1-2008

A Data Mining Address Book

Douglas A. Lyon
Fairfield University, dlyon@fairfield.edu

Published Citation

Douglas Lyon, "A Data Mining Address Book", Journal of Object Technology, Volume 7, no. 1 (January 2008), pp. 15-26

JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©JOT, 2007

Vol. 7, No. 1, January-February 2008

A Data Mining Address Book

Douglas Lyon, Ph.D.

Abstract

This paper describes how to use web-based data mining to populate a flat-file database called the *JAddressBook*. The *JAddressBook* represents a next-generation address book program that is able to use web-based data mining. As it is also able to print mailing labels, and even initiate phone calls, it is useful for marketing. More over, its introduction to data mining has educational value, for those new to network programming, in Java.

The methodology for converting the web data source into internal data structures is based on using HTML as input (called screen scraping). We explore a variety of techniques for reading the HTML data, as input. Our example focuses on mining data for lawyers.

1 THE PROBLEM

Web-based data is generally available in an HTML format. Given a web-based source of an HTML formatted database, we would like to find a way to create an underlying data structure that is type-safe and well formulated, in response to a given query. Basically, we want POJOs (Plain Old Java Objects), extracted from HTML.

The motivation for studying this type of problem ranges from the educational to the compelling next-generation killer applications (e.g., a 3rd generation address book). The web is a source of data that is probably here to stay. It is almost certainly the largest source of data, as well as the fastest growing. Considering the general lack of the adoption of standards for the presentation of the data (particularly by those who would prefer not to share the data), we would like to take the data in an unstructured format and populate our data structures.

Once we have the data, in our own database, we are at liberty to make use of it for a variety of applications. We will show an application that is able to print mailing labels from the addresses, for the purpose of marketing. We find students in our network programming classes are interested in data mining and that using real-world sources of data encourages students to become very creative in their approach to parsing the data.

2 FINDING THE DATA

Finding freely available data, on-line, is a necessary first step toward this type of data mining. Databases of lawyers are available from a web site called *lawyers.com*. The

Cite this column as follows: Douglas A. Lyon "A Data Mining Address Book", in *Journal of Object Technology*, vol. 7, no. 1, January - February 2008, pp. 15-26

URL that can obtain a list of intellectual property attorneys in the state of Connecticut looks like:

http://www.lawyers.com/Intellectual-Property/CONNECTICUT/All-Cities/law-firms-p1.html?searchtype=Q

This creates an output on the screen that looks like:

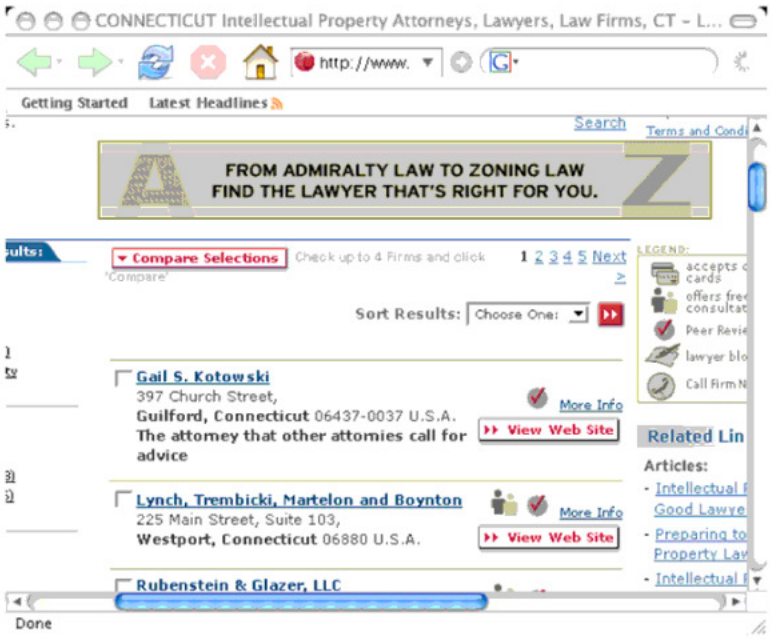


Figure 2-1. Sample Output

The output shown in Figure 2-1 is typical of the type of output encountered on the web. There are all manner of distracting artifacts on the page. Javascript, advertisements, side articles, unrelated buttons, etc. Most disconcerting is the lack of any indication of the number of records or number of pages needed in order to obtain the entire result set from the query. That is, by clicking on the *Next* button, we get to scroll to an entire new page of results from our query, but we don't know how many times we need to click *Next* in order to exhaust the result set.

