatting characters and any characters above 128 are not processed.

```
      } else if ((ch != '\n') && (ch != '\t') && (ch != '\r')) {
        if ((ch) < 128) {
            sentence.append((char) ch);
        }
      }
```

The following tags begin new sections. The **`sentence`** is cleared if one is encountered and we start over.

```
  } else if (ch == 0) {
      // clear anything before a body tag
      if (html.getTag().getName().equalsIgnoreCase("body")
            || html.getTag().getName().equalsIgnoreCase("br")
            || html.getTag().getName().equalsIgnoreCase("li")
            || html.getTag().getName().equalsIgnoreCase("p")
            || html.getTag().getName().equalsIgnoreCase("h1")
            || html.getTag().getName().equalsIgnoreCase("h2")
            || html.getTag().getName().equalsIgnoreCase("h3")
            || html.getTag().getName().equalsIgnoreCase("td")
            || html.getTag().getName().equalsIgnoreCase("th")) {
        sentence.setLength(0);
      }
```

If we are not already in the "ignore until" mode, then we determine if we should now switch to this mode.

```
      // ignore everything between script and style tags
      if (ignoreUntil == null) {
```

If the tag encountered is a **`<script>`** tag, then we ignore all text until an ending **`</script>`** tag is found.

```
        if (html.getTag().getName().equalsIgnoreCase("script")) {
            ignoreUntil = "/script";
        } else if (html.getTag().getName()
              .equalsIgnoreCase("style")) {
            ignoreUntil = "/style";
        }
```

```
      } else {
```

If we encounter the tag that we have been looking for, then we clear the **ignoreUntil** variable and begin processing characters again.

```
        if (html.getTag().getName().equalsIgnoreCase(ignoreUntil))
{
            ignoreUntil = null;
        }
      }
```

A space is added after any tag is encountered.

```
      // add a space after the tag
      if (sentence.length() > 0) {
        if (sentence.charAt(sentence.length() - 1) != ' ') {
            sentence.append(' ');
        }
      }
  }
} while (ch != -1);
```

Processing continues until the end is reached. At the end of this process, the sentences are sorted into good and bad. Only those sentences that contained a valid year are considered.

Training the Neural Bot

The **TrainBot** class is executed to train the neural bot. This class makes use of several other classes in the **train** package. The **TrainBot** class is shown in Listing 13.8.

