Machine Learning - Fall 2014
Final Project and Extra Credit
Report

Carlos Jaramillo
cjaramillo@gradcenter.cuny.edu

Juan Pablo Muñoz
jmunoz2@gradcenter.cuny.edu

December 2014

1 Introduction

This document describes our final project for Professor Haralick's Machine Learning class. We have been asked to implement a data generator and a classifier that learns a decision rule from a portion of the data set provided. In Section 2.1, we start by describing the model abiding the generation of instances for the data set we experiment upon in Section 2.2. Our supervised learning algorithm is then explained in Section 3, and it is then compared with other popular classifiers (Section 4).

Finally, in Section 5, we present our high-accuracy solution for learning the Extra Credit data set provided to us.

In order to disambiguate our work, we provide a short section on notation before all and provide the source code listings in the appendix Section 7.

1.1 Symbol Notation

In this subsection, we present the notation style used throughout the remaining of this manuscript:

- Vectors are notated in boldface upright lowercase. The elements of an n-vector are indexed from 0 to $n-1$.
- Matrices (or tables) are represented in boldface upright uppercase, such as for a table of rules $\mathbf{R}$.
- Sets are written in uppercase calligraphic font and its elements are listed within curly braces such as $\mathscr{S} = \{s_0, \ldots, s_N\}$.
- We can address to a particular element of a vector or matrix using its position index. We interchangeably use both square brackets [ ] and post-subscript styles. For example,$\mathbf{v}[i]$ or v_i refer to the element found at the $(i+1)^{\text{th}}$ position in $\mathbf{v}$.

- Double vertical bars ("||") on each side of a vector expression are used to denote its magnitude: the Euclidean norm for a vector or the number of elements in a set.
- Function names use upright lowercase letters and additional subscript names based on context. To distinguish vector-valued functions from scalar-valued functions, we use boldface, such as for the $\mathbf{scale}(\mathbf{v}, s)$ function which outputs the scaled vector $\mathbf{v}$ by some value s.

1.2 Requirements and Assumptions

The following problem requirements and assumptions are important for the successful engineering and discovery of rules from our data set.

(a) We must design a labeled two-class (binary) data set $\mathbf{D}$ of $Z = 100,000$ instance vectors of $N = 10$ dimensions. Therefore, $\|\mathbf{D}\| = Z$, and $\mathbf{c}$ is the array of class labels:

$$\mathbf{c} = [0, \quad 1], \text{ or more generally } \mathbf{c} = [c_0, \quad c_1] \tag{1.1}$$

for class labels c_0 and c_1. In our implementation, c_0='A' and c_1='B'.

(b) The component values of the 10-dimensional vector (instance vector) are limited to the set $\mathscr{D} = \{0, \ldots, 9\}$, the digits of the decimal numbering system. Hence, any instance vector $\mathbf{x} \in \mathscr{D}^{10}$.

(c) Given a labeled data set $\mathbf{D}$ of $Z = 100,000$ instances, it must be split evenly between c_0 and c_1. Here, we have 50,000 instances of c_0 and 50,000 of c_1.

(d) Out of the entire data set $\mathbf{D}$, we must allocate 50% as $\mathbf{D}_L$ for learning (Section 2.1) and 50% as $\mathbf{D}_T$ for the final testing phase (Section 3.2).

(e) The test data set $\mathbf{D}_T$ must produce a classification accuracy of less than 60% on the required classifiers (Section 4), but above 90% for our classification method.

(f) In order to reduce the J number of decision rules (Section 2.1) that govern the data set generation, we keep the K number of relevant components to a minimum as long as (e) is satisfied.

(g) During learning, we make no assumption on the notion of $\mathbf{k}$,the vector of relevant component indices, since they get discovered in the training phase of our algorithm (Section 3.1).

2 Building the Data Set

In this section, we begin by describing the model that is consequently applied for the data set generation in Section 2.2.

2.1 Data Generation Model

In order to reduce the J total number of decision rules necessary for the data set generation (Section 2.2), we pair $\mathscr{D}_L = \{0,...,4\}$, which are the five lower values of $\mathscr{D}$, with the remaining values $\mathscr{D}_H = \{5,...,9\}$ in a well-established fashion. Nominally, we establish the following grouping vector of 2-tuples (pairs):

$$\mathbf{g} = \big[(0,5),(1,6),(2,7),(3,8),(4,9)\big] \tag{2.1}$$

The grouping $\mathbf{g}$ is maintained along all K relevant components indices represented by $\mathbf{k}$, which are chosen arbitrarily, as well. Empirically, we find that $K \geq 3$ satisfies requirement (f) during the testing phase (Section 3.2).

Thus, we create a table $\mathbf{R}$ of $J = \|\mathbf{g}\|^K = 5^K$ vector rules $(\mathbf{r}_i, y_i)$, where $\mathbf{r}_i = [r_{i,0}, \quad r_{i,1}, \quad \ldots, \quad r_{i,K-1}]$ is a K-vector instance of indexes of $\mathbf{g}$ corresponding to each relevant feature. Note, that we actually generalize the choice of components in this explanation, but it may appear as if they are K consecutively relevant features and this is not necessarily the case.

Initially, $\mathbf{R}$ is populated alone from the decomposed enumeration of its J rules, which encode the index position of pairs in $\mathbf{g}$, such that

$$\begin{aligned}
\mathbf{r}_0 &= \Big[0, \quad 0, \quad \ldots, \quad 0\Big] \\
\mathbf{r}_1 &= \Big[0, \quad 0, \quad \ldots, \quad 1\Big] \\
&\vdots \\
\mathbf{r}_{J-2} &= \Big[4, \quad \ldots, \quad 4, \quad 3\Big] \\
\mathbf{r}_{J-1} &= \Big[4, \quad 4, \quad \ldots, \quad 4\Big]
\end{aligned}$$

With $K = 3$, we get $J = 5^3 = 125$ distinct rules $\mathbf{r}_i$ where, $i \in \{0,\ldots,J-1\}$. For example, some $\mathbf{r} = [3, \quad 0, \quad 1]$ refers to instances with values in their corresponding pairs as $[v_0 \in \mathbf{g}[3], \quad v_1 \in \mathbf{g}[0], \quad v_2 \in \mathbf{g}[1]]$. In Section 2.2, we describe how we expand each pair in $\mathbf{g}$ to generate the instances of our data set.

Eventually, we fill up a consistent table of rules by randomly assigning classes $y_i \in \mathbf{c}$ to each $\mathbf{r}_i$ vector rule as follows:

$$\mathbf{R} := \big\{(\mathbf{r}_i, y_i) \mid y_i \leftarrow \text{randint}(c_0, c_1), \forall i \in \{0,\ldots,J-1\}\big\} \tag{2.2}$$

where randint$(low, high)$ is a pseudo random integer generator for values from a given range $[low, high]$ inclusive.

> Note: Upon creation of the three elements of this model: $\mathbf{R}$, $\mathbf{g}$ and $\mathbf{k}$, we freeze their state as an indicative that an actual classification model exists for the data set $\mathbf{D}$ that will be generated next (in Section 2.2).

Knowing the decision table $\mathbf{R}$, any component instance $\mathbf{x}_z$ in the data set $\mathbf{D}$ can be classified as y_z by first resolving its rule index i from its K relevant component values,

$\mathbf{p}_z$. Here, $\mathbf{m}_z \leftarrow \textbf{reduce}(\mathbf{p}_z)$ via a reduction function that maps the values in $\mathbf{p}_z \in \mathscr{D}^K$ onto its lowest pair $\mathbf{m}_z \in \mathscr{D}_L^K$, such that $\textbf{reduce} := \mathscr{D}^K \mapsto \mathscr{D}_L^K$, defined as follows

$$\mathbf{m} \leftarrow \textbf{reduce}(\mathbf{p}) := [p \bmod \|\mathbf{g}\|,\ \forall p \in \mathbf{p}] \tag{2.3}$$

where mod is the modulus operator and $\|\mathbf{g}\| = 5$ in our case.

With $\mathbf{m}_z$, we can resolve the rule address i for $\mathbf{x}_z$ via function address() defined as

$$i \leftarrow \text{address}(\mathbf{g}, \mathbf{x}_z) := \textbf{reduce}(\mathbf{x}_z) \cdot \mathbf{b} \tag{2.4}$$

the dot product between the reduced vector $\mathbf{m}_z$ and $\mathbf{b} = [\|\mathbf{g}\|^0, \quad \|\mathbf{g}\|^1, \quad \ldots, \quad \|\mathbf{g}\|^{K-1}]$ as the constant multiple factor vector.

