Forest Cover Type Prediction

ECE 539 Term Project

Brett Meyer
9013360269
bhmeyer@students.wisc.edu

12/21/01

Introduction

Although machine-learning theory is still in its infancy, new applications of it are appearing every day. Given the processing power of even the smallest computer chips, classification problems with small data sets can be modeled with relative ease. This project intended to explore the difficulties of large data sets, where issues of file size and computation time are apparent.

Selected from the University of California-Irvine Knowledge Discovery in Databases Archive (http://kdd.ics.uci.edu/), the data set selected for this project was the Forest Cover Type data set compiled initially for use by Jock A. Blackard. in his comparison of linear discriminant analysis and back-propagation. In his disertation, Blackard compared the classification abilities of linear discriminant analysis with a back-propagation algorithm. Blackard was able to achieve 70% classification with a back-propagation network, compared to 58% with linear discriminant analysis (Blackard, 1999). The goal of this project was to match, if not exceed, Blackard's results.

The support vector machine class of learning algorithms was chosen as the method of classification. The LibSVM implementation was selected to carry it out because of its ease of use. This implementation provided a remarkably easy, yet robust interface for performing support vector machine classification.

Support Vector Machine Basics

The goal of a support vector machines is to find hyper-planes that separate data points into their respective classes. The better the separation achieved, the better the data

classification that's ultimately possible. In order to determine the equation of the hyper-plane, the support vector machine searches for those data points the lie the closest to data points of another class. These points are called "support vectors."

Linear Hyper-Plane Classifier

The most basic Linear Hyper-Plane Classifier finds support vectors by using a linear hyper-plane in two-dimensional space. The number of input vectors defines the dimension of the input space. The simplest case has two input variables and thus its input space is two-dimensional. If there are two linearly separable classes of data, the goal is to find a line that separates the two classes from each other, thereby establishing the input values that define the two classes. The following are class definitions, where $\mathbf{w}$ is an adjustable weight vector, $\mathbf{x}$ is an input vector and d is a class identifier.

$$\mathbf{w}^T\mathbf{x} + b \geq 0 \quad for\ d_i = +1$$
$$\mathbf{w}^T\mathbf{x} + b \leq 0 \quad for\ d_i = -1$$

Therefore, the optimal hyper-plane is defined as

$$\mathbf{w}_0^T\mathbf{x} + b_0 = 0\,.$$

This weight vector $\mathbf{w}_0$ and the bias b_0 provide the maximum possible separation between the two classes. Quadratic optimization is used to find this optimal hyper-plane, the first step being to construct the Lagrangian function

$$J(\mathbf{w}, b, a) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{N} a_i d_i \mathbf{w}^T\mathbf{x}_i - b\sum_{i=1}^{N} a_i d_i + \sum_{i=1}^{N} a_i$$

Given the training sample $\{(\mathbf{x}_i, d_i)\}_{i=1}^{N}$, the Lagrange multipliers $\{a_i\}_{i=1}^{N}$ are calculated such the cost function

$$Q(a) = \sum_{i=1}^{N} a_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i a_j d_i d_j \mathbf{x}_i^T\mathbf{x}_j$$

is maximized, where

$$\sum_{i=1}^{N} a_i d_i = 0$$
$$a_i \geq 0 \qquad for\ i = 1, 2, \ldots, N$$

.

Once $a_{0,i}$, has been determined, the weights and bias may be calculated as

$$\mathbf{w}_0 = \sum_{i=1}^{N} a_{0,i} d_i \mathbf{x}_i$$
$$b_0 = 1 - \mathbf{w}_o^T \mathbf{x}^{(s)} \quad for\ d^{(s)} = 1$$

(Haykin 1999, pp. 321, 324).

Non-Separable Hyper-Plane Classifier

Most data sets are not made up of separable data. Non-separable data implies that it may be impossible to find a hyper-plane that perfectly divides the input space with the different classes on either side. In order to allow for data points that lie within the separation margin, the regularization parameter *C* is introduced, giving us the function

$$Q(a) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j d_i d_j \mathbf{x}_i^T \mathbf{x}_j ,$$

where

$$\sum_{i=1}^{N} a_i d_i = 0$$
$$0 \leq a_i \leq C \qquad for\ i = 1, 2, \ldots, N$$

.

Once $a_{0,i}$ is determined, the optimum weights are found in the same manner as before.

Non-Linear Inner-Product Kernels

Non-linear inner-product kernels create a mapping where the decision surface in the input space is non-linear while its image in the feature space is linear (Haykin 1999, p. 323). This mapping makes it possible to use of the same optimization used for linear

hyper-plane classifiers for more robust non-linear classifiers. This, in turn, allows for the classification of data that does not fit a linear model.