Listing 13.8: Training the Bot (TrainBot.java)

```
package com.heatonresearch.book.introneuralnet.ch13.train;

import java.io.IOException;

import com.heatonresearch.book.introneuralnet.ch13.CommonWords;
import com.heatonresearch.book.introneuralnet.ch13.Config;
import com.heatonresearch.book.introneuralnet.ch13.NetworkUtil;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardNetwork;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.Train;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
train.backpropagation.Backpropagation;
import com.heatonresearch.book.introneuralnet.neural.util.
SerializeObject;
```

```
/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * TrainBot:Train the bot with the data gathered.
 */
public class TrainBot {

      public static void main(final String args[]) {
            try {
                  final TrainBot trainBot = new TrainBot();
                  trainBot.process();
            } catch (final Exception e) {
                  e.printStackTrace();
            }
      }

      private int sampleCount;
      private CommonWords common;
      private double input[][];
      private double ideal[][];
      private FeedforwardNetwork network;
      private AnalyzeSentences goodAnalysis;
      private AnalyzeSentences badAnalysis;
      WordHistogram histogramGood;
      WordHistogram histogramBad;

      private final TrainingSet trainingSet;

      public TrainBot() throws IOException {
            this.common = new CommonWords(
                  Config.FILENAME_COMMON_WORDS);
            this.trainingSet = new TrainingSet();
      }

      private void allocateTrainingSets() {
            this.input = new double[this.sampleCount]
                  [Config.INPUT_SIZE];
            this.ideal = new double[this.sampleCount]
                  [Config.OUTPUT_SIZE];
      }

      private void copyTrainingSets() {
            int index = 0;
            // first the input
```

```
		for (final double[] array : this.trainingSet.
			getInput()) {
			System.arraycopy(array, 0, this.input[index],
				0, array.length);
			index++;
		}
		index = 0;
		// second the ideal
		for (final double[] array : this.trainingSet.
			getIdeal()) {
			System.arraycopy(array, 0, this.ideal[index],
			0, array.length);
			index++;
		}

	}

	public void process() throws IOException {
		this.network = NetworkUtil.createNetwork();
		System.out.println("Preparing training sets...");
		this.common = new CommonWords(
			Config.FILENAME_COMMON_WORDS);
		this.histogramGood = new WordHistogram(this.common);
		this.histogramBad = new WordHistogram(this.common);

		// load the good words
		this.histogramGood.buildFromFile(
			Config.FILENAME_GOOD_TRAINING_TEXT);
		this.histogramGood.buildComplete();

		// load the bad words
		this.histogramBad.buildFromFile(
			Config.FILENAME_BAD_TRAINING_TEXT);
		this.histogramBad.buildComplete();

		// remove low scoring words
		this.histogramGood
				.removeBelow((int)
			this.histogramGood.calculateMean());
		this.histogramBad.removePercent(0.99);

		// remove common words
		this.histogramGood.removeCommon(this.histogramBad);

		this.histogramGood.trim(Config.INPUT_SIZE);
		this.goodAnalysis = new AnalyzeSentences(
			this.histogramGood,
				Config.INPUT_SIZE);
		this.badAnalysis = new AnalyzeSentences(
				this.histogramGood,
				Config.INPUT_SIZE);

		this.goodAnalysis.process(this.trainingSet, 0.9,
				Config.FILENAME_GOOD_TRAINING_TEXT);
		this.badAnalysis.process(this.trainingSet, 0.1,
				Config.FILENAME_BAD_TRAINING_TEXT);

		this.sampleCount = this.trainingSet.getIdeal().size();
		System.out
				.println("Processing "
			+ this.sampleCount + " training sets.");

		allocateTrainingSets();

		copyTrainingSets();

		trainNetworkBackpropBackprop();
		SerializeObject.save(Config.FILENAME_WHENBORN_NET,
			this.network);
		SerializeObject.save(Config.FILENAME_HISTOGRAM,
			this.histogramGood);
		System.out.println("Training complete.");

	}

	private void trainNetworkBackpropBackprop() {
		final Train train =
			new Backpropagation(this.network, this.input,
				this.ideal, 0.7, 0.7);

		int epoch = 1;

		do {
			train.iteration();
			System.out.println("Backprop:Iteration #"
				+ epoch + " Error:"
				+ train.getError());
			epoch++;
		} while ((train.getError() > Config.ACCEPTABLE_ERROR));
	}
}
```

The **`process`** method is the first method that is called by the **`main`** method of the **`TrainBot`** class. The **`process`** method will be discussed in the next section.

Processing the Training Sets

The **`process`** method begins by creating a neural network and reporting that the training sets are being prepared.

```
this.network = NetworkUtil.createNetwork();
System.out.println("Preparing training sets...");
```

Only the top 1,000 most common English words are allowed to influence the neural network. Occurrences of these common words in both the good and bad sentence sets will be analyzed. Histograms, which will be covered later in this chapter, keep track of the number of occurrences of each of the common words in both the good and bad sentences.

```
this.common = new CommonWords(Config.FILENAME_COMMON_WORDS);
this.histogramGood = new WordHistogram(this.common);
this.histogramBad = new WordHistogram(this.common);
```

First, the good words are loaded into their histogram.

```
this.histogramGood.buildFromFile(Config.FILENAME_GOOD_TRAINING_
TEXT);
this.histogramGood.buildComplete();
```

Next, the bad words are loaded into their histogram.

```
this.histogramBad.buildFromFile(Config.FILENAME_BAD_TRAINING_TEXT);
this.histogramBad.buildComplete();
```

We would like to remove any words that occur in both the good and bad training lists. The occurrence counts are used to trim the lists; there is likely to be considerable overlap. To overcome this overlap, we remove any words from the good histogram that are below the average number of occurrences. We then remove the lowest 99% of bad words.

```
this.histogramGood
    .removeBelow((int) this.histogramGood.calculateMean());
this.histogramBad.removePercent(0.99);
```

Next, we remove the words that appear in both the good and bad histograms.

```
// remove common words
this.histogramGood.removeCommon(this.histogramBad);
```

Finally, we trim the good histogram to the number of inputs.

```
this.histogramGood.trim(Config.INPUT_SIZE);
```