2.2 Data Set Generation

Out of the set of rules $\mathbf{R}$ engineered in Section 2.1, we have to generate an initial data set $\mathbf{D}$ of $Z = 100,000$ instances according to the requirements described in Section 1.2. Recall, we must provide exactly 50,000 instances pertaining to class c_0 and 50,000 to class c_1. Thus, we start by systematically separating $\mathbf{R}$ into subset $\mathbf{R}_0$ containing all the rules $\mathbf{r}$ belonging to $y = c_0$, and subset $\mathbf{R}_1$ with rules belonging to $y = c_1$, such that

$$\mathbf{R}_0 := \big\{(\mathbf{r}, y) \mid y = c_0, \forall \mathbf{r} \in \mathbf{R}\big\} \tag{2.5}$$

$$\mathbf{R}_1 := \big\{(\mathbf{r}, y) \mid y = c_1, \forall \mathbf{r} \in \mathbf{R}\big\} \tag{2.6}$$

and it holds that

$$\mathbf{R} \leftarrow \mathbf{R}_0 \bigcup \mathbf{R}_1$$

$$\emptyset \leftarrow \mathbf{R}_0 \bigcap \mathbf{R}_1$$

Consequentially, it is trivial to generate $\mathbf{D}_0$ with 50,000 random instances out of $\mathbf{R}_0$. Similarly, $\mathbf{D}_1$ is formed by 50,000 instances out of $\mathbf{R}_1$ to produce

$$\mathbf{D} \leftarrow \{\mathbf{D}_0, \mathbf{D}_1\} \tag{2.7}$$

We proceed to expand the values on the relevant components whose indices are given in $\mathbf{k}$. In equation (2.1), we set up $\mathbf{g}$ with non-overlapping value pairs. We take advantage of this grouping pattern in order to expand the initial value assignments on the K relevant components of each instance of $\mathbf{D}$, as follows:

$$\mathbf{D}[z, j] \leftarrow \mathbf{D}[z, j] + \|\mathbf{g}\| * \text{randint}(0, 1), \forall z \in \{0, \ldots, Z-1\}, \forall j \in \{0, \ldots, J-1\} \tag{2.8}$$

where $\|\mathbf{g}\| = 5$ in our case.

Finally, we fill up the values of the irrelevant components of all Z instance vectors by using a random value assignment from $\mathscr{D}$, and we proceed to shuffle the entire data set of Z examples.

3 Learning our Classification Model

The task of learning the most accurate classification model uses the specified portion, $\mathbf{D}_L$, from the data set according to requirement (d). The learner assumes no knowledge of the actual K number of relevant components nor their identities. We only know that the possible decision rules $\tilde{\mathbf{R}}$ obey a pair-wise grouping indicated by $\tilde{\mathbf{g}}$ as a vector of tuples (without membership redundancy) and some vector $\tilde{\mathbf{k}}$ for the indices of the relevant components.

Our learner's goal is to discover these three crucial elements of the classification model described in Section 2.1 during the training phase. The method for evaluating the accuracy of correct classification is discussed in Section 3.2.

3.1 Training Phase

Overall, the best hypothesized classification model aims to maximize its classification accuracy when predicted on $\mathbf{D}_{Lv}$, a 50% of $\mathbf{D}_L$ used for validation within the training phase. Hence, we split $\mathbf{D}_L$ such that $\mathbf{D}_L \leftarrow \mathbf{D}_{Lt} \bigcup \mathbf{D}_{Lv}$, where $\mathbf{D}_{Lt}$ is the other 50% of the data set that we use for training.

We attempt the learning via brute-force discovery of relevant components candidates referred by their indices list $\tilde{\mathbf{k}}$. Simultaneously, we discover the candidate pairings $\tilde{\mathbf{g}}$. The hypothesized decision rules $\tilde{\mathbf{R}}$ are generated and used to estimate the accuracy of classification $\tilde{a}$ for the current hypothesis model (as explained in Section 3.2).

First, we construct the matrix $\mathbf{G}$ holding all the valid vectors of pairs out of their possible combinations. In this case, there are exactly 945 candidates for $\tilde{\mathbf{g}}$. Notice that we have not assumed the structure of the original pairing in order to generalize the learning for all possible grouping patterns.

The `learn()` procedure listed on the next page explains in detail how our learning algorithm works. It basically consists of an exhaustive search among grouping candidates from $\mathbf{G}$ and among all candidates in $\mathbf{K}_K$, which is the set of all possible combinations of K number of components out of the N-dimensions. Both combinations are searched in order to find the best vectors $\tilde{\mathbf{g}}$ and $\tilde{\mathbf{k}}$ that predict classes with a high classification accuracy (see Section 3.2 for the definition of `estimateAccuracy`). In other words, for each hypothesized classification model constructed from the training data $\mathbf{D}_{Lt}$, we seek the one which maximizes the classification accuracy $\tilde{a}$ as a result of its application on the validation training set $\mathbf{D}_{Lv}$. In fact, the learning stops once a_K has began to decrease after a last iteration with trials of $K+1$ components.

3.2 Accuracy of Correct Classification (The Test)

The following estimation of classification accuracy is used both during the training's validation phase and on the final test data set, $\mathbf{D}_T$, on which the learned classification rules are applied.

First, a predictor procedure takes an instance vector $\mathbf{x}_\mathbf{k}$ that has been stripped out of its irrelevant components, resulting into $\mathbf{p}_\mathbf{k}$ as explained in Section 2.1. Accompanied by the list of pairs $\mathbf{g}$ and the decision rules $\mathbf{R}$ learned for the classification model,

Procedure learn($\mathbf{D}_{Lt}$, $\mathbf{D}_{Lv}$, $\mathbf{G}$)

Input: $\mathbf{D}_{Lt}$: training data set (measurements $\mathbf{x}$ and corresponding classes y)
Input: $\mathbf{D}_{Lv}$: validation data set (measurements $\mathbf{x}$ and corresponding classes y)
Input: $\mathbf{G}$: list of possible grouping vectors
Output: $\tilde{\mathbf{k}}$: Learned vector of relevant component indices
Output: $\tilde{\mathbf{g}}$: Learned vector of value pairs
Output: $\tilde{\mathbf{R}}$: Table of decision rules for the learned classification model

```
// Parameter initialization
```
$\tilde{a} \leftarrow -1$. `// Best accuracy value`
$a_K \leftarrow 0$. `// Initialize accuracy due to` K `components`
$\mathbf{k}_K \leftarrow [0,1,2,3,4,5,6,7,8,9]$ `// Every component is assumed relevant`
$\mathbf{g}_K \leftarrow$ None `// No assumption before learning`

`// Try` K `number of relevant components up to` N `dimensions`
for $K \in \{1,\ldots,N\}$ **do**
  if $\tilde{a} < a_K$ **then**
    `// Set best elements of learned classification model`
    $\tilde{\mathbf{k}} \leftarrow \mathbf{k}_K$
    $\tilde{\mathbf{g}} \leftarrow \mathbf{g}_K$
    $\tilde{a} \leftarrow a_K$
    $a_K \leftarrow 0$. `// Reset accuracy due to` K `components`
    `// Generate combinations of size` K `out of` N `components`
    $\mathbf{K}_K \leftarrow$ **componentsCombinations**(K)
    foreach $\mathbf{g}_K \in \mathbf{G}$ **do**
      foreach $\mathbf{k}_K \in \mathbf{K}_K$ **do**
        $\mathbf{R}_K \leftarrow$ **generateRulesLUT**$(\mathbf{D}_{Lt}, \mathbf{g}_K, \mathbf{k}_K)$
        `// Validate accuracy due to hypothesized rules`
        $a_h \leftarrow$ estimateAccuracy$(\mathbf{g}_K, \mathbf{k}_K, \mathbf{R}_K, \mathbf{D}_{Lv})$
        if $a_K < a_h$ **then**
          $a_K \leftarrow a_h$
  else
    break
    `// When accuracy starts to decrease, it's time to stop because we have found the plateau concerning the number of relevant components`
$\tilde{\mathbf{R}} \leftarrow$ **generateRulesLUT**$(\mathbf{D}_{Lt}, \tilde{\mathbf{g}}, \tilde{\mathbf{k}})$
return $\tilde{\mathbf{k}}$, $\tilde{\mathbf{g}}$, $\tilde{\mathbf{R}}$

a predicted class $\mathbf{y}'$ is returned. `predictClass()` procedure below describes the prediction formally.

Thus, the `estimateAccuracy()` procedure computes the relative probability of correctly classified instance vectors $(\mathbf{x}_K, y)$ predicted with `predictClass()`.

The final test using the learned elements for classification, such that:

$$\tilde{\mathbf{R}}, \tilde{\mathbf{k}}, \tilde{\mathbf{g}} \leftarrow \text{learn}(\mathbf{D}_{Lt}, \mathbf{D}_{Lv}, \mathbf{G})$$
$$a_{final} \leftarrow \text{estimateAccuracy}(\tilde{\mathbf{g}}, \tilde{\mathbf{k}}, \tilde{\mathbf{R}}, \mathbf{D}_{\mathbf{T}})$$

classifies with near-perfect accuracy of 100% due to the brute-force approach taken for the discovery (learning) of this classification model.