As long as the kernel $K(\mathbf{x}_i, \mathbf{x}_j)$ satisfies Mercer's Theorem, (Mercer, 1908; Courant and Hilbert, 1970) the constrained optimization is defined as

$$Q(a) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j d_i d_j K(\mathbf{x}_i, \mathbf{x}_j),$$

where

$$\sum_{i=1}^{N} a_i d_i = 0$$
$$0 \le a_i \le C \qquad for\ i = 1, 2, \ldots, N$$

.

After determining $a_{0,i}$, the weight vector of the hyper-plane is calculated as

$$\mathbf{w}_0 = \sum_{i=1}^{N} a_{0,i} d_i j(\mathbf{x}_i),$$

where $\mathbf{f}(\mathbf{x}_i)$ is a set of non-linear transformations from the input space to the feature space and b_0 is the first component of $\mathbf{w}_0$.

Of particular interest are three kernels: polynomial, radial-basis, and sigmoid.

SVM Type	**Kernel**
Polynomial	$\left(\mathbf{x}^T \mathbf{x} + y\right)^p$
Radial-Basis	$\exp\left(-g \cdot \left\|\mathbf{x} - \mathbf{x}\right\|^2\right)$
Sigmoid	$\tanh\left(g \cdot \mathbf{x}^T \mathbf{x} + y\right)$

In the above definitions, figures *y*, *p*, and ? are determined *a priori*.

Data Preprocessing

Raw Data Analysis

The data set was composed of cartographical and geological data gathered from over 581,012 30 x 30 meters square cells of undisturbed forest. The observations consisted of 12 measures, broken into 54 distinct input variables. Of these 54, 10 were quantitative measures while the remaining 44 were Boolean values representing soil conditions. The actual forest type and other independent variables were determined using data available from the US Forest Service and US Geological Survey. See Appendix A for a complete description of the input features.

Each data point was associated with one of seven specific forest cover types. The distribution of these seven classes, however, was uneven.

Forest Cover Type	Number of Records	Percent of Records
Spruce-Fir	211,840	36.4
Lodgepole Pine	283,301	48.8
Ponderosa Pine	35,754	6.1
Cottonwood/Willow	2,747	0.5
Aspen	9,493	1.6
Douglas-Fir	17,367	2.9
Krummholz	20,510	3.5

The data was presented, without scaling, in a comma-delimited list, the last column of each row being the class designation. A sample row is presented below.

2596,51,3,258,0,510,221,232,148,6279,1,0,0,0,0,0,0,0,0,0,0,0, ⋯

0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,5

Visit http://kdd.ics.uci.edu/databases/covertype/covertype.html to download the raw data.

LibSVM Requirements

The goal of preprocessing was to convert raw data into a format that was compatible with the target learning algorithm implementation. In this case, the implementation in question was LibSVM, authored by Chih-Chung Chang and Chih-Jen Lin. LibSVM was a very powerful library of support vector machine implementations, regression and distribution estimation algorithms. For more detailed information regarding the capabilities of LibSVM, visit http://www.csie.ntu.edu.tw/~cjlin/libsvm/index.html.

LibSVM had two major components, a training program and a prediction program. The training program looked at input data and attempted to develop a support vector model. This model was then used by the second component to classify or predict the class label of the testing data. Both the training program and prediction program required the data to be in a specific format, demonstrated below.

<label> <index1>:<value1> <index2>:<value2> … <indexN>:<valueN>

"Label" was the target class value, and the "valueX" fields were the input features. "IndexX" was used by LibSVM to keep track of which values are different measures of the same variable. Comparing the target format with the raw data format revealed that the raw data had to be modified before it could be used with LibSVM.

In order to account for this, the raw data was manipulated such that it reflected the formatting required by LibSVM. Premature training trials demonstrated that formatting the data would not be sufficient if were reasonable classification rates were expected. Careful analysis of the data and the significance of the various input variables revealed

that the features with the greatest numerical various were limited to basic cartographic measures like altitude and distances to roads and water. The vast majority of the information, on the other hand, is a Boolean representation of soil- and wilderness- type, information that was more relevant for this specific data classification task. Since the LibSVM algorithm separated data into classes based on the numerical values present in the training samples, data scaling was necessary. After scaling, with all columns varying in fixed ranges, no single input feature was inadvertently granted more significance than it was due. Proper scaling of the data set, therefore, comprised the second preprocessing task.

File Size Considerations

Because of the scope of this project, large file sizes brought to the forefront two significant issues: limited available disk space and limited available computing resources. In light of said limitations, special care was taken to avoid exceeding disk quotas. The lack of dedicated computing power also necessitated file size limitations in order to ensure reasonable run-times for training and testing.