We then analyze the good and bad sentences. This will allow us to create both an input array and an ideal array based on each sentence. The input array reflects how many of the good words were present in each sentence. The output array reflects whether this sentence contains the birth year or not. We begin by allocating two **`AnalyzeSentence`** objects.

```
this.goodAnalysis = new AnalyzeSentences(this.histogramGood,
    Config.INPUT_SIZE);
this.badAnalysis = new AnalyzeSentences(this.histogramGood,
    Config.INPUT_SIZE);
```

We then process both the good and bad sentences. They will all be added to the **`trainingSet`** collection that is passed to the method.

```
this.goodAnalysis.process(this.trainingSet, 0.9,
    Config.FILENAME_GOOD_TRAINING_TEXT);
this.badAnalysis.process(this.trainingSet, 0.1,
    Config.FILENAME_BAD_TRAINING_TEXT);
```

Notice the values of 0.9 and 0.1 above. These are the ideal output values. The closer the value of the output neuron is to 0.9, the more likely that the sentence contains the birth year of the famous person. The closer it is to 0.1, the less likely.

Next, we report the number of training sets that were collected.

```
this.sampleCount = this.trainingSet.getIdeal().size();
System.out
    .println("Processing " + this.sampleCount + " training
sets.");
```

To allow the training sets to grow easily, they are stored as a collection. In order to use them to train the neural network, they must be converted into the usual 2-dimensional array, as were the samples used to train previous neural networks in this book. We begin by allocating the training sets.

```
allocateTrainingSets();
```

Next, the **`trainingSet`** collection is copied into the actual training sets.

```
copyTrainingSets();
```

The neural network is now trained using backpropagation.

```
trainNetworkBackpropBackprop();
```

The neural network and the histogram are then saved.

```
SerializeObject.save(Config.FILENAME_WHENBORN_NET, this.network);
SerializeObject.save(Config.FILENAME_HISTOGRAM,
this.histogramGood);
System.out.println("Training complete.");
```

The training is now complete.

Analyzing Sentence Histograms

A histogram is a linear progression of frequencies. The histograms in this application are managed by the **WordHistogram** class. They are used to store the frequency with which each common word occurs in a sentence. The **WordHistogram** class is shown in Listing 13.9.