Procedure predictClass(**g**, **R**, **x**)

Input: **g**: vector of value pairs
Input: **R**: decision rules for classification model
Input: **x**: instance vector (only measurement)
Output: y': predicted class

$i \leftarrow \text{address}(\mathbf{g}, \mathbf{x})$ `// Address resolution defined in equation (2.4)`
`// Recall that address() calls reduce() on x.`
$y' \leftarrow \mathbf{R}[i]$ `// A direct table look-up for the class of a rule`
return y'

Procedure estimateAccuracy(**g**, **k**, **R**, **D**)

Input: **g**: vector of value pairs
Input: **k**: vector of relevant component indices
Input: **R**: decision rules for classification model
Input: **D**: data set to be evaluated (measurements **x** and corresponding classes y)
Output: a: The accuracy of correctly predicted classes

$\mathbf{D}_K \leftarrow [\mathbf{D}[k],\ \forall k \in \mathbf{k}]$ `// Data subset out of` K `relevant components`
`// Initialize counters:`
$correct \leftarrow 0$
$total \leftarrow 0$
`// Run the prediction on all instances from the data subset:`
for $(\mathbf{p}_K, y) \in \mathbf{D}_K$ **do**
    $y' \leftarrow \text{predictClass}(\mathbf{g}, \mathbf{R}, \mathbf{p}_K)$
    $total \leftarrow total + 1$ `// Accumulate total count`
    if $y' = y$ **then**
        $correct = correct + 1$ `// count correct predictions`
$a \leftarrow correct/total$
return a

---

4 Comparison to Some Popular Classification Methods

As required by the assignment, we ran our engineered data set on the classifiers listed in the instructions for the final project, but also in some additional classifiers. None of the other classifiers were able to surpass the 60% accuracy requirement. Our method classified the data with 99% accuracy. For this exercise, we empirically noticed that $K = 3$ relevant components is enough to confuse all other machine learning methods listed in Table 1. This table compares both the relative time performance (on an 2.4 GHz Intel Core i5 CPU) and the classification accuracy for several popular classifiers available for exploration in *Weka* and *scikit-learn*. We employ the same 50% split for the learning and testing sets.

Table 1: Comparison of the performance of popular classifiers in our data set

Classification Method	Model Build Time [s]	Accuracy [%]
Our method	2242.5	99.9
Required methods to compare against		
Discrete Naive Bayes	1.0	52.6
Naive Bayes Gaussian Classifier	0.03	52.7
Logistic Regression	0.5	53.0
Support Vector Machines *Kernels*: linear, polynomial, rbf*, sigmoid	480.0 *	54.2 *
Multivariate Gaussian Classifiers *All Covariances*: spherical, diag, tied, full	0.3	52.0
Fisher Linear Discriminant Analysis	0.08	52.6
Two Layer Neural Network Classifier (Multilayer Perceptron Back Propagation)	46.2	49.9
Other popular methods		
LibSVM	6344.6	53.8
Multipass Learning Vector Quantization	0.3	52.5
Multilayer Perceptron	3.6	53.3
Real Adaboost	0.5	59.4

5 Extra Credit

In addition to completing our final project, we pursued an efficient solution (time and memory wise) for the extra credit assignment. The extra credit problem achieves above a 94% accuracy on a test set using quantization and discrete Bayes decision rule. The data set has 100,000 instances of four dimensions with decimal point values between 0.0 and 1.0 (inclusive). We assume that all the components of the instance vector are relevant, in contrast to what is done for the learning method of the final project, where we have to find the possible relevant components.

Table 2: One solution to the Extra Credit assignment using 6 intervals per component

Component	Boundaries
0	-inf 0.1762663413247462 0.368299962446487 0.5967535523440505 0.6480591180242109 0.7718240785632018 +inf
1	-inf 0.14311962768044778 0.38061905886313 0.5815324392905542 0.755037835931495 0.8744193888296643 +inf
2	-inf 0.11572512198666729 0.2623241751950748 0.5533195893686684 0.8216255357340024 0.9187389784063063 +inf
3	-inf 0.10711488966131297 0.36197351265946925 0.6014295394003168 0.7984508850612501 0.92662890179109 +inf

We wrote two programs, one in Java and one in Python. The listing for the Java version is included in the Appendix section. Each program solves the extra credit assignment using slightly different approaches. However, both approaches are based on the Quantization slides from Prof. Haralick's Machine Learning course at the CUNY Graduate Center.

It is important to mention that due to the randomness present in the algorithm, our current implementation does not always reaches 94% accuracy in the relatively short time (about 30 minutes) allocated to the model building in the learning phase.

In the following sections, we explain how our implementation finds solutions to the problem, but first, we present one solution to the problem in Table 2. The solution given in Table 2 uses only 6 intervals for each of the 4 components. It yielded approximately 94.5% accuracy on the validation data set and 94.6% on the test data set.

In the following sections, we describe each step of execution of our program. It is important to clarify that since we are using random processes, each trial results in different values. For this experiment, all the trials were processed on a 2.6 GHz Intel Core 2 Duo CPU. In each of the following sections, we also include the output for one arbitrary successful trial.

5.1 Randomizing Data

First, we randomize the given data and divide it in three disjoint data subsets. As advised by Prof. Haralick, 30% is used for training, 30% for learning validation and optimization, and the remaining 40% for the final testing. Each trial uses random selected instances for each of the data sets.

Output of our program in an arbitrary run:
Number of instances for training: 29732
Number of instances for validation: 30330
test size: 39938

5.2 Learning the Number of Bins

We used a brute force approach to find a good number of quantized intervals in each component. After have received advice from Prof. Haralick, we set our programs to try different permutations using 3 to 6 intervals for each component and compute an accuracy value for each permutation. This decision greatly reduced the amount of computation required. Another decision that we made, after observing a few trials, was to make our program keep only the permutations that had achieved more than 71% accuracy.

Output of our program in an arbitrary run:
Brute force optimization to find values of M using a range from 3 to 6 intervals in each component

Candidate Configuration of Number of Intervals per Component	**Accuracy** $[\%]$
$[5,\ 5,\ 3,\ 6]$	71.08256880733945
$[5,\ 5,\ 4,\ 6]$	71.42618849040867
$[5,\ 5,\ 5,\ 3]$	71.0558798999166
$[5,\ 5,\ 5,\ 6]$	72.37364470391994
$[5,\ 5,\ 6,\ 3]$	72.08673894912427
$[5,\ 5,\ 6,\ 5]$	71.75646371976647
$[5,\ 5,\ 6,\ 6]$	73.1442869057548
$[5,\ 6,\ 5,\ 6]$	71.1092577147623
$[5,\ 6,\ 6,\ 6]$	71.74979149291076
$[6,\ 5,\ 4,\ 6]$	71.04920767306089
$[6,\ 5,\ 5,\ 6]$	71.62969140950792
$[6,\ 5,\ 6,\ 3]$	71.00583819849875
$[6,\ 5,\ 6,\ 5]$	71.02251876563803
$[6,\ 5,\ 6,\ 6]$	72.55045871559633
$[6,\ 6,\ 6,\ 6]$	71.06588824020017

5.3 Boundary Optimization

After we gather a list of candidates (i.e., configuration of components with different number of intervals), we run a brute force optimization in order to find a better position of the boundaries in each component. This procedure works as follows:

1. A component and a boundary of that component are chosen at random.
2. A first method moves the boundary using larger steps and continue to move the boundary in the same direction as long as the accuracy continues to increase.
3. After reaching 89% accuracy, a second method is used to move the boundaries. This time in very small steps. If a movement of a boundary results in better accuracy, the program keeps that new position for that boundary. Otherwise, the previous boundary is preserved.

We ran the boundary optimization routine for 30 minutes on each candidate configuration.