We break the problem into two sub-parts. The first is the creation of a URL for obtaining data; the second is the analysis of the results. To synthesize the URL needed to get the data, we use:

```
public static URL getLaywerUrl(String lawyerType, String stateName, int pageNumber)
        throws MalformedURLException {
    //http://www.lawyers.com/
    // Criminal/New-York/All-Cities/law-firms-p8.html?searchtype=Q&site=466
    String urlString = "http://www.lawyers.com/" +
        lawyerType +
        "/" +
        stateName +
        "/All-Cities/law-firms-p" +
        pageNumber +
        ".html?searchtype=Q";
    System.out.println(urlString);
    return new URL(
```

```
            urlString);
    }
```

There are three parameters used to create this URL, the *lawyerType* and *stateName* come from lists that are presented to the user via a dialog box.

In order to fetch the data, given the URL, we write a simple helper utility that contains:

```
public static String getLawyerText(String lawyerType,
                                   String stateName,
                                   int pageNumber)
        throws IOException, BadLocationException {
    return UrlUtils.html2text(getLawyerUrl(lawyerType, stateName, pageNumber));
}
```

The heart of the program is *html2text,* and this is covered in the following section.

3 ANALYSIS

Html2text converts the HTML referenced by the URL into text, stripping out JavaScript, HTML, etc. and leaving one large text string that can be used for the purpose of data mining. There are many approaches to writing such a program (and we have tried a few of them):

```
public static String html2text(URL url)
        throws IOException, BadLocationException {
    InputStreamReader inputStreamReader = new InputStreamReader(
            url.openConnection().getInputStream());
    final StringBuffer stringBuffer = new StringBuffer(1000);
    EditorKit editorKit = new HTMLEditorKit();
    final HTMLDocument htmlDocument = new HTMLDocument() {
        public HTMLEditorKit.ParserCallback getReader(int pos) {
            return new TagStripper(stringBuffer);
        }
    };
    htmlDocument.putProperty("IgnoreCharsetDirective", Boolean.TRUE);
    try {
        editorKit.read(inputStreamReader, htmlDocument, 0);
    }
    catch (ChangedCharSetException e) {
        //If the encoding is incorrect, get the correct one
        inputStreamReader = new InputStreamReader(
                url.openConnection().getInputStream(), e.getCharSetSpec());
        try {
            editorKit.read(inputStreamReader, htmlDocument, 0);
        }
        catch (ChangedCharSetException ccse) {
            System.out.println("Couldn't set correct encoding: " + ccse);
```

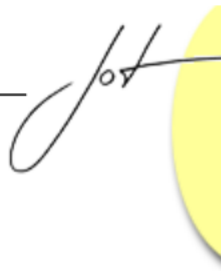
```
            }
        }
        return stringBuffer.toString();
    }
```

The *EditorKit* in Swing makes use of the *inversion-of-control* design pattern [GoF]. Inversion-of-control is a means by which a callback method is supplied in order to write a component that can be used in a larger framework. Basically, a class called *TagStripper* is used to remove the HTML tags found in the document. The removal is done by a series of handlers that are invoked when HTML tags are encountered:

```
package net.web;
import javax.swing.text.html.*;
import javax.swing.text.MutableAttributeSet;

public class TagStripper extends HTMLEditorKit.ParserCallback {
   private StringBuffer sb;

   public TagStripper(StringBuffer out) {
      this.sb = out;
   }

   public void handleEndTag(HTML.Tag t, int pos) {
      if (t.equals(HTML.Tag.TD))
         sb.append("<newRecord>\n");
      if (t.equals(HTML.Tag.STRONG))
         sb.append("|");
   }

   public void handleStartTag(HTML.Tag t, MutableAttributeSet a, int pos){
      if (t.equals(HTML.Tag.B))
         sb.append("|");
      if (t.equals(HTML.Tag.BR))
         sb.append("|");
   }

   public void handleText(char[] text, int position) {
      sb.append(text);
   }
}
```