Listing 13.9: Managing Histograms (WordHistogram.java)

```
package com.heatonresearch.book.introneuralnet.ch13.train;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.Serializable;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.StringTokenizer;
import java.util.TreeSet;

import com.heatonresearch.book.introneuralnet.ch13.CommonWords;
import com.heatonresearch.book.introneuralnet.ch13.Config;


/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * WordHistogram: Build a histogram of how many occurrences of
 * each word.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class WordHistogram implements Serializable {

      /**
       * Serial id for this class.
       */
      private static final long serialVersionUID =
4712929616016273693L;

      public static void main(final String args[]) {
```

```
try {
      final CommonWords common = new CommonWords(
                  Config.FILENAME_COMMON_WORDS);
      final WordHistogram histogramGood =
            new WordHistogram(common);
      final WordHistogram histogramBad =
            new WordHistogram(common);

      // load the good words
      histogramGood.buildFromFile(
            Config.FILENAME_GOOD_TRAINING_TEXT);
      histogramGood.buildComplete();

      // load the bad words
      histogramBad.buildFromFile(
            Config.FILENAME_BAD_TRAINING_TEXT);
      histogramBad.buildComplete();

      // remove low scoring words
      histogramGood.removeBelow((int)
            histogramGood.calculateMean());
      histogramBad.removePercent(0.99);

      // remove common words
      histogramGood.removeCommon(histogramBad);
      // histogramBad.removeCommon(histogramGood);

      System.out.println("Good Words");
      for (final HistogramElement element :
            histogramGood.getSorted()) {
            System.out
                        .println(
      element.getWord() + ":" + element.getCount());
      }
      System.out.println("\n\n\n");
      System.out.println("Bad Words");
      for (final HistogramElement element :
            histogramBad.getSorted()) {
            System.out
                        .println(element.getWord()
            + ":" + element.getCount());
      }

} catch (final Exception e) {
      e.printStackTrace();
}
```

```
    }

    private final CommonWords common;
    private final Map<String, HistogramElement> histogram =
          new HashMap<String, HistogramElement>();

    private final Set<HistogramElement> sorted =
          new TreeSet<HistogramElement>();

    public WordHistogram(final CommonWords common) {
          this.common = common;
    }

    public void buildComplete() {
          this.sorted.clear();
          this.sorted.addAll(this.histogram.values());
    }

    public void buildFromFile(final InputStream is)
          throws IOException {
          String line;

          final BufferedReader br = new
                BufferedReader(new InputStreamReader(is));
          while ((line = br.readLine()) != null) {
                buildFromLine(line);
          }
          br.close();
    }

    public void buildFromFile(final String filename)
          throws IOException {
          final FileInputStream fis =
                new FileInputStream(filename);
          buildFromFile(fis);
          fis.close();
    }

    public void buildFromLine(final String line) {
          final StringTokenizer tok = new StringTokenizer(line);
          while (tok.hasMoreTokens()) {
                final String word = tok.nextToken();
                buildFromWord(word);
          }
    }
```

```
public void buildFromWord(String word) {
      word = word.trim().toLowerCase();
      if (this.common.isCommonWord(word)) {
            HistogramElement element =
                  this.histogram.get(word);
            if (element == null) {
                  element = new HistogramElement(word, 0);
                  this.histogram.put(word, element);
            }
            element.increase();
      }
}

public double calculateMean() {
      int total = 0;
      for (final HistogramElement element : this.sorted) {
            total += element.getCount();
      }
      return (double) total / (double) this.sorted.size();
}

public double[] compact(final String line) {
      final double[] result = new double[this.sorted.size()];

      final StringTokenizer tok = new StringTokenizer(line);

      while (tok.hasMoreTokens()) {
            final String word = tok.nextToken().
                  toLowerCase();
            final int rank = getRank(word);
            if (rank != -1) {
                  result[rank] = 0.9;
            }
      }
      return result;
}

public int count(final String line) {
      int result = 0;

      final StringTokenizer tok = new StringTokenizer(
            line);

      while (tok.hasMoreTokens()) {
            final String word = tok.nextToken().
            toLowerCase();
```

```
                final int rank = getRank(word);
                if (rank != -1) {
                      result++;
                }
          }
          return result;
    }

    public HistogramElement get(final String word) {
          return this.histogram.get(word.toLowerCase());
    }

    public CommonWords getCommon() {
          return this.common;
    }

    public int getRank(final String word) {
          int result = 0;

          for (final HistogramElement element : this.sorted) {
                if (element.getWord().equalsIgnoreCase(word)) {
                      return result;
                }
                result++;
          }
          return -1;
    }

    public Set<HistogramElement> getSorted() {
          return this.sorted;
    }

    public void removeBelow(final int value) {
          final Object[] elements = this.sorted.toArray();
          for (int i = 0; i < elements.length; i++) {
                final HistogramElement element = (
                HistogramElement) elements[i];
                if (element.getCount() < value) {
                      this.histogram.remove(element.getWord());
                      this.sorted.remove(element);
                }
          }
    }

    public void removeCommon(final WordHistogram other) {
          for (final HistogramElement element :
```

```
		other.getSorted()) {
			final HistogramElement e =
		this.get(element.getWord());
			if (e == null) {
				continue;
			}
			this.sorted.remove(e);
			this.histogram.remove(element.getWord());

		}
	}

	public void removePercent(final double percent) {
		int countdown = (int) (this.sorted.size() * (1 -
		percent));

		final Object[] elements = this.sorted.toArray();
		for (int i = 0; i < elements.length; i++) {
			countdown--;
			if (countdown < 0) {
				final HistogramElement element =
				(HistogramElement) elements[i];
				this.histogram.remove(element.getWord());
				this.sorted.remove(element);
			}
		}
	}

	public void trim(final int size) {

		final Object[] elements = this.sorted.toArray();
		for (int i = 0; i < elements.length; i++) {
			if (i >= size) {
				final HistogramElement element =
				(HistogramElement) elements[i];
				this.histogram.remove(element.getWord());
				this.sorted.remove(element);
			}
		}
	}

}
```

The individual words in the histogram are stored in the **HistogramElement** class. A collection of **HistogramElement** objects is stored inside the **WordHistogram** class. The **HistogramElement** class is shown in Listing 13.10.