Keeping file sizes to a minimum was not a trivial task, however. Decompressed, the raw data was approximately 75 Megabytes (MB) in size. This was of particular concern since the raw data was sparsely formatted; any conversion or manipulation would significantly enlarge the file. Minute changes on a record by record basis had incredible effects, as they were repeated over 580,000 times, once for each data point. Essentially, for every two characters added to a record, the data file grew by more than 1 MB.

Formatting

As previously mentioned, formatting was essential. Additionally, the process of formatting the data was not one that could be streamlined to reduce file size, as the end result had to meet the specific requirements of LibSVM. The Java program (SVMPreProcess) converted the raw data into a format usable by LibSVM. See Appendix B for details of this program. A sample row of the formatted data is presented below.

5 1:2596 2:51 3:3 4:258 5:0 6:510 7:221 8:232 9:148 10:6279 11:1 12:0 13:0 ⋯

14:0 15:0 16:0 17:0 18:0 19:0 20:0 21:0 22:0 23:0 24:0 25:0 26:0 27:0 28:0 ⋯

29:0 30:0 31:0 32:0 33:0 34:0 35:0 36:0 37:0 38:0 39:0 40:0 41:0 42:0 43:1 ⋯

44:0 45:0 46:0 47:0 48:0 49:0 50:0 51:0 52:0 53:0 54:0

After formatting, the data set was approximately 165 MB in size.

Scaling

The formatting process presented a situation in which file growth was unavoidable. Scaling, on the other hand, offered many choices.

One incredibly straightforward option involved using Matlab and the function scale.m (see Appendix B) to import, scale, and then export the data. A small, 6 MB sample of the data set was selected for scaling using Matlab. This sample was scaled on a range of –5 to 5, column by column, and then exported in ASCII format. The resulting file was over 40 MB, growth by nearly a factor of eight! This was not surprising, however, when the ASCII export format that Matlab used was examined. Each value was recorded in scientific format with 8 decimal places and a three-character exponent. Considering the majority of the values in the data set were 1's and 0's, this representation

was deemed inefficient, to say the least. Applied to the entire formatted data set, the file would have been over 1 Gigabyte (GB) in size, clearly unacceptable. These results proved that data manipulation needed to occur in a way that took advantage of the fact that most of the data was represented as either a 0 or a 1, an approach that would achieve satisfactory scaling yet keep file size under control. This intention yielded the approach that was eventually applied to the entire data set.

LibSVM contained a useful program called svm-scale, which, scaled data that was already in the LibSVM format. The program scaled columns individually to a specified range. The program saved the data with six significant figures (or five decimal places), and since it used the LibSVM format, no pre- or post-processing of the data was required in order to maintain that formatting.

Since only the first 10 columns required such scaling (columns 2 through 11, since the first column is the class label), several programs were written to separate the data into two parts, modify them appropriately, and then merge them again. The Java program LeftRight separated the first 11 columns (class label and non-Boolean data) into one file and the remaining 44 Boolean-valued columns into another. The "left" file was then scaled with svm-scale, using the bounds –5 and 5. The "right" file was modified with the Java program Fives. Fives systematically replaced all 0's with –5's and all 1's with 5's. Next, Merge took both scaled files and combined them, creating a coherent data set. See Appendix B for the source code for Fives and Merge. A sample row of the final data set is presented below.

5 1:-1.31316 2:-3.58333 3:-4.54545 4:-3.15319 5:-2.76486 6:-4.28341 ⋯

7:3.70079 8:4.13386 9:0.826772 10:3.75366 11:5 12:-5 13:-5 14:-5 15:-5 ⋯

16:-5 17:-5 18:-5 19:-5 20:-5 21:-5 22:-5 23:-5 24:-5 25:-5 26:-5 27:-5 28:-5 ⋯

29:-5 30:-5 31:-5 32:-5 33:-5 34:-5 35:-5 36:-5 37:-5 38:-5 39:-5 40:-5 41:-5 ⋯

42:-5 43:5 44:-5 45:-5 46:-5 47:-5 48:-5 49:-5 50:-5 51:-5 52:-5 53:-5 54:-5

After scaling, the data set was approximately 215 MB in size. Compressed (using tar and gzip), this file was approximately 20 MB in size.

Batch Preparation

Although at this point the data was formatted and scaled appropriately, further steps were taken to ease the training and testing process. To better facilitate training on uniformly distributed but randomly selected data points, three more programs were written. These programs systematically gathered data samples from the complete data set. First, the program SeparateClasses separated the large data set into a single file for each of the seven classes. Second, the program RandomRead read a user-defined number of lines from random positions in the class files. Finally, the program Append concatenated the seven random samples into one training file for LibSVM to use. See Appendix B for the source code for these programs.