The tags of interest include *TD* (Table Data), *Strong*, *B* (Bold), and *BR* (line break). We assume that table data signifies a new record, and so insert the *<newRecord>* tag in order to ease parsing, downstream. The other tags are used to denote fields within the record. Processing data, using an *HTMLEditorKit,* makes data mining of text data a little easier enabling us to work at a higher level. The output looks like:

```
<newRecord>
Webb & Eley, P.C.|7475 Halcyon Pointe Drive, |Montgomery, Alabama 36124 U.S.A.
General Civil and Appellate Practice, Corporate Business Law, Commercial Law,
Insurance Defense, Health Care, Civil Rights, Workers Compensations
```



In order to determine how many results are available, we can inspect the HTML output of a query. For example:

…found 1111 listings…

This indicates that there are 1,111 records in the result set. Also, from inspection of the web page, we conclude that there may be up to 25 records per page. Thus, we require 45 queries (1,111/25) in order to screen scrape the results. The ad-hoc nature of the parsing scheme is even more evident when we examine the string manipulations needed to obtain low-level data types:

```
public static int getNumberOfAddresses(String matcherString){
    //found 325 listings
    Matcher matcher =
            Pattern.compile(
                "([0-9]+).+ listings").matcher(matcherString);
    matcher.find();
    if (matcher.groupCount()==1) {
       String x = matcher.group(1);
       return Integer.parseInt(x);
    }
    return -1;
  }
```

The use of the *regexp* "([0-9]+).+ listings" introduces a more powerful string matching tool than the low-level string processing (e.g. *indexOf* or *StringTokenizer*). "[ ]" matches a single character that is contained within the brackets. For example, [012] matches "0", "1", or "2". "[0-9]" matches any digit. The plus sign indicates that there is at least 1 of the previous expression. "." Matches any single character. Thus we are searching for a string of digits followed by any character followed by the word "listings". Once we know how many records there are and how many pages there are, we write a loop to process the data;

```
Lawyers.numTotal = LawyerSearchUtils.getNumberOfAddresses(
            getLawyerText(lawyerType, stateName, 1));
    int numberOfPages = (int) Math.ceil(Lawyers.numTotal / 25.0);
    final boolean[] stopFlag = new boolean[]{false};
    ProgressDialog2 pd = getProgressDialog(stopFlag);
    AddressDataBase adb = AddressDataBase.getAddressBookDatabase();
    for (int i = 1; i <= numberOfPages; i++) {
       process(getLawyerText(lawyerType, stateName, i),
            adb, pd);
       if (stopFlag[0]) break;
    }
    pd.setVisible(false);
```

From a human interface point-of-view, data mining is a time-consuming operation, so it is good practice to provide a progress dialog box (with a cancel button) to keep the user updated about the state of the search. That is why we pass the progress dialog

into the process method. The *AddressDataBase* is obtained from a singleton-pattern class that is used as a flat-file storage facility for our result set.