Listing 13.10: Histogram Elements (HistogramElement.java)

```
package com.heatonresearch.book.introneuralnet.ch13.train;

import java.io.Serializable;

/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * HistogramElement: An element in the histogram.  This is the
 * word, and the number of times it came up in either the good
 * or bad set.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class HistogramElement implements
Comparable<HistogramElement>,
            Serializable {
      /**
       * Serial id for this class.
       */
      private static final long serialVersionUID =
-5100257451318427243L;
      private String word;
      private int count;

      public HistogramElement(final String word, final int count) {
            this.word = word.toLowerCase();
            this.count = count;
      }

      public int compareTo(final HistogramElement o) {
            final int result = o.getCount() - getCount();
            if (result == 0) {
                  return this.getWord().compareTo(o.getWord());
            } else {
                  return result;
            }
      }

      /**
       * @return the count
       */
      public int getCount() {
            return this.count;
      }
      /**
       * @return the word
       */
      public String getWord() {
            return this.word;
      }

      public void increase() {
            this.count++;
      }

      /**
       * @param count
       *            the count to set
       */
      public void setCount(final int count) {
            this.count = count;
      }

      /**
       * @param word
       *            the word to set
       */
      public void setWord(final String word) {
            this.word = word.toLowerCase();
      }

}
```

The **HistogramElement** class is very simple. It stores a word and the number of times the word occurs. You can see both of these values in the above listing. Additionally, the **HistogramElement** class implements the **Comparable** interface, which it uses for sorting. If you examine the above listing, you will see that the **compareTo** method causes the sorting to occur first by the number of times a word occurs, and then alphabetically by the word itself.

The **WordHistogram** class, as seen in Listing 13.9, provides many services. These are used by the neural bot in all three modes of operation. These services include the following:

- Count the number of times words occur to build a histogram
- Compact a sentence to an input pattern
- Remove common words that also appear in another histogram
- Remove the bottom percentage of words
- Calculate the average number of occurrences of the words in the histogram

One of the most important functions of the **WordHistogram** class is to compact a sentence into an input pattern for the neural network. The input pattern for the neural network is simply a relative count of the number of occurrences of the most common "good words" in the sentence. The number of good words expressed in the input pattern is the number of input neurons. The single output neuron specifies the degree to which the neural network believes the sentence contains a birth year.

The input pattern is built inside the **compact** method of the **WordHistogram** class. The signature for the **compact** method is shown here:

```
public double[] compact(final String line)
```

The result is the same size as the number of words in this histogram. This size is the same as the number of input neurons in the neural network.

```
final double[] result = new double[this.sorted.size()];
```

A **StringTokenizer** object is used to parse the sentence into words.

```
final StringTokenizer tok = new StringTokenizer(line);
```

We continue to loop as long as there are more tokens to read.

```
while (tok.hasMoreTokens()) {
```

The next word is then obtained and converted to lowercase.

```
  final String word = tok.nextToken().toLowerCase();
```

The rank for the word is then given. If the word is present, the input neuron is set to a relatively high 0.9.

```
  final int rank = getRank(word);
  if (rank != -1) {
      result[rank] = 0.9;
  }
}
```

This process continues while there are more words.

```
return result;
}
```

Once the process is complete, the resulting input array for the neural network is returned. This input array will be used to build training sets or to actually query the neural network, once it has been trained.

Building the Training Sets

The training sets for the neural network are managed by the **TrainingSet** class. The **TrainingSet** class is shown in Listing 13.11.

Listing 13.11: Traning Set Management (TrainingSet.java)

```
package com.heatonresearch.book.introneuralnet.ch13.train;

import java.util.ArrayList;
import java.util.List;

/**
 * Chapter 13: Bot Programming and Neural Networks
 *
 * TrainingSet: Used to build a training set based on both
 * good and bad sentences.
 *
 * @author Jeff Heaton
 * @version 2.1
 */
public class TrainingSet {
      private List<double[]> input = new ArrayList<double[]>();
      private List<double[]> ideal = new ArrayList<double[]>();

      public void addTrainingSet(final double[] addInput, final dou-
ble addIdeal) {
            // does the training set already exist
            for (final double[] element : this.input) {
                  if (compare(addInput, element)) {
                        return;
                  }
            }

            // add it
            final double array[] = new double[1];
            array[0] = addIdeal;
            this.input.add(addInput);
            this.ideal.add(array);

      }

      private boolean compare(final double[] d1, final double[] d2)
{
            boolean result = true;

            for (int i = 0; i < d1.length; i++) {
                  if (Math.abs(d1[i] - d2[i]) > 0.000001) {
                        result = false;
                  }
            }
```

```
			return result;
		}

		/**
		 * @return the ideal
		 */
		public List<double[]> getIdeal() {
			return this.ideal;
		}

		/**
		 * @return the input
		 */
		public List<double[]> getInput() {
			return this.input;
		}

		/**
		 * @param ideal
		 *            the ideal to set
		 */
		public void setIdeal(final List<double[]> ideal) {
			this.ideal = ideal;
		}

		/**
		 * @param input
		 *            the input to set
		 */
		public void setInput(final List<double[]> input) {
			this.input = input;
		}

}
```