These are the results for an arbitrary trial:

Output of our program in an arbitrary run:

Candidate Configuration of Number of Intervals per Component	**Accuracy Accuracy** $[\%]$ at 30 minutes running
$[5, \quad 5, \quad 3, \quad 6]$	81.36556208742487
$[5, \quad 5, \quad 4, \quad 6]$	85.59466221851543
$[5, \quad 5, \quad 5, \quad 3]$	80.35362802335280
$[5, \quad 5, \quad 5, \quad 6]$	87.56630525437865
$[5, \quad 5, \quad 6, \quad 3]$	80.34695579649708
$[5, \quad 5, \quad 6, \quad 5]$	88.22351959966639
$[5, \quad 5, \quad 6, \quad 6]$	90.09841534612176
$[5, \quad 6, \quad 5, \quad 6]$	89.52460383653045
$[5, \quad 6, \quad 6, \quad 6]$	89.59132610508758
$[6, \quad 5, \quad 4, \quad 6]$	84.64317383165293
$[6, \quad 5, \quad 5, \quad 6]$	90.18181818181819
$[6, \quad 5, \quad 6, \quad 3]$	80.32360300250209
$[6, \quad 5, \quad 6, \quad 5]$	90.25187656380317
$[6, \quad 5, \quad 6, \quad 6]$	90.24854045037531
$[6, \quad 6, \quad 6, \quad 6]$	94.50832274884685

5.4 Class Prediction on the Test Data Set

As mentioned previously, we put aside ≈40% of the original data set for testing. After our programs have performed the optimization of the number of intervals and the location of the boundaries, we run the best configuration on the test data.

Output of our program in an arbitrary run:
Final prediction on test data
Component Configuration: $[6, \quad 6, \quad 6, \quad 6]$
Accuracy: 94.59601521826192 %

As shown here, our program performs a successful trial of +94% accuracy on the extra credit data set in approximately 30 minutes (or less).

6 Conclusion

We have fulfilled the requirements for the final project and the extra credit assignment. Our final project implementation is capable to generate such data sets where their high classification accuracy is satisfied. Similarly, for the data set provided as the extra credit assignment, we learn the quantization boundaries that achieve the desired accuracy.

As any complex implementation, our programs can always be improved. In this project(s), we haven't exploited efficiency (in terms of running time) for the afore-

mentioned algorithms. We welcome the opportunity, if presented, to demonstrate the execution of our programs, and/or assist anybody interested in running our code.

We thank Professor Haralick for his sincere and enlightening advice during the early stages of this project. We thank him for such an interesting semester.

7 Appendix: Implementation Source Code

7.1 Project Program

data_generation.py

```
from __future__ import division
from __future__ import print_function

import numpy as np
import datetime
import common_tools

def generate_rules(class_labels, num_of_components, relevant_components,
    num_of_groups):
    '''
    Generate learning_driver rules as a big LUT where irrelevant values
        get an invalid value of -1
    Valid values for components are only nonnegative integers in the
        interval [0, 9].
    Here, each possible combination of values among the relevant
        components becomes a rule.
    Thus, a table of decision rules is generated using random
        learning_driver.
    The length of the decision table depends on the number of groups on
        the components and the total number of relevant components.
    For example, for groups of 2 (pairs) and only 3 relevant components,
        there exists (10/2)^3 = 5^3 = 125 decision rules.

    @param class_labels: list of nonnegative integers for allowed
        classes (categories)
    @param num_of_components: Total number of components (a.k.a features)
    @param relevant_components: A list of those relevant components in
        the rules and any data set generated employing the table of
        decision rules created here.
    @param num_of_groups: If set to 10, then grouping does not have an
        effect since every value is a singleton.

    @return: The numpy array of rules where the last column corresponds
        to learning_driver labels.
    '''
    # Create a LUT of num_of_components by n_rules
    n_rules = num_of_groups ** len(relevant_components)
```

```
    components_LUT = np.zeros((n_rules, num_of_components),
        dtype='int8') - 1

    # Fill up the relevant LUT of components with indices on the
        respective places
    indices = np.arange(n_rules)

    multiple = n_rules
    # Break up the indices in to its digits
    for comp in relevant_components:
        components_LUT[:, comp] = (indices % multiple) // int(multiple /
            num_of_groups)
        multiple = multiple / num_of_groups

    num_of_classes = len(class_labels)

    # Create an array of labels of total_num_of_instances length
    classes = np.asarray(class_labels, dtype='int8')
    labels = classes[np.random.randint(num_of_classes, size=(n_rules))]

    # Shuffle labels
    np.random.shuffle(labels)

    # Initialize all values in dataset to -1
    complete_LUT = np.zeros((n_rules, num_of_components + 1),
        dtype='int8') - 1
    complete_LUT[:, :num_of_components] = components_LUT
    complete_LUT[:, -1] = labels

    return complete_LUT

def generate_dataset(rules, relevant_components, total_num_of_instances,
    instances_percentages, num_of_groups):
    '''
    Irrelevant components are filled up with random values from a
        uniform distribution
    Last, the generated instance vectors (measurement records or rows
        including their associated class label) of the data set are
        shuffled and returned as the final result.

    @param rules: Numpy ndarray of rules organized as a 2D table of
        components and classes (last column)
    @param relevant_components: The list of relevant components
    @param total_num_of_instances: The desired number of instances
    @param instances_percentages: The split (percentage-wise) of
        instances from each category (class)
    @param num_of_groups: If set to 10, then grouping does not have an
        effect because groups would have a single element each.

    @return: The generated data set as a numpy array in which the last
```

```
        column corresponds to learning_driver labels.
    '''
    low = 0 # Lowest valid measurement value
    high = 9 # Highest valid measurement value

    num_of_components = rules.shape[1] - 1
    dataset = np.ndarray((total_num_of_instances, num_of_components +
        1), dtype='int8')

    rules_c0 = rules[np.where(rules[..., -1] == 0)]
    rules_indices_c0 = np.random.randint(len(rules_c0),
        size=(instances_percentages[0] * total_num_of_instances / 100))
    dataset[:len(rules_indices_c0)] = rules_c0[rules_indices_c0]

    rules_c1 = rules[np.where(rules[..., -1] == 1)]
    rules_indices_c1 = np.random.randint(len(rules_c1),
        size=(instances_percentages[1] * total_num_of_instances / 100))
    dataset[len(rules_indices_c0):] = rules_c1[rules_indices_c1]

    if num_of_groups < 10:
        for f in relevant_components:
            # Butterfly approach:
            for q in range(num_of_groups): # Random reassignment done in
                reverse order to avoid overwrites
                target_idx = np.where(dataset[:, f] == q)[0]
                dataset[target_idx, f] = dataset[target_idx, f] +
                    num_of_groups * np.random.randint(2,
                    size=len(target_idx))

    # Fill random values on irrelevant components
    for f in range(num_of_components):
        if f not in relevant_components:
            # Draw samples from a uniform distribution.
            irrelevant_comp_data = np.random.uniform(low, high,
                size=total_num_of_instances)
            dataset[:, f] = irrelevant_comp_data

    # Shuffle rows
    np.random.shuffle(dataset)

    return dataset


if __name__ == '__main__':
    class_labels = [0, 1]
    instances_percentages = [50, 50] # Must add to 100%
    relevant_components = [0, 1, 8] # Arbitrary component numbers
    total_num_of_instances = 100000
    num_of_components = 10
    dataset_header = ["d" + str(fname + 1) for fname in
```

```
    range(num_of_components)]
dataset_header = ', '.join(dataset_header) + ', class'

path_to_output_files = "./InputData/"

# vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
will_generate_rules = True # <<<<< Important to SET
quant_levels = 5 # If set to 10, then quantization does not have an
    effect.
save_class_as_string = True # <<<<< Determines whether to save
    classes as Strings!
k_comp_str = str(len(relevant_components))
filename_time = k_comp_str +
    "rel_components-quantized_butterfly-2014-12-27"
# ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

if save_class_as_string:
    custom_fmt = ["%s" for fname in range(num_of_components)] +
        ["%s"]

if will_generate_rules:
    rules = generate_rules(class_labels, num_of_components,
        relevant_components, quant_levels)
    # Use current time for filename suffix
    now = datetime.datetime.now()
    # filename_time = now.strftime("%Y-%m-%d-%H-%M-%S")
    rules_filename = path_to_output_files + "rules-" + filename_time
        + ".csv"

    if save_class_as_string:
        rules_labels = np.where(rules[:, -1] == 0, 'A', 'B') # Save
            table of rules to file
        rules_as_str = np.hstack((rules[:, :-1].astype('S2'),
            rules_labels.reshape(-1, 1)))
        np.savetxt(rules_filename, rules_as_str, delimiter=",",
            header=dataset_header, fmt=custom_fmt)
    else:
        np.savetxt(rules_filename, rules, delimiter=",",
            header=dataset_header, fmt='%d')

    print("Saving rules to: %s" % rules_filename)
else:
    # Load rules from file
    rules_filename = path_to_output_files + "rules-" + filename_time
        + ".csv"
    print("Loading rules from: %s" % rules_filename)
    all_comp_data, all_classes =
        common_tools.read_data(rules_filename, ",", \
              use_string_labels=False,
                  given_string_labels=save_class_as_string, \
```

```
                num_components=num_of_components, dtype='int8',
                    has_header=False, shuffle_rows=False)

        rules = np.hstack((all_comp_data, all_classes.reshape(-1, 1)))

    # Generate instance from LUT of rules filling up irrelevant
        components
    dataset = generate_dataset(rules, relevant_components,
        total_num_of_instances, instances_percentages, quant_levels)

    dataset_filename = path_to_output_files + "dataset-" + filename_time
        + ".csv"
    # Export data set to CSV file
    if save_class_as_string:
        dataset_labels = np.where(dataset[:, -1] == 0, 'A', 'B') # Save
            table of rules to file
        dataset_as_str = np.hstack((dataset[:, :-1].astype('S2'),
            dataset_labels.reshape(-1, 1)))
        np.savetxt(dataset_filename, dataset_as_str, delimiter=",",
            header=dataset_header, fmt=custom_fmt)
    else:
        np.savetxt(dataset_filename, dataset, delimiter=",",
            header=dataset_header, fmt='%u') # save only unsigned
            decimal integers

    print("Done. Saved data set file to: %s" % dataset_filename)
```