These programs, used in a script, simplified the automation of the training and testing process. The script first decompressed the data files if necessary. Next, it generated the training data and a small validation sample, generally of the same size as the training data. The validation data was used to get a general idea of the testing performance without waiting for the testing to complete, as testing often took 4-8 hours. Once validation had completed, testing in full was conducted. See Appendix B for a sample script.

Training, Testing and Results

The first trial was conducted prior to data scaling, as previously mentioned. Similar to the data set used by Jock A. Blackard (1998), the first 11,340 (2% of the sample space) samples were used for training, the next 3,780 (less than 1% of the sample space) for validation, and the validation data set in addition to the remaining 565,892 data points were used for testing. The training and validation samples both consisted of a uniform distribution of classes. Given the poor results of this trial, 15.03% classification for validation and 49.59% for testing, further trials were not conducted until the data was properly scaled and the other support programs were written.

Once the data was scaled and the previously mentioned script was prepared, training and testing was simply a matter of deciding what SVM configurations to try and determining how many trials of each configuration should be run. At this point, training and testing went through three phases.

The purpose of the first phase was to determine the expected variance between trials when the same support vector machine configuration was used. Using the svm-train defaults (radial basis kernel, cost parameter = 1; see Appendix C for other default values), eight trials were run. In all eight cases, 5810 (1% of the sample space) randomly sampled data points were used for training and additional 5810 randomly sampled data points were used for validation. To simplify the process, testing was performed on the whole data set instead of leaving out the samples used for training and validation . The results of these trials are presented below.

	Validation Rate	Testing Rate
Trial 1	72.89	60.28
Trial 2	72.03	61.57
Trial 3	72.18	61.55

Trial 4	73.27	61.52
Trial 5	73.33	61.99
Trial 6	72.08	61.31
Trial 7	72.99	61.12
Trial 8	73.08	60.70
Average	72.73	61.26
Standard Deviation	0.553	0.545

After determining that a high level of consistency existed between trials, the second phase of training and testing began. The purpose of this phase was to ascertain which configurations would best classify the data. Trials were run using the polynomial, radial basis and sigmoid kernels, and in all cases the cost parameter and training sample sizes were varied. In the case of the polynomial kernel, the degree was also varied. The configurations and results of these trials are presented below. In all cases the training sample size matched the validation sample size. For any parameters not mentioned, default values were used (see Appendix C).

Kernel	Cost Parameter	Other	Sample Size	Validation Rate	Testing Rate
Radial	10^9		5810	79.94	65.13
Radial	10^9		11620	76.12	63.93
Poly	1	Degree 3	5810	79.00	66.71
Poly	10^9	Degree 3	5810	DNC	–
Poly	1	Degree 5	5810	DNC	–
Poly	10^9	Degree 5	5810	DNC	–
Poly	1	Degree 7	5810	DNC	–
Poly	10^9	Degree 7	5810	DNC	–
Radial	*10^9*		*11620*	*82.42*	*69.26*
Sigmoid	1		5810	30.58	34.78
Sigmoid	1		11620	–	–
Sigmoid	10^9		5810	32.42	–
Sigmoid	10^9		11620	–	–
Poly	*1*	*Degree 3*	*11620*	*82.16*	*69.08*
Poly	10^9	Degree 3	2905	DNC	–
Poly	1	Degree 5	2905	66.83	55.86

Some training did not converge (DNC) in a reasonable period of time (three hours, in general). Dashes signify tests that were not completed because of a lack of convergence, poor validation result, or poor results in similar tests. The two tests in italics were chosen for additional trials in phase three.

In the third phase, additional trials were conducted in order to verify the classification rates of those configurations that performed well in the second phase. In the first series, a radial basis kernel was used, with a cost parameter of 10^9 and 11,620 samples in both the training and validation sets.

	Validation Rate	Testing Rate
Phase 2 Result	82.42	69.26
Trial 1	83.23	70.71
Trial 2	DNC	–
Trial 3	DNC	–
Trial 4	83.56	69.45
Average	83.07	69.81
Standard Deviation	0.587	0.788

In the second series, a polynomial kernel was used, with a cost parameter of one, degree of three and 11,620 samples in both the training and validation sets.

	Validation Rate	Testing Rate
Phase 2 Result	82.16	69.08
Trial 1	82.62	69.24
Trial 2	82.77	68.35
Trial 3	82.09	68.53
Trial 4	82.44	68.89
Average	82.42	68.82
Standard Deviation	0.291	0.372

No further trials were performed since these two series of trials verified the performance witnessed in phase two.