Like any other supervised training set in this book, the neural bot training set uses an input array and an ideal array. The most important job of the **TrainingSet** class is to allow new training sets to be added without introducing duplicates. It accomplishes this using the **addTrainingSet** method. The signature for the **addTrainingSet** method is shown here:

```
public void addTrainingSet(final double[] addInput, final
		double addIdeal)
```

Two arguments are passed to the **addTrainingSet** method. The first, **addInput**, specifies the input array. The second, **addIdeal**, specifies the ideal values for the specified input pattern.

We begin by checking to see if the training set already exists. To do this, we loop over all of the elements.

```
// does the training set already exist
for (final double[] element : this.input) {
```

If the training set is already present, then we simply return and ignore it.

```
  if (compare(addInput, element)) {
      return;
  }
}
```

If it does not exist, we add it to the list of sets we have already accumulated.

```
// add it
final double array[] = new double[1];
array[0] = addIdeal;
this.input.add(addInput);
this.ideal.add(array);
```

Once all of the training sets have been added, the neural network can be trained.

Training the Neural Network

Backpropagation is used to train the neural network. This procedure is essentially the same as all other examples of backpropagation in this book. The training occurs in the **`trainNetworkBackprop`** method of the **`TrainBot`** class. The signature for the **`trainNetworkBackprop`** method is shown here:

```
private void trainNetworkBackprop()
```

First, a training object is created. Because this neural network uses a sigmoidal activation function, a relatively high learning rate and high momentum can be specified. For more information on learning rate and momentum refer to chapter 5.

```
final Train train = new Backpropagation(this.network, this.input,
      this.ideal, 0.7, 0.7);
```

The starting epoch is 1.

```
int epoch = 1;
```

We begin a training loop.

```
do {
```

Each epoch of the training is performed and the error at each epoch is reported.

```
  train.iteration();
  System.out.println("Backprop:Iteration #" + epoch + " Error:"
        + train.getError());
  epoch++;
} while ((train.getError() > Config.ACCEPTABLE_ERROR));
}
```

Looping continues as long as we are above the acceptable error level. Once the training is complete, the neural network is saved, as covered earlier in this chapter.

Querying the Neural Bot

Once the bot has been trained, it can be used to lookup the birth year for a famous person. The main bot for performing this function is contained in the package named **`born`**. To execute the bot, the class **`YearBornBot`** is executed. The **`YearBornBot`** class is shown in Listing 13.12.