We now list the source code for our supervised learning method based on mating (grouping of values) across the supplied data set of components and classification labels. In short, the algorithm continues to exhaust all the possibilities for as long as accuracy of correct classification can be increased.

mate_learning.py

```
from __future__ import division
from __future__ import print_function

import numpy as np
import itertools
import time

class MateFinder(object):

    '''
    Used for the particular binary classification problem based on mates
        (usually groups of 2 - pairs) across each component of a
        discrete-valued data set.
    This class provides procedures such as the removal of irrelevant
        components (selection of relevant components),
```

```
as well as essential learning (training and validation) and testing
    procedures from learned decision tables of pairs.

Once a MateFinder object has been instantiated through the
    constructor, the interfacing methods employed by an external
    classifier are:
learn_by_mate_discovery() and compute_prediction_accuracy()
'''

def __init__(self, component_data, classification_labels,
    mating_size=2):
    '''
    Constructor for the PairFinder object out of the given data set.
    @summary: The data set (and corresponing classification labels)
        are split into the following portions:
       50% for learning phase: 25% training + 25% validation
       50% for testing phase: the real test using the learned table
           of decision rules

       This constructor initializes the reusable member variables
           that are called from other functions.

    @param component_data: A 2D matrix (numpy ndarray) of
        measurements (observations) to work with. A copy is created
        of this.
    @param classification_labels: The category classification_labels
        as a 1D ndarray. Expected to be nonnegative integers
    @param mating_size: It should default to 2 (for pairs), but as
        in life, mating (a.k.a grouping) can be done using larger
        number of members.
    Note, the number of values (10 digits for this exercise) must be
        divisible by the mating_size. For now, we basically can have
        only groups of 2 or 5.
    '''

    self.all_data = np.copy(component_data)
    self.all_classes = classification_labels
    self.num_of_instances = self.all_data.shape[0]
    self.num_of_components = self.all_data.shape[1]
    self.num_of_values = 10 # digits
    self.mating_size = mating_size
    self.num_of_mating_groups = self.num_of_components //
        self.mating_size
    self.map_base = np.repeat(np.arange(self.num_of_mating_groups),
        repeats=mating_size) # produces [0,0,1,1,2,2,3,3,4,4]

    N = self.num_of_instances
    # Get 50% for training:
    N_split = N // 2
    self.feat_data_learning, self.class_learning =
```

```
            self.all_data[:N_split], self.all_classes[:N_split]
        self.feat_data_test, self.class_test = self.all_data[N_split:],
            self.all_classes[N_split:]

        N_split_train_validate = N_split // 2
        self.feat_data_learning_train =
            self.feat_data_learning[:N_split_train_validate]
        self.class_learning_train =
            self.class_learning[:N_split_train_validate]
        self.feat_data_learning_validate =
            self.feat_data_learning[N_split_train_validate:]
        self.class_learning_validate =
            self.class_learning[N_split_train_validate:]
        self.N_train = N_split_train_validate

        # Segregate training data by class:
        self.c0_indices_train = np.where(self.class_learning_train ==
            0)[0].reshape(-1, 1)
        self.c1_indices_train = np.where(self.class_learning_train ==
            1)[0].reshape(-1, 1)
        self.feat_train_c0 = np.take(self.feat_data_learning_train[:,
            0], self.c0_indices_train)
        self.feat_train_c1 = np.take(self.feat_data_learning_train[:,
            0], self.c1_indices_train)
        for c in range(1, self.feat_data_learning_train.shape[1]):
            self.feat_train_c0 = np.hstack((self.feat_train_c0,
                np.take(self.feat_data_learning_train[:, c],
                self.c0_indices_train)))
            self.feat_train_c1 = np.hstack((self.feat_train_c1,
                np.take(self.feat_data_learning_train[:, c],
                self.c1_indices_train)))

        self.values_list = np.arange(self.num_of_values)

        mates_candidates = list(itertools.combinations(self.values_list,
            self.mating_size))
        all_groupings =
            np.array(list(itertools.combinations(mates_candidates,
            self.num_of_mating_groups)))
        self.mates_candidates_per_component =
            self.validate_groupings(all_groupings)

        # The address LUT multiplication factor (base)
        self.address_multiples = np.array([self.num_of_mating_groups **
            i for i in range(self.num_of_components)]).reshape(-1, 1)


    def validate_groupings(self, candidate_lists):
        '''
        It removes overlapping groupings (matings) among the list of
```

```
            groupings provided.

        @param candidate_lists: An ndarray of all possible groupings (in
            our particular exercise, these are all possible combinations
            of 2 out of 10 digits)

        @return: the filtered list of non-overlapping groupings. The
            reduction in length for this list is crucial for faster
            results during learning procedure.
        '''

        print("Removing overlapping groupings...", end="")
        t_start = time.clock()
        grouping_size = len(candidate_lists[0].ravel()) # A constant
            length
        skimmed_list = []
        for candidate in candidate_lists:
            c_unraveled = candidate.ravel()
            if len(np.unique(c_unraveled)) == grouping_size: # It means
                no overlaps exist, so it's a valid grouping
                skimmed_list.append(c_unraveled) # Provide unraveled list
                    instead of the array of groups (the candidate)
        t_end = time.clock()
        t_elapsed = t_end - t_start
        print("done! Elapsed: %.9f s" % (t_elapsed))

        return np.array(skimmed_list)

    def learn_by_mate_discovery(self):
        '''
        Brute force discovery of mates and relevant components during
            the training phase so that accuracy is the highest among all
            possibilities.

        @return The 1-d numpy array (table) of learned rules.
        @return List of best combination of mates (pairs) across the
            discovered relevant components
        @return The list of discovered relevant components from the
            training phase
        '''

        best_accuracy = 0
        best_matings = None
        relevant_components = None

        min_num_of_relevant_comps = 1
        for i in range(min_num_of_relevant_comps, self.num_of_components
            + 1):
            t_start = time.clock()
            current_accuracy, current_matings, comps =
```

```
                self.select_K_components(k_comps=i, verbose=True)
            if current_accuracy > best_accuracy:
                best_accuracy = current_accuracy
                best_matings = current_matings
                relevant_components = comps
            else: # When accuracy starts to decrease, it's time to stop
                because we have found the plateau concerning the number
                of relevant components.
                break

        t_end = time.clock()
        t_elapsed = t_end - t_start
        print("\nElapsed: %.9f s" % (t_elapsed))
        print("TRAINED Accuracy=%.2f%s with %d Selected best features:"
            % (best_accuracy * 100., "%", i), relevant_components)
        print("Using matings:", best_matings)

        decision_rules = self.generate_decision_rules_LUT(best_matings,
            relevant_components)

        return decision_rules, best_matings, relevant_components

    def select_K_components(self, k_comps, verbose=True):
        '''
        Find the k number of components that produce the highest
            prediction accuracy from the training procedure:

        Learning of rules is performed on the training portion of the
            data set (technically, the 25% of the entire data set)
        Accuracy validation to choose best is performed on the
            validation data set (technically, the other 25% of the
            entire data set)

        @param k_comps: Indicates the number of components to be
            attempted the "brute-force" search upon.

        @return The accuracy (normalized value from 0 to 1.0) obtained
            using the best selection of k components
        @return List of best combination of mates (pairs) across the
            best selected components
        @return The list of the best selected components
        '''

        components_list = range(self.num_of_components)

        best_accuracy = 0
        best_matings = None
        best_comps = None

        print("Performing %d-component selection" % (k_comps))
```

```
        t_start = time.clock()

        comps_combs = list(itertools.combinations(components_list,
            k_comps))
        for matings in self.mates_candidates_per_component:
            for subset in comps_combs:
                decisions_LUT = self.generate_decision_rules_LUT(matings,
                    subset)
                current_accuracy =
                    self.compute_prediction_accuracy(matings,
                    decisions_LUT, subset,
                    self.feat_data_learning_validate,
                    self.class_learning_validate, verbose=False)

                if current_accuracy > best_accuracy:
                    best_accuracy = current_accuracy
                    best_matings = matings
                    best_comps = subset

            print(".", end="")

        t_end = time.clock()
        t_elapsed = t_end - t_start
        if verbose:
            print("\nAccuracy = %.2f%s" % (best_accuracy * 100., "%"),
                "with matings:", best_matings)
            print("\tand components:", best_comps)
            print("\tIt took %.9f s" % (t_elapsed))

        return best_accuracy, best_matings, best_comps

    def _generate_training_LUT(self, cols):
        '''
        Generates an empty look-up table that will be addressed to train
            the learned with the given data.

        @param cols: the number of columns (components) in the data set
            that are being used for the training.

        @return: An empty 2-dimensional matrix that can be addressed
            while performing decision statistics
        '''
        len_of_table = self.num_of_mating_groups ** cols
        LUT = np.zeros((len_of_table, 2))
        return LUT

    def get_address(self, matings, component_data):
        '''
        Resolves the address in the training LUT pertaining the given
            matings (a list of candidate matches) on the data set
```

```
            portion provided.

        @param matings: List of indices associated with the groupings
            (matches)
        @param component_data: The ndarray of component data for which
            addresses are resolved for

        @return: the numpy n-dimensional array of resolved addresses for
            the provided data and desired matings
        '''

        address = np.empty_like(component_data, dtype="uint8")

        # Get value indices in matings array
        value_indices = np.zeros_like(matings) # + self.values_list

        for i in self.values_list:
            value_indices[i] = np.where(matings == i)[-1]

        # semi-vectorized address look-up
        for comp in range(component_data.shape[1]):
            address[:, comp] =
                self.map_base[value_indices[component_data[:, comp]]]

        return address

    def get_index_address_table(self, address):
        '''
        This is a vectorized function that resolves the index (a.k.a.
            hash address) on the 1-D array of rules
        employing in the learning/testing procedure).

        @param address: An array of address arrays to be hashed for on a
            1-D table using the pertaining base (such as base 5 when
            dealing with groups of 2)

        @return: the resolved 1-D array of indices corresponding the
            input address.
        '''
        hashed_address = np.dot(address,
            self.address_multiples[:len(address[-1])]) # Nice!
        return hashed_address

    def generate_decision_rules_LUT(self, matings, comp_subset):
        '''
        A decision rules are learned from the provided list of matings
            (non-overlapping groups) and the chosen components

        @param matings: An ordered list of indices used to group
            employing the map_based of choice set in the constructor.
```

```
        By default, the mapping is pair-wise ordered from left to right.
        @param comp_subset: The list of selected component indices upon
            where the decision table is generated

        @return: The 1-D table of decision rules extracted from
            frequency analysis out of the learning portion of the data
            set
        '''
        k_comps = len(comp_subset)
        learning_LUT_of_probs = self._generate_training_LUT(k_comps) #
            For relative statistics work

        addresses_0 = self.get_address(matings, self.feat_train_c0[:,
            comp_subset]) # Mapped addresses
        address_LUT_0 = self.get_index_address_table(addresses_0)
        addresses_1 = self.get_address(matings, self.feat_train_c1[:,
            comp_subset]) # Mapped addresses
        address_LUT_1 = self.get_index_address_table(addresses_1)

        # Count per catagory addresses and fill the table
        count_c0 = np.bincount(address_LUT_0[:, 0])
        count_c1 = np.bincount(address_LUT_1[:, 0])

        learning_LUT_of_probs[:len(count_c0), 0] = count_c0
        learning_LUT_of_probs[:len(count_c1), 1] = count_c1
        # Normalize
        learning_LUT_of_probs = learning_LUT_of_probs / self.N_train

        # Fill up decision rules LUT
        decision_rules_LUT = np.argmax(learning_LUT_of_probs, axis=-1)
        # WISH: For now, the LUT will get assigned zeros if there are
            any ties! which may appear biased towards class 0!

        return decision_rules_LUT

    def compute_prediction_accuracy(self, matings, decisions_LUT,
        relevant_components=None, comp_data_test=None, class_test=None,
        verbose=False):
        '''
        The accuracy of correct classification using the provided
            decision rules is computed statistically.

        @param matings: An ordered list of indices used to group
            employing the map_based of choice set in the constructor.
        By default, the mapping is pair-wise ordered from left to right.
        @param decisions_LUT: The learned rules in a LUT
        @param relevant_components: By default is set to None, so all
            components will be used. Otherwise, this specifies the
            selection of components to work with.
        @param comp_data_test: If any, this is the component data set to
```

```
            use for predicting upon and testing the classification
                accuracy with the provided matings and decision rules.
                Otherwise, it defaults to using the test data set (the 50%
                of the entire data set passed in the constructor)
            @param class_test: The classification labels associated with
                comp_data_test if any.
            @param verbose: Indicates whether to enable message verbosity.

            @return: The normalized accuracy of correct classification
                computed according to the specified parameters.
            '''
            if comp_data_test == None:
                observations, labels_true = self.feat_data_test,
                    self.class_test
            else:
                observations, labels_true = comp_data_test, class_test

            if relevant_components != None:
                observations = observations[:, relevant_components]

            N = observations.shape[0]

            predictions = self.predict_category(matings, decisions_LUT,
                observations)
            labels_true = labels_true.reshape(*predictions.shape)
            errors = predictions - labels_true
            correct_count = N - np.count_nonzero(errors)
            prob_correct = correct_count / float(N)
            if verbose:
                print("Accuracy = %.2f%s" % (prob_correct * 100., "%"), "with
                    matings:", matings)

            return prob_correct

        def predict_category(self, matings, decisions_LUT, instance_vector):
            '''
            With the given list of matings (groupings) and the decision
                rules table, the list of measurement vectors (instances) are
                classified accordingly.

            @param matings: An ordered list of indices used to group
                employing the map_based of choice set in the constructor.
            By default, the mapping is pair-wise ordered from left to right.
            @param decisions_LUT: The learned rules in a LUT
            @param instance_vector: The list of measurement vectors
                (instances, a.k.a observations) to classify.

            @return: The classification results corresponding to the
                supplied instance vectors for the specified rules and
                matings.
```

```
        '''
        addresses = self.get_address(matings, instance_vector) # Hashed
            address
        address_LUT = self.get_index_address_table(addresses)
        return decisions_LUT[address_LUT]
```