Conclusion

In the end, 70% classification was achieved, matching the results obtained by Blackard in his comparison of linear discriminant analysis and a back-propagation algorithm. This classification rate was obtained using a radial basis kernel, 11,620 randomly sampled and uniformly distributed data points, with a cost parameter of 10^9.

The sheer enormity of the data set did cause difficulties independent of the actual data. Care was taken with file storage and running times were monitored. It should not escape notice that greater classification rates may have been possible. It was unclear whether or not the trials that did not converge would improve upon the 70% classification rate that was achieved. However, as the training sample sets grew, the time needed to complete training and testing grew at an alarming rate. In addition, several configurations failed to converge within three or even six hours. In light of these facts, a classification rate of 70% was satisfactory.

References

Blackard, Jock A. 1998. *Comparison of Neural Networks and Discriminant Analysis in Predicting Forest Cover Types*: Ph.D. dissertation. Department of Forest Sciences, Colorado State University: Fort Collins, Colorado.

Haykin, Simon. 1999. *Neural Networks: A Comprehensive Foundation*. 2nd edition. Prentice-Hall: Upper Saddle River, New Jersey.

Appendix A – Data Set

The following is an excerpt (slightly modified) from covtype.info, the informational file that defines the structure of the Forest Cover Type data set.

This file, along with the data set itself, are available at http://kdd.ics.uci.edu/databases/covertype/covertype.html.

```
Variable Information

Name                                    Data Type         Measurement       Description

Elevation                               quantitative      meters            Elevation in meters
Aspect                                  quantitative      azimuth           Aspect in degrees azimuth
Slope                                   quantitative      degrees           Slope in degrees
Horizontal_Distance_To_Hydrology        quantitative      meters            Horz Dist to nearest surface water features
Vertical_Distance_To_Hydrology          quantitative      meters            Vert Dist to nearest surface water features
Horizontal_Distance_To_Roadways         quantitative      meters            Horz Dist to nearest roadway
Hillshade_9am                           quantitative      0 to 255 index    Hillshade index at 9am, summer solstice
Hillshade_Noon                          quantitative      0 to 255 index    Hillshade index at noon, summer soltice
Hillshade_3pm                           quantitative      0 to 255 index    Hillshade index at 3pm, summer solstice
Horizontal_Distance_To_Fire_Points      quantitative      meters            Horz Dist to nearest wildfire ignition points
Wilderness_Area (4 binary columns)      qualitative       0 or 1            Wilderness area designation
Soil_Type (40 binary columns)           qualitative       0 or 1            Soil Type designation (0=absence, 1=presence)
Cover_Type (7 types)                    integer           1 to 7            Forest Cover Type designation

Code Designations

Wilderness Areas:
    1 -- Rawah Wilderness Area
    2 -- Neota Wilderness Area
    3 -- Comanche Peak Wilderness Area
    4 -- Cache la Poudre Wilderness Area

Soil Types:
    1 to 40 : based on the USFS Ecological Landtype Units for this study area.

Forest Cover Types:
    1 -- Spruce/Fir
    2 -- Lodgepole Pine
    3 -- Ponderosa Pine
    4 -- Cottonwood/Willow
    5 -- Aspen
    6 -- Douglas-fir
    7 -- Krummholz
```

Appendix B – Source Code

Automation Script (data.sim)

```
echo Extracting files
echo

gzip -d data.tar.gz
tar -xf data.tar
rm data.tar -f

echo Gathering training samples
echo

java RandomRead covtype.scaled_1 data_1 830
java RandomRead covtype.scaled_2 data_2 830
java RandomRead covtype.scaled_3 data_3 830
java RandomRead covtype.scaled_4 data_4 830
java RandomRead covtype.scaled_5 data_5 830
java RandomRead covtype.scaled_6 data_6 830
java RandomRead covtype.scaled_7 data_7 830

echo
echo Consolidating training samples
echo

java Append data_1 data_2 tmp1
java Append tmp1 data_3 tmp2
java Append tmp2 data_4 tmp1
java Append tmp1 data_5 tmp2
java Append tmp2 data_6 tmp1
java Append tmp1 data_7 tmp2

mv tmp2 covtype.train

echo
echo Gathering validation samples
echo

java RandomRead covtype.scaled_1 data_1 830
java RandomRead covtype.scaled_2 data_2 830
java RandomRead covtype.scaled_3 data_3 830
java RandomRead covtype.scaled_4 data_4 830
java RandomRead covtype.scaled_5 data_5 830
java RandomRead covtype.scaled_6 data_6 830
java RandomRead covtype.scaled_7 data_7 830

echo
echo Consolidating validation samples
echo

java Append data_1 data_2 tmp1
java Append tmp1 data_3 tmp2
java Append tmp2 data_4 tmp1
```

```
java Append tmp1 data_5 tmp2
java Append tmp2 data_6 tmp1
java Append tmp1 data_7 tmp2

mv tmp2 covtype.validate

echo
echo Cleaning up
echo

rm -f data_?
rm tmp?

echo Performing training
echo

svm-train covtype.train covtype.model > train.log

echo Performing validation
echo

svm-predict covtype.validate covtype.model validate.out > validate.log

echo Performing testing
echo

svm-predict covtype.scaled covtype.model test.out > test.log

echo Done!
```