Listing 13.12: WhenBornBot Class (WhenBornBot.java)

```
package com.heatonresearch.book.introneuralnet.ch13.born;

import java.io.IOException;
import java.net.URL;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import com.heatonresearch.book.introneuralnet.ch13.Config;
import com.heatonresearch.book.introneuralnet.ch13.ScanReportable;
import com.heatonresearch.book.introneuralnet.ch13.Text;
import com.heatonresearch.book.introneuralnet.ch13.train.
WordHistogram;
import com.heatonresearch.book.introneuralnet.common.YahooSearch;
import com.heatonresearch.book.introneuralnet.neural.feedforward.
FeedforwardNetwork;
import com.heatonresearch.book.introneuralnet.neural.util.
SerializeObject;

/**
 * Bot example.
 *
 * @author Jeff Heaton
 * @version 1.1
 */

public class YearBornBot implements ScanReportable {
      public static final boolean LOG = true;
      /*
       * The search object to use.
```

```
	 */
	static YahooSearch search;

	/**
	 * The main method processes the command line arguments
	 * and then calls
	 * process method to determine the birth year.
	 *
	 * @param args
	 *            The person to scan.
	 *
	 */
	public static void main(final String args[]) {

		if (args.length < 1) { System.out.
println("YearBornBot [Famous
		Person]"); } else
		{
			try {
				final YearBornBot when = new YearBornBot();
				// when.process(args[0]);
				when.process("Bill Gates");
			} catch (final Exception e) {
				e.printStackTrace();
			}
		}
	}

	private final FeedforwardNetwork network;

	private final WordHistogram histogram;

	/*
	 * This map stores a mapping between a year, and how many
	 * times that year
	 * has come up as a potential birth year.
	 */
	private final Map<Integer, Integer> results =
		new HashMap<Integer, Integer>();

	public YearBornBot() throws IOException,
		ClassNotFoundException {
		this.network = (FeedforwardNetwork) SerializeObject
				.load(Config.FILENAME_WHENBORN_NET);
		this.histogram = (WordHistogram) SerializeObject
				.load(Config.FILENAME_HISTOGRAM);
}

/**
 * Get birth year that occurred the largest number of times.
 *
 * @return The birth year that occurred the largest
 * number of times.
 */
public int getResult() {
      int result = -1;
      int maxCount = 0;

      final Set<Integer> set = this.results.keySet();
      for (final int year : set) {
            final int count = this.results.get(year);
            if (count > maxCount) {
                  result = year;
                  maxCount = count;
            }
      }

      return result;
}

/**
 * @param year
 */
private void increaseYear(final int year) {
      Integer count = this.results.get(year);
      if (count == null) {
            count = new Integer(1);
      } else {
            count = new Integer(count.intValue() + 1);
      }
      this.results.put(year, count);
}

/**
 * This method is called to determine the birth year for
 * a person. It obtains 100 web pages that Yahoo returns
 * for that person. Each of these pages is then searched
 * for the birth year of that person. Which ever year
 * is selected the largest number of times is selected as
 * the birth year.
 *
 * @param name
 *            The name of the person you are seeing
 * the birth year for.
 * @throws IOException
 *             Thrown if a communication error occurs.
 */
public void process(final String name) throws IOException {
      search = new YahooSearch();

      if (YearBornBot.LOG) {
            System.out.println(
            "Getting search results form Yahoo.");
      }
      final Collection<URL> c = search.search(name);
      int i = 0;

      if (YearBornBot.LOG) {
            System.out.println(
                  "Scanning URL's from Yahoo.");
      }
      for (final URL u : c) {
            try {
                  i++;
                  Text.checkURL(this, u, null);
            } catch (final IOException e) {

            }
      }

      final int resultYear = getResult();
      if (resultYear == -1) {
            System.out.println("Could not determine when "
                  + name
                  + " was born.");
      } else {
            System.out.println(name + " was born in "
                  + resultYear);
      }

}

public void receiveBadSentence(final String sentence) {
      // TODO Auto-generated method stub

}

public void receiveGoodSentence(final String sentence) {
```

```
            if (this.histogram.count(sentence) >=
            Config.MINIMUM_WORDS_PRESENT) {
                  final double compact[] =
                  this.histogram.compact(sentence);
                  final int year = Text.extractYear(sentence);

                  final double output[] =
                  this.network.computeOutputs(compact);

                  if (output[0] > 0.8) {
                        increaseYear(year);
                        System.out.println(year + "-"
                        + output[0] + ":" + sentence);
                  }
            }

      }

}
```

The first method called by the **main** method is the **process** method. This method processes the request to determine the birth year of the famous person. The **process** method is covered in the next section. Other methods called by the **process** method are covered in later sections of this chapter.