The following script drives the supervised learning project for binary classification. It uses the data set generated in order to learn and validate the decision rules with extremely high accuracy (almost perfect classification of 100%).

Popular classification techniques, such as logistic regression, Bayes multinomial classification, and kernel-based support vector machines (SVM), etc. fail to classify accurately (with less than 60% correctness as shown in Table 1).

learning_driver.py

```
from __future__ import division
from __future__ import print_function

import common_tools
import time

def classify_dataset(data_file_prefix, load_data_from_pickle=False):
    '''
    Reads the data set from the file name provided in order to perform
        learning and final classification test providing the resulting
        accuracy of correct classification.

    @param data_file_prefix: The prefix of the data set file name
    @param load_data_from_pickle: When set to True, a Python pickle
        (.pkl extension) will be used instead of the traditional comma
        separated valued (.csv extension) data set.
    '''

    from mate_learning import MateFinder

    if load_data_from_pickle:
        all_comp_data, all_classes =
            common_tools.load_dataset_from_pickle(data_file_prefix)
    else:
        all_comp_data, all_classes =
            common_tools.read_data("./InputData/" + data_file_prefix +
            ".csv", ",", \
                      use_string_labels=False, given_string_labels=True,
                          \
                      num_components=10, dtype='uint8', has_header=True,
                          shuffle_rows=True)

        common_tools.save_dataset_to_pickle(all_comp_data, all_classes,
            data_file_prefix)
```

```
    t_start = time.clock()
    match_maker = MateFinder(all_comp_data, all_classes, mating_size=2)

    # Learning phase
    decision_rules_LUT, best_matings, relevant_components =
        match_maker.learn_by_mate_discovery()

    # Final test
    final_accuracy =
        match_maker.compute_prediction_accuracy(best_matings,
        decision_rules_LUT, relevant_components)
    t_end = time.clock()
    t_elapsed = t_end - t_start
    print("_"*80)
    print("FINAL Accuracy = %.2f%s" % (final_accuracy * 100., "%"))
    print("_"*80)
    print("TOTAL elapsed time: %.9f s" % (t_elapsed))


if __name__ == '__main__':
    data_file_prefix =
        "dataset-3rel_components-quantized_butterfly-2014-12-27" #
        relevant_features = [0, 1, 8]

    classify_dataset(data_file_prefix, load_data_from_pickle=True)
```

7.2 Extra Credit Program

In this section, we list our Java implementation that solves the Extra Credit assignment. We start we the entry point of the program, the `ExtraCredit` class, which creates and runs all the necessary objects to solve the assignment.