Append.java

```
import java.io.*;

class Append {
  // This program appends one file to the end of another.

  public static void main(String[] args) throws IOException {
    if (args.length < 3) {
      System.out.println("Usage: java Append <main> <new> <output>");
      System.exit(0);
    } // if

    BufferedReader fileMain = new BufferedReader(new
FileReader(args[0]));
    BufferedReader fileNew = new BufferedReader(new
FileReader(args[1]));
    BufferedWriter fileOut = new BufferedWriter(new
FileWriter(args[2]));

    System.out.print("Working..");
        int watch = 0;

    while (fileMain.ready()) {
      if (watch%25000==0)
```

```
                System.out.print(".");
      fileOut.write(fileMain.readLine()+'\n');
      watch++;
    }

    while (fileNew.ready()) {
      if (watch%25000==0)
                System.out.print(".");
      fileOut.write(fileNew.readLine()+'\n');
      watch++;
    }

    System.out.print("done!\n");

    fileMain.close();
    fileNew.close();
    fileOut.close();

  } // main

} // class
```

Fives.java

```
import java.io.*;
import java.util.*;

class Fives {
  // This program takes the boolean data and replaces all 1's with 5's
and all
  // 0's with -5's.

  public static void main(String[] args) throws IOException {
    if (args.length < 1) {
      System.out.println("Usage: java Fives <destination>");
      System.exit(0);
    } // if

    BufferedReader fileIn = new BufferedReader(new
FileReader("right.data"));
    BufferedWriter fileOut = new BufferedWriter(new
FileWriter(args[0]));

    int watch = 0;
    System.out.print("Working..");

    while(fileIn.ready()) {
      if (watch%25000==0)
                System.out.print(".");

      String lineIn = fileIn.readLine();
      String lineOut = "";

      // tokenize the string
      StringTokenizer valueTokens = new StringTokenizer(lineIn, " ");
```

```
      for (int i=0;i<44;i++) {
        String token = valueTokens.nextToken();
        String index = token.substring(0, 2);
        String value = token.substring(3,4);

        if (1 == Integer.parseInt(value))
          lineOut = lineOut + index + ":" + "5 ";
        else
          lineOut = lineOut + index + ":" + "-5 ";
      } // for

      fileOut.write(lineOut + '\n');
      watch++;
    } // while

    System.out.print("done!\n");

    fileIn.close();
    fileOut.close();
  }

} // Fives
```

LeftRight.java

```
import java.io.*;
import java.util.*;

class LeftRight {
  // This file splits the data set into two files, one (right) for the
boolean
  // data and another (left) for the numerical data.

  public static void main(String[] args) throws IOException {
    if (args.length < 1) {
      System.out.println("Usage: java LeftRight <input filename>");
      System.exit(0);
    } // if

    BufferedReader fileIn = new BufferedReader(new
FileReader(args[0]));
    BufferedWriter left = new BufferedWriter(new
FileWriter("left.data"));
    BufferedWriter right = new BufferedWriter(new
FileWriter("right.data"));

    int watch = 0;
    System.out.print("Working..");

    while(fileIn.ready()) {
      if (watch%25000==0)
                System.out.print(".");

      String lineIn = fileIn.readLine();
```

```
      String leftOut = "";
      String rightOut = "";

      // tokenize the string
      StringTokenizer valueTokens = new StringTokenizer(lineIn, " ");

      String token;
      for (int i = 0;i < 11;i++) {
        token = valueTokens.nextToken();
        leftOut = leftOut + token + " ";
      } // for

      while (valueTokens.hasMoreTokens()) {
        token = valueTokens.nextToken();
        rightOut = rightOut + token + " ";
      } // while

      left.write(leftOut + '\n');
      right.write(rightOut + '\n');
      watch++;
    } // while

    System.out.print("done!\n");

    fileIn.close();
    left.close();
    right.close();

  } // main

} // LeftRight
```