Processing the Request

The **process** method accepts the name of the famous person and attempts to determine the birth year for that famous person. The signature for the **process** method is shown here:

```
public void process(final String name) throws IOException
```

The **process** method begins by performing a Yahoo search on the **name** specified.

```
search = new YahooSearch();

if (YearBornBot.LOG) {
  System.out.println("Getting search results form Yahoo.");
}
```

The results of the Yahoo search are stored in the **c** object.

```
final Collection<URL> c = search.search(name);
int i = 0;

if (YearBornBot.LOG) {
  System.out.println("Scanning URL's from Yahoo.");
}
```

We loop through the entire list and process each URL. As each URL is processed, the sentences that contain birth years are returned to the **receiveGoodSentence** method. The **receiveGoodSentence** method will be covered in the next section.

```
for (final URL u : c) {
  try {
      i++;
      Text.checkURL(this, u, null);
```

All errors are ignored. We hit a large number of websites and some are sure to be down.

```
  } catch (final IOException e) {

  }
}
```

As we find years with a high probability of being the birth year, we count the number of times they occur. The **getResult** method returns the birth year that is the most likely candidate. The **getResult** method is covered later in this chapter.

```
final int resultYear = getResult();
```

Negative one is returned if no likely birth year is found.

```
if (resultYear == -1) {
  System.out.println("Could not determine when " + name
        + " was born.");
```

Otherwise, the most likely birth year is found.

```
} else {
  System.out.println(name + " was born in " + resultYear);
}
```

The **process** method made use of several other methods as it executed. The next two sections will cover these methods.

Receiving Sentences

Just as the gathering process used the **Text** class to parse individual sentences, so does the query process. As the **Text** class parses the data, the **receiveGoodSentence** method is called each time a sentence is identified. The signature for the **receiveGoodSentence** is shown here:

```
public void receiveGoodSentence(final String sentence)
```

First, we check to see if the sentence found has the minimum number of recognized words. If it does not, then the sentence is ignore.

```
if (this.histogram.count(sentence) >= Config.MINIMUM_WORDS_PRESENT)
```

```
{
```

The sentence is compacted into an input array for the neural network.

```
  final double compact[] = this.histogram.compact(sentence);
```

The potential birth year is obtained from the **Text** class.

```
  final int year = Text.extractYear(sentence);
```

The output from the neural network is returned using the input array generated from the current sentence.

```
  final double output[] = this.network.computeOutputs(compact);
```

If the specified sentence causes an output greater than 0.8, then there is a decent chance that this is a birth year. The count is increased for this year and the year is reported.

```
  if (output[0] > 0.8) {
      increaseYear(year);
      System.out.println(year + "-" + output[0] + ":" + sentence);
  }
}
```

Finding the Best Birth Year Candidate

Once all of the year counts have been updated, the **getResult** method is called to determine which year has the highest count. This is the year that will be proposed as the birth year for this person. The signature for the **getResult** method is shown here:

```
public int getResult()
```

The **result** is set to a default of -1. If no potential birth years are found, then this -1 value is returned. The **maxCount** variable is used to track which year has the maximum count.

```
int result = -1;
int maxCount = 0;
```

All of the years identified as potential birth years are reviewed.

```
final Set<Integer> set = this.results.keySet();
for (final int year : set) {
```

The count for the specified year is obtained.

```
  final int count = this.results.get(year);
  if (count > maxCount) {
      result = year;
      maxCount = count;
  }
```

```
}
```

We continue to loop and update the result as better candidates are found.

```
return result;
}
```

This process continues until the best candidate has been selected.

Chapter Summary

Bots are computer programs that perform repetitive tasks. Often, a bot makes use of an HTTP protocol to access websites in much the same way as do humans. Because a bot generally accesses a very large amount of data, neural networks can be useful in helping the bot understand the incoming data.

This chapter presented a neural network that attempts to determine the birth year for a famous person. This neural network is divided into three distinct modes of operation. The first mode uses a threaded bot to gather lots of information on the birth year of specified famous people. Using these famous people and their birth years, the bot is trained in the second mode. The final mode allows the user to query the network and determine the birth years of other famous people.

This book has covered a number of different types of neural networks and methods for using them. Despite all they are able to do, neural networks still fall considerably short of achieving the ability to reason like a human brain. The next chapter will look at the current state of neural network research and where neural networks are headed in the future.

Vocabulary

Bot

Histogram

HTTP Protocol

Parsing

Questions for Review

1. How does this chapter suggest to transform a sentence into an input pattern for the neural network?

2. Other than the method discussed in this chapter, how else might a sentence be transformed into input for a neural network? Is it necessary for every sentence to produce the same number of inputs for the neural network?

3. Why was the sigmoidal activation function a good choice for an activation function for the neural bot?

4. What is text parsing, and how is it useful?

5. If you wanted to expand this bot to also locate the name of the famous person's spouse, how might this be done?