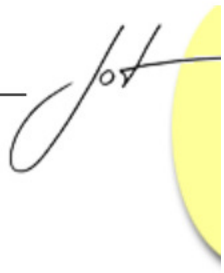
What follows is an ad-hoc parsing scheme, using *StringTokenizer* to detect new lines and regular expressions to parse address records:

```
private static void process(String text, AddressDataBase adb, ProgressDialog2 pd) {
        System.out.println("Process.....");
    StringTokenizer st = new StringTokenizer(text, "\n");
    while (st.hasMoreTokens()) {
        String line = st.nextToken();
        if (line.equals("<newRecord>")) continue;
        Pattern pattern = Pattern.compile(".*,.*,.*,.*[0-9]{5}.*<newRecord>");
        Matcher matcher = pattern.matcher(line);
        if (matcher.find()) {

            // Progress Bar Updating
            long currentTime = System.currentTimeMillis();
            long timeElapsed = (currentTime - Lawyers.timeStart) / 1000;
            int percentage = (int) (100 * (double) Lawyers.numFound / (double)
Lawyers.numTotal);
            pd.setText("Found " + Lawyers.numTotal + " Lawyers.  Processing: " +
Lawyers.numFound + " ("
                    + percentage
                    + "%). About " + (percentage != 0 ? (((timeElapsed * 100) / percentage) -
timeElapsed) : "-")
                    + " seconds left.");
            pd.setValue(Lawyers.numFound);

            // Line Parsing
            Lawyers.numFound++;
            line = line.replaceAll("<newRecord>", "");
            line = line.replaceAll("U.S.A.", "|");
            System.out.println("NEW LINE");
            System.out.println(line);

            adb.addRecord(getAddressRecord(line));
            adb.sort();
            adb.update();
            IndexPanel.getRunIndexPanel().updateLabels();
        }
    }
}
```

The heart of the parser is in the regexp:

Pattern pattern = Pattern.compile(".*,.*,.*,.*[0-9]{5}.*<newRecord>");

The only new feature is "[0-9]{5}.*<newRecord>" which says that we are looking for exactly 5 digits, followed by any number of characters, followed by "<newRecord>". The assumption is that 5 digits at the end of a record must be a zip code (9 digits with a hyphen will work too). This is a critical assumption. Should the



suppler alter the format of the data, this code will break (and has done so in the past). The question of how to make this type of processing more robust remains open.

To obtain the *AddressRecord* we proceed to use an ad-hoc application of a *StringTokenizer*:

```
private static AddressRecord getAddressRecord(String s) {
    AddressRecord ar = new AddressRecord();
    StringTokenizer st = new StringTokenizer(s, "|");
    for (int i = 0; st.hasMoreTokens(); i++) {
       String s1 = st.nextToken();
       if (i == 0) ar.setName(s1);
       if (i == 1) ar.setAddress(s1);
       // this is the city, state zip, replace full state name with two
       // letter symbol
       if (i == 2) {
          String letterAbbreviation = LawyerSearchUtils.getTwoLetterAbbreviation(s1);
          System.out.println("letterAbbrevation:"+letterAbbreviation);
          ar.setAddress(ar.getAddress() + "\n" +
                letterAbbreviation);
       }
       if (i == 3) ar.setInfo(s1);
       if (i > 3) ar.setInfo(ar.getInfo()+"\n"+s1);
       System.out.println(s1);
    }
    System.out.println(ar);
    return ar;
 }
```

The *AddressRecord* contains the name, address, information and some phone numbers:

```
public class AddressRecord
      implements Serializable, Comparable {
   private String name = "";
   private String address = "";
   private String info = "";
   private String dial1 = "";
   private String dial2 = "";
   private String dial3 = "";…
```

For the purpose of this type of this application, phone numbers are not present in the data, and so this area of the record is left blank.

4 DISPLAY

We are interested in a new "killer application" for development, called the *JAddressBook* program. This program is able to display stock quotes (and manage an

address book, dial the phone, print labels, do data-mining, etc.). The program can be run (as a web start application) from:

http://show.docjava.com:8086/book/cgij/code/jnlp/addbk.JAddressBook.Main.jnlp

And provides an interactive GUI for data mining addresses.