ExtraCredit.java

```
package edu.ml.extraCredit;

import java.util.ArrayList;
import java.util.Random;

import edu.ml.project.utils.Dataset;
import edu.ml.project.utils.Utils;

/**
 * Routines to solve the extra credit assignment.
 * @author Juan Pablo Munoz
 * @author Carlos Jaramillo
```

```
 * @version 1.0
 */
public class ExtraCredit {

    private Dataset dataset;
    private Component [] components;
    private DecisionRule decider;
    private Quantizer q;

    private ArrayList<Character> trainingLabels;
    private ArrayList<Double[]> validation;
    private ArrayList<Character> validationLabels;
    private ArrayList<Double[]> testData;
    private ArrayList<Character> testLabels;

    private int numberOfComponents = 4;
    private double timeStart;
    private double timeLimit = 30;
    double minAcc = 0.71;
    int start = 3;
    int end = 6;

    /**
     * For final prediction of test data
     */
    private ArrayList<ArrayList<Double>> bestKconfigAndBoundaries = new
        ArrayList<ArrayList<Double>>(4);
    private double bestAccuracy;

    /**
     * Entry point for the program that solves the extra credit
         assignment
     * @param args
     */
    public static void main(String [] args){
        ExtraCredit ec = new ExtraCredit();
        ec.runAll("ML_python/InputData/pr_data.csv", 0.30, 0.30);
    }

    /**
     * Loads the data from a file and runs all the methods to optimize
         the number of levels and the position of the boundaries.
         Finally it uses the best result to predict the classes on the
         test data.
     * @param file
     * @param percentageTraining
     * @param percentageValidation
     */
    public void runAll(String file, double percentageTraining, double
        percentageValidation){
```

```
dataset =
    Utils.readAndRandomizeData("ML_python/InputData/pr_data.csv",
    0.30, 0.30);
components = dataset.components;
trainingLabels = dataset.trainingLabels;
validation = dataset.validation;
validationLabels = dataset.validationLabels;
testData = dataset.testData;
testLabels = dataset.testLabels;
numberOfComponents = dataset.numberOfComponents;
q = new Quantizer(components, trainingLabels);
q.relevantFeatures.add(0);
q.relevantFeatures.add(1);
q.relevantFeatures.add(2);
q.relevantFeatures.add(3);
decider = new DecisionRule(q);
System.out.println("Brute force optimization to find values of M
    using a range from 3 to 6 intervals in each component \n");
char [] c = new char[end-start+1];
for(int i=start; i<=end; i++){
    c[i-start] = Character.forDigit(i, 10);
}
ArrayList<String> candidates =
    Utils.bruteForceOptimization(minAcc, c, dataset, decider);
// Brute force optimization of M
System.out.println("Candidate\taccuracy");
for(String s : candidates){
    for(int i=0; i<numberOfComponents; i++){
        Utils.initComponent(components[i], Integer.parseInt("" +
            s.charAt(i)));
    }
    decider.reCreateTable();
    System.out.println(s + "\t\t" +
        Utils.computeAccuracy(validation, validationLabels,
        decider));
}
System.out.println("Brute force optimization to find values of
    boundaries for each candidate\n");
for(String s : candidates){
    timeStart = System.currentTimeMillis();
    int [] candidate = new int[numberOfComponents];
    for(int i=0; i<numberOfComponents; i++){
        candidate[i] = Integer.parseInt("" + s.charAt(i));
        Utils.initComponent(components[i], candidate[i]);
    }
    decider.reCreateTable();
    optimizeBoundaries(candidate);
}

// Final Prediction using Test data
```

```
    String bestConf = "";
    for(int i = 0; i<numberOfComponents; i++){
        int k = bestKconfigAndBoundaries.get(i).size();
        Utils.initComponent(components[i], k);
        components[i].restoreBoundaries(bestKconfigAndBoundaries.get(i));
        bestConf += k;
    }
    decider.reCreateTable();
    System.out.println("Final prediction on test data");
    System.out.println("Conf: " + bestConf + "\t\tAccuracy: " +
        Utils.computeAccuracy(testData, testLabels, decider));
    q.printBoundaries();
}

/**
 * This is an additional method that moves the boundaries to find
     the optimal boundaries. The difference between this method and
     the method that follows the procedure detailed in Prof.
     Haralick's slides is that when this method obtains a better
     accuracy by moving a boundary, it keeps moving this boundary in
     larger steps.
 * Boosting optimization
 * @param allFeatureValues
 */
private void optimizeBoundaries(int [] allFeatureValues){
    double accuracy = 0;
    for(int i=0; i<q.relevantFeatures.size(); i++){
            if(q.relevantFeatures.contains(new Integer(i))){
                Utils.initComponent(components[i],
                    allFeatureValues[i]);
            }
    }
    decider.reCreateTable();
    accuracy = Utils.computeAccuracy(validation, validationLabels,
        decider);
    double deltaFactor = 100;
    Random r = new Random();
    mainLoop: while(accuracy < 0.870){
        int i = r.nextInt(numberOfComponents);
        int bo = r.nextInt(components[i].getBoundaries().size()-1);
        Random r2 = new Random();
        double step = r2.nextDouble();
        double change = (2*step-1)/deltaFactor;
        ArrayList<Double> oldB = components[i].moveRandomBoundary(bo,
            change);
        if(!components[i].boundariesChanged){
            continue mainLoop;
        }
        components[i].boundariesChanged = false;
        decider.reCreateTable();
```

```
            double newAccuracy = Utils.computeAccuracy(validation,
                validationLabels, decider);
            if(newAccuracy < accuracy){
                components[i].restoreBoundaries(oldB);
            }
            while(newAccuracy > accuracy){
                accuracy = newAccuracy;
                oldB = components[i].moveRandomBoundary(bo, change);
                if(!components[i].boundariesChanged){
                    continue mainLoop;
                }
                components[i].boundariesChanged = false;
                decider.reCreateTable();
                newAccuracy = Utils.computeAccuracy(validation,
                    validationLabels, decider);
                if(newAccuracy < accuracy){
                    components[i].restoreBoundaries(oldB);
                }
            }
            if(isOverTime()){
                System.out.println(allFeatureValues[0] + " " +
                    allFeatureValues[1] + " " + allFeatureValues[2] + " "
                    + allFeatureValues[3] + " After " + timeLimit + "
                    minutes, accuracy: " + accuracy);
                if(bestAccuracy <= accuracy){
                    bestAccuracy = accuracy;
                    bestKconfigAndBoundaries.removeAll(bestKconfigAndBoundaries);
                    for(int i1=0; i1<numberOfComponents; i1++){
                        bestKconfigAndBoundaries.add(components[i1].getBoundaries());
                    }
                }
                return;
            }

        }
        System.out.print(allFeatureValues[0] + " " + allFeatureValues[1]
            + " " + allFeatureValues[2] + " " + allFeatureValues[3] + "
            ");
        optimizeBoundaries2();
    }

    /**
     * It optimizes the boundaries using the procedure detailed in Prof.
         Haralick's slides
     */
    private void optimizeBoundaries2(){
        decider.reCreateTable();
        double accuracy = Utils.computeAccuracy(validation,
            validationLabels, decider);
        double deltaFactor = 100;
```

```
Random r = new Random();
while(true){
    int i = r.nextInt(4);
    int bo = r.nextInt(components[i].getBoundaries().size()-1);
    Random r2 = new Random();
    double step = r2.nextDouble();
    double change = (2*step-1)/deltaFactor;
    double maxAccuracy = accuracy;
    ArrayList<Double> maxBound = null;
    ArrayList<Double> oldB = components[i].getCopyOfBoundaries();
    ArrayList<Double> savedEqualStep = null;
    boolean savedBestStepEqualAcc = false;
    for(int j=1;
        j<r2.nextInt(components[i].getBoundaries().size()*8);
        j++){
        components[i].moveRandomBoundary(bo, change);
        if(!components[i].boundariesChanged){
            change /= 2; // 1.5 //
        }
        else{
            components[i].boundariesChanged = false;
            decider.reCreateTable();
            double newAccuracy = Utils.computeAccuracy(validation,
                validationLabels, decider);
            if(newAccuracy > maxAccuracy){
                maxAccuracy = newAccuracy;
                maxBound = new
                    ArrayList<Double>(components[i].getCopyOfBoundaries());
            }
            else if(newAccuracy == maxAccuracy){
                if(!savedBestStepEqualAcc){
                    savedBestStepEqualAcc = true;
                    savedEqualStep = new
                        ArrayList<Double>(components[i].getCopyOfBoundaries());
                }
            }
        }
    }
    if(maxAccuracy < accuracy){
        components[i].restoreBoundaries(oldB);
    }
    else if(maxAccuracy == accuracy && savedBestStepEqualAcc){
        components[i].restoreBoundaries(savedEqualStep);
    }
    else if(maxAccuracy == accuracy){
        components[i].restoreBoundaries(oldB);
    }
    else{
        accuracy = maxAccuracy;
        components[i].restoreBoundaries(maxBound);
```

```
            }
            if(isOverTime()){
                System.out.println(" After "+ timeLimit +" minutes,
                     accuracy: " + accuracy);
                if(bestAccuracy <= accuracy){
                    bestAccuracy = accuracy;
                    bestKconfigAndBoundaries.removeAll(bestKconfigAndBoundaries);
                    for(int i1=0; i1<numberOfComponents; i1++){
                        bestKconfigAndBoundaries.add(components[i1].getBoundaries());
                    }
                }
                return;
            }

        }
    }

    /**
     * @return whether to continue running the current configuration.
     */
    private boolean isOverTime() {
        if((timeStart + (timeLimit * 60000)) <
             System.currentTimeMillis()){
             return true;
        }
        return false;
    }
}
```