Merge.java

```
import java.io.*;

class Merge {
  // This program takes two files generated by LeftRight and merges
them.

  public static void main(String[] args) throws IOException {
   if (args.length < 3) {
      System.out.println("Usage: java Merge <left> <right>
<destination>");
      System.exit(0);
    } // if

    BufferedReader left = new BufferedReader(new FileReader(args[0]));
    BufferedReader right = new BufferedReader(new FileReader(args[1]));
    BufferedWriter fileOut = new BufferedWriter(new
FileWriter(args[2]));

    String leftIn = "";
    String rightIn = "";
```

```
    int watch = 0;
    System.out.print("Working..");

    while(left.ready() && right.ready()) {
      if (watch%25000==0)
                System.out.print(".");

      leftIn = left.readLine();
      rightIn = right.readLine();
      fileOut.write(leftIn + rightIn + "\n");

      watch++;
    }

    System.out.print("done!\n");

    left.close();
    right.close();
    fileOut.close();
  }

} // Merge
```

scale.m

```
function [x,xmin,xmax]=scale(x,low,high,type)
% Usage: [x,xmin,xmax]=scale(x,low,high,type)
%
% linearly scale x into the range of low to high
%  it must be that high -low > 0
% if type = 0 (default), low and high are scalars and scaling is
uniform in each dim
%         = 1, scaling is done at each individual dimension in the
range
%              vectors low and high
% output: x - scaled x,  xmax: original maximum, xmin: original min
% input: x - original x, low: desired min, high: desired max.
% copyright (c) 1996 by Yu hen Hu
% created 9/21/96
% modified 9/22/96: add xmin, xmax to output
% modified 4/5/2001: add type to allow scaling to indivisual ranges in
each dim.
%
if nargin==3,
   type=0;
end

[K,M]=size(x);
range=high-low; % type 0, scalar, type 1, may be a 1 X M vector
if type==0,
   xmax=max(max(x));xmin=min(min(x));xrange=xmax-xmin;
   x=(x-xmin)*range/xrange+low;
elseif type==1,
   if size(high)==size(low) == [1 1];
      high=ones(1,M)*high; low=ones(1,M)*low;
```

```
   end

   low=diag(diag(low))'; range=diag(diag(range))'; % make them 1 x M
vectors
   xmax=max(x); xmin=min(x); xrange=xmax-xmin; % each 1 x M vector
   mask=[xrange<1e-3]; % maskout elements too small, 1 X M
   range=(1-mask).*range+mask; xrange=(1-mask).*xrange+mask;
   x=(x-ones(K,1)*xmin)*diag(range./xrange)+ones(K,1)*low;
end
```

SeparateClasses.java

```
import java.io.*;

class SeparateClasses {
  // This program separates entries for different data classes into
different files
  // based on the assumption that the class indicator is the last digit
in a row
  // and is a single integer value.

  // Assumes 7 classes

  public static final int classes = 7;

  public static void main(String[] args) throws IOException {
  // verify correct number of arguments
  if (args.length < 1) {
    System.out.println("Usage: java SeparateClasses <filename>");
    System.exit(0);
  } // if

  // define file to read
  BufferedReader fileIn = new BufferedReader(new FileReader(args[0]));

  // define files to write
  int classes = 7;
  BufferedWriter[] fileOut = new BufferedWriter[classes];

  for (int i=1;i<=classes;i++) {
    fileOut[i-1] = new BufferedWriter(new FileWriter(args[0] + "_" +
i));
  } // for

  System.out.print("Working..");
  int watch = 0;

  while(fileIn.ready()) {
    if (watch%25000==0)
      System.out.print(".");

    String lineIn = fileIn.readLine();
    String classString = lineIn.substring(0, 1);
    int classNum = Integer.parseInt(classString);
    fileOut[classNum-1].write(lineIn+"\n");
```

```
    watch++;
  } // while

  System.out.print("done!\n");

  // clean up
  fileIn.close();

  for (int i=0;i<classes;i++) {
    fileOut[i].close();
  } // for

  } // main
} // SeparateClasses
```

SVMPreProcess.java

```
import java.io.*;
import java.util.*;

class SVMPreProcess {
  // Generates a new file with formatting specific to the LibSVM SVM
implementation.
  // Assumes 54 columns, comma delimited

  public static void main(String[] args) throws IOException {
    if (args.length < 2) {
      System.out.println("Usage: java SVMPreProcess <input filename>
<output filename>");
      System.exit(0);
    } // if

    BufferedReader fileIn = new BufferedReader(new
FileReader(args[0]));
    BufferedWriter fileOut = new BufferedWriter(new
FileWriter(args[1]));

    System.out.print("Working..");
        int watch = 0;

    while(fileIn.ready()) {
      // lines must be of the format:
      // <label> <index1>:<value1> <index2>:<value2> ...

      if (watch%25000==0)
                System.out.print(".");

      String lineIn = fileIn.readLine();
      String lineOut = "";

      int last = lineIn.lastIndexOf(",");

      lineOut = lineIn.substring(last + 1, lineIn.length()) + " ";
      lineIn = lineIn.substring(0, last);
```

```
      // tokenize the remaining string
      StringTokenizer valueTokens = new StringTokenizer(lineIn, ",");

      String token;
      int i = 1;
      while (valueTokens.hasMoreTokens()) {
        token = valueTokens.nextToken();
        lineOut = lineOut + i + ":" + token + " ";
        i++;
      }

      fileOut.write(lineOut + '\n');
      watch++;
    } // while

    System.out.print("done!\n");

    fileIn.close();
    fileOut.close();

  } // function main
}
```