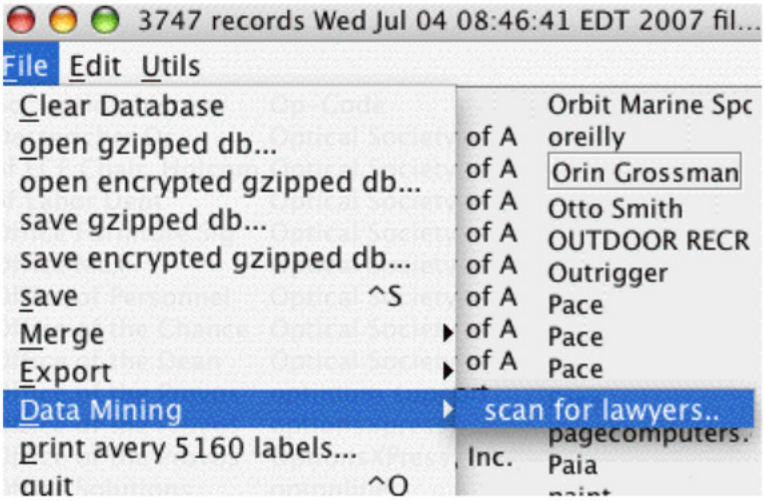


Figure 4-1. The Scanning for lawyers

Figure 4-1 shows an image of the *data mining addressbook*. Users are asked to select a state from a standard list of states (using two-letter postal codes):

Figure 4-2. State Selection Dialog

Figure 4-2 shows an image of the state selection dialog.

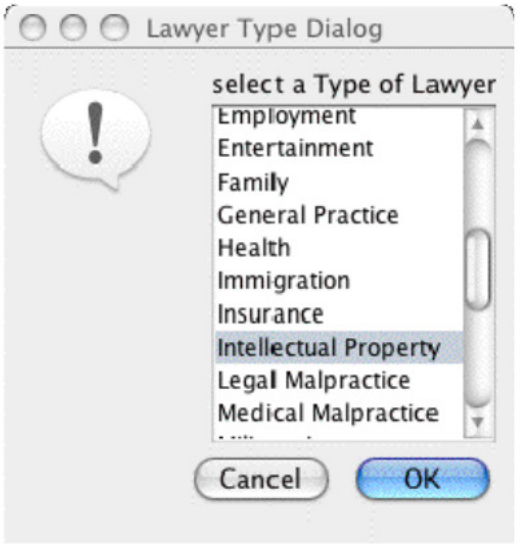


Figure 4-3. The Lawyer Type Dialog

Figure 4-3 shows an image of the lawyer type dialog.

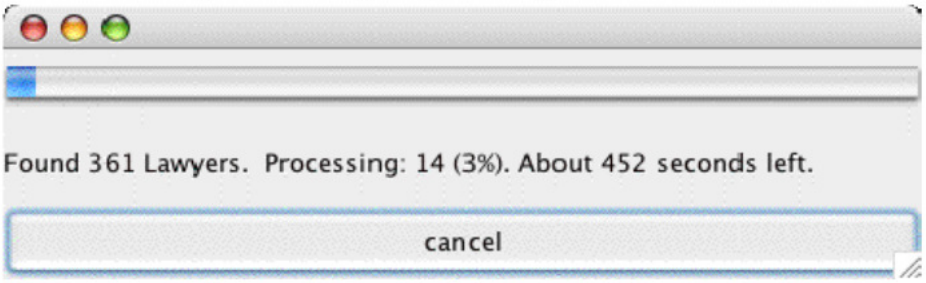


Figure 4-4. The Progress Dialog

The progress dialog is updated, dynamically, in order to provide feedback to the user. Considering the amount of time it takes (several minutes is not that unusual) a multi-threaded feedback mechanism improves the usability of the application.

5 IMPLEMENTING AN ADDRESSBOOK

In order to provide a consistent rendering of the address book database, the singleton design pattern is used. The class is made final, so that it cannot be sub-classed. The constructor is left private, so that the class cannot be instanced by other classes:

```
public final class AddressDataBase extends Observable {
   private Vector addressVector = new Vector();
   private int recordNumber = 0;

   private boolean modifiedButNotSaved = false;

   private static AddressDataBase adb = new AddressDataBase();

   public static AddressDataBase getAddressBookDatabase() {
      return adb;
   }
```

```
    private AddressDataBase() {
       restoreGzDb();

    }
```

The database is stored in a compressed serialized gzipped file. If the file name cannot be retrieved, the database is initialized with a single blank record. Databases can be stored in a variety of different formats (XML, CSV, etc.). Also, data can be merged from a variety of different formats.

```
private void restoreGzDb() {
       String file = getSaveFileName();
       if (file == null) {
          addRecord(new AddressRecord());
          return;
       }

       try {
          openGzDb(file);
       } catch (IOException e) {
          In.message(e);
       }
       top();
       modifiedButNotSaved = false;
    }
```

6 RESOURCE BUNDLING

The file name is stored in the users' preferences, as described in described in [Lyon 05A].

```
public String getSaveFileName() {
       PreferencesBean pb = PreferencesBean.restore();
       return pb.getFileName();
    }
```

This enables the program to remember where the last database file was held. By making use of the user preferences in this way, the location of the database file is user-root specific. This means that different users will get different files when they make use of different accounts:

```
public class PreferencesBean implements Serializable{
    private static final String key = "JAddressBook.PreferencesBean";
    private String fileName = null;
    /**
     * saves the properties to the Preferences of the userRoot
     */
```

```
    public void save() {
        try {
            Preferences p = Preferences.userRoot();
            ByteArrayOutputStream baos = new
                    ByteArrayOutputStream();
            ObjectOutputStream oos = new
                    ObjectOutputStream(baos);
            oos.writeObject(this);
            baos.close();
            p.putByteArray(key, baos.toByteArray());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public String toString() {
        return CompactJava.toXml(this);
    }

    /**
     * restores the properties from the preference in the user root.
     */
    public static PreferencesBean restore() {
        try {
            Preferences p = Preferences.userRoot();
            byte b [] = p.getByteArray(key, null);
            if (b == null)
                return new PreferencesBean();
            ByteArrayInputStream bais = new
                    ByteArrayInputStream(b);
            ObjectInputStream ois = new
                    ObjectInputStream(bais);
            Object o = ois.readObject();
            bais.close();
            return (PreferencesBean) o;
        } catch (IOException e) {
            //e.printStackTrace();
        } catch (ClassNotFoundException e) {
            // e.printStackTrace();
        }
        return new PreferencesBean();
    }

    public String getFileName() {
        return fileName;
    }

    public void setFileName(String fileName) {
        this.fileName = fileName;
    }
}
```

Serialization of the entire address book to the user root preferences is not possible (due to size limits). However, the preferences can easily handle a file name. Should that file name become corrupt, the opening of a new file (from the *file menu*) will reset the location of the most recently used file.

7 CONCLUSION

We showed how an *EditorKit*, regular expression and *StringTokenizers* were combined using ad-hoc parsing techniques to obtain data from the web. Variation in data format on the web remains a topic of concern for the robustness of this class of programs. As we proceed to attack more sophisticated data mining tasks we refine our toolkit. Thus, we are still learning by experimenting with new sources of data.

The web represents a hodge-podge of data sources and parsing is proving to be a hard task (though very worth-while!). The more general the data format that we attempt to attack; the more sophisticated our tools become.

Our program has broken at least once before (as the source of the data changed format). The question of how to deal with changing data formats remains open. A generalized means of providing some kind of meta-data description that gives semantic meaning to HTML documents is needed. HTML tags serve as important landmarks when performing this type of language processing. The most general approach might convert HTML into XML, with specific semantic tags. The question of how this is implemented remains open.

REFERENCES

[Lyon 05A] "Resource Bundling for Distributed Computing," by Douglas A. Lyon, Journal of Object Technology , vol. 4, no. 1, January-February 2005, pp. 45-58. http://www.jot.fm/issues/issue_2005_01/column4/

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA, Addison-Wesley, 1995.

About the author

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 30 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.