RandomRead.java

```
import java.io.*;
import java.util.*;

class RandomRead {
  // This program reads a specified number of lines from random
positions within a
  // source files and writes them to a destination file.

  public static void main(String[] args) throws IOException {
    if (args.length < 3) {
      System.out.println("Usage: java RandomRead <source> <destination>
<count>");
      System.exit(0);
    } // if

    RandomAccessFile fileIn = new RandomAccessFile(args[0], "r");
    BufferedWriter fileOut = new BufferedWriter(new
FileWriter(args[1]));
    int count = Integer.parseInt(args[2]);

    int watch = 0;
    System.out.print("Working..");

    Random rand = new Random();
    fileIn.skipBytes(rand.nextInt((int) fileIn.length()));

    while(count>0) {
      int randCount = rand.nextInt((int) fileIn.length()/2);
```

```
      if (watch%25000==0)
                System.out.print(".");

      if (fileIn.skipBytes(randCount) < randCount) {
        fileIn.seek(0);
        fileIn.skipBytes(randCount);
      }

      boolean notnl = true;

      while(notnl) {
        int i = fileIn.read();
        if (i == '\n')
          notnl = false;
        if (i == -1) {
          fileIn.seek(0);
          notnl = false;
        }
      } // while

      String lineIn = fileIn.readLine();

      if (lineIn == null) {
        fileIn.seek(0);
      }
      else {
        fileOut.write(lineIn + '\n');
        count--;
      }

      watch++;
    } // while

    System.out.print("done!\n");

    fileIn.close();
    fileOut.close();
  } // main

} // RandomRead
```

ReadWrite.java

```
import java.io.*;

class ReadWrite {
  // This program reads a specified number of lines from a source file
and writes them
  // to a destination file.

  public static void main(String[] args) throws IOException {
    if (args.length < 4) {
      System.out.println("Usage: java ReadWrite <source> <destination>
<start line> <# of lines>");
      System.exit(0);
```

```
    } // if

    BufferedReader fileIn = new BufferedReader(new
FileReader(args[0]));
    BufferedWriter fileOut = new BufferedWriter(new
FileWriter(args[1]));

    System.out.print("Working..");
        int watch = 0;

    int count = 1;

    while (count < Integer.parseInt(args[2])) {
      if (watch%25000==0)
                System.out.print(".");
      fileIn.readLine();
      count++;
      watch++;
    }

    count = 0;

    while (count < Integer.parseInt(args[3]) && fileIn.ready()) {
      if (watch%25000==0)
                System.out.print(".");
      fileOut.write(fileIn.readLine()+'\n');
      count++;
      watch++;
    }

    System.out.print("done!\n");

    fileIn.close();
    fileOut.close();

  } // main

} // class
```

Appendix C – LibSVM

The following is an excerpt from the README file that accompanies the LibSVM program files. For more information regarding LibSVM and its capabilities, visit http://www.csie.ntu.edu.tw/~cjlin/libsvm/index.html.

```
`svm-train' Usage
=================

Usage: svm-train [options] training_set_file [model_file]
options:
-s svm_type : set type of SVM (default 0)
      0 -- C-SVC
      1 -- nu-SVC
      2 -- one-class SVM
      3 -- epsilon-SVR
      4 -- nu-SVR
-t kernel_type : set type of kernel function (default 2)
      0 -- linear: u'*v
      1 -- polynomial: (gamma*u'*v + coef0)^degree
      2 -- radial basis function: exp(-gamma*|u-v|^2)
      3 -- sigmoid: tanh(gamma*u'*v + coef0)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/k)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR
(default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR
(default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default
0.1)
-m cachesize : set cache memory size in MB (default 40)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking: whether to use the shrinking heuristics, 0 or 1 (default
1)
-wi weight: set the parameter C of class i to weight*C in C-SVC
(default 1)
-v n: n-fold cross validation mode

The k in the -g option means the number of attributes in the input
data.

option -v randomly splits the data into n parts and calculates cross
validation accuracy/mean squared error on them.
```