Quantizer.java

```
package edu.ml.extraCredit;

import java.util.ArrayList;
import java.util.Hashtable;

/**
 * This class contains all the components and the labels for the
     training data.
 * It computes the address table.
 *
 * @author Juan Pablo Munoz
 * @author Carlos Jaramillo
 * @version 1.0
 */
public class Quantizer {
    int numberOfInstances;
    ArrayList<Character>labels;
    Component [] components;
    public ArrayList<Integer> relevantFeatures;
```

```
/**
 * Constructor
 * @param comps Array of components.
 * @param labelCollection The correct labels in the training set.
 */
public Quantizer(Component [] comps,
    ArrayList<Character>labelCollection){
    components = comps;
    labels = labelCollection;
    numberOfInstances = labels.size();
    relevantFeatures = new ArrayList<Integer>();
}

/**
 * @return a Hash table with addresses and probabilities for classes
     A and B
 */
public Hashtable<String, Double[]> getAddressHashTable() {
    Hashtable<String, Double[]> table = new Hashtable<String,
        Double[]>();
    for(int i = 0; i<numberOfInstances; i++){
        String key = "";
        int countComp = 0;
        for(Component c :components){
            if(relevantFeatures.contains(new Integer(countComp))){
                key += c.getInterval(c.getDataPoints().get(i));
            }
            countComp++;
        }
        Double[] val;
        if(table.containsKey(key)){
            val = table.get(key);
        }
        else{
            val = new Double[2];
            val[0] = 0.0;
            val[1] = 0.0;
        }
        if(labels.get(i) == 'A'){
            val[0]++;
        }
        else{
            val[1]++;
        }
        table.put(key, val);
    }
    ArrayList<Double[]> vals = new
        ArrayList<Double[]>(table.values());
    for(Double[] d : vals){
```

```
            d[0] = d[0]/numberOfInstances;
            d[1] = d[1]/numberOfInstances;
        }
        return table;
    }
    /**
     * Creates a key string from a vector. This key can be used to query
         entries in the hash table.
     * @param vector
     * @return
     */
    public String getStringAddress(Double [] vector) {
        String key = "";
        int i = 0;
        for(Component c :components){
            if(relevantFeatures.contains(new Integer(i))){
                key += c.getInterval(vector[i]);
            }
            i++;
        }
        return key;
    }

    /**
     * @return the size of the hash table.
     */
    public int getHashTableMaxSize(){
        int maxSizeTable = 1;
        for(Component c :components){
            maxSizeTable *= c.getBoundaries().size();
        }
        return maxSizeTable;
    }

    /**
     * Prints the boundaries for each of the components.
     */
    public void printBoundaries(){
        int comp = 0;
        for(Component c : components){
            if(relevantFeatures.contains(new Integer(comp))){
                System.out.println("Component " + comp + " Boundary:
                     -inf");
                for(Double d : c.getBoundaries()){
                    if(c.getBoundaries().indexOf(d) !=
                         c.getBoundaries().size()-1){
                        System.out.println("Component " + comp + "
                             Boundary: " + d);
                    }
                    else{
```

```
                    System.out.println("Component " + comp + "
                        Boundary: +inf");
                }
            }

        }
        comp++;
    }
  }
}
```

Component.java

```
package edu.ml.extraCredit;

import java.util.ArrayList;
import java.util.Arrays;
/**
 * This class stores all the values of a component of a vector. It also
     contains the values for the location of each boundary.
 * @author Juan Pablo Munoz
 * @author Carlos Jaramillo
 * @version 1.0
 *
 */
public class Component {
    private ArrayList<Double> dataPoints;
    private Object[] sortedPoints;
    private ArrayList<Double> boundaries;
    public boolean boundariesChanged;

    /**
     * Constructor
     * Initializes dataPoints ArrayList.
     *
     */
    public Component(){
        dataPoints = new ArrayList<Double>();
    }
    /**
     * Initializes uniform boundaries
     *
     * @param nLevels The number of levels (a.k.a intervals) that this
         component has.
     */
    public void initialBoundaries(int nLevels){
        boundaries = new ArrayList<Double>();
        sortedPoints = dataPoints.toArray();
        Arrays.sort(sortedPoints);
```

```
    int pointsInInterval = sortedPoints.length/nLevels;
    int counter = 0;
    for(int i=0; i<sortedPoints.length; i++){
        counter++;
        if(counter == pointsInInterval){
            boundaries.add((Double)sortedPoints[i]);
            counter = 0;
        }
    }
    boundaries.set(boundaries.size()-1, Double.MAX_VALUE);
}
/**
 * sets data points for training
 * @param trainingData
 */
public void setDataPoints(ArrayList<Double> trainingData){
    dataPoints = trainingData;
}
/**
 * Finds the interval that corresponds to a particular value
 * @param value
 * @return level / interval
 */
public int getInterval(double value){
    for(int i=0; i<boundaries.size(); i++){
        if(i==0){
            if(value <= boundaries.get(i)){
                return i;
            }
        }
        else{
            if(value>boundaries.get(i-1) && value<=boundaries.get(i)){
                return i;
            }
        }
    }
    System.err.println("Error getting interval");
    return -99;
}
/**
 * @return all the values in this component.
 */
public ArrayList<Double> getDataPoints(){
    return dataPoints;
}

/**
 * @return the current boundaries
 */
public ArrayList<Double> getBoundaries(){
```

```
    return boundaries;
}

/**
 * @return a copy of the current boundaries
 */
public ArrayList<Double> getCopyOfBoundaries(){
    ArrayList<Double> copy = new ArrayList<Double>();
    for(Double d : boundaries){
        copy.add(d);
    }
    return copy;
}

/**
 * It restores the boundaries.
 * @param restoreBoundaries
 */
public void restoreBoundaries(ArrayList<Double> restoreBoundaries) {
    boundaries = restoreBoundaries;
}
/**
 * It moves a boundary.
 * @param boundaryNumber
 * @param variation
 * @return
 */
public ArrayList<Double> moveRandomBoundary(int boundaryNumber,
     double variation) {
    ArrayList<Double> oldBoundary = new ArrayList<Double>();
    for(Double d : boundaries){
        oldBoundary.add(d);
    }
    if(boundaryNumber == 0){
        if((boundaries.get(boundaryNumber) + variation) >
             boundaries.get(boundaryNumber+1) ||
             (boundaries.get(boundaryNumber) + variation) < 0){
            boundariesChanged = false;
            return oldBoundary;
        }
    }
    else if((boundaries.get(boundaryNumber) + variation) >
         boundaries.get(boundaryNumber+1) ||
         (boundaries.get(boundaryNumber) + variation) <
         boundaries.get(boundaryNumber-1)){
        boundariesChanged = false;
        return oldBoundary;
    }
    boundaries.set(boundaryNumber,
         boundaries.get(boundaryNumber)+variation);
```

```
        boundariesChanged = true;
        return oldBoundary;
    }
}
```

DecisionRule.java

```
package edu.ml.extraCredit;

import java.util.ArrayList;
import java.util.Hashtable;

/**
 * The decision rule. It looks for an address in a table and predicts
     the class for an instance.
 * @author Juan Pablo Munoz
 * @author Carlos Jaramillo
 * @version 1.0
 *
 */
public class DecisionRule {
    Quantizer quantizer;
    Hashtable<String, Double[]> table;

    /**
     * Constructor
     * Receives a quantizer object and creates the hash table that
         contains addresses and probabilities.
     * @param q The quantizer object
     */
    public DecisionRule(Quantizer q){
        quantizer = q;
        table = new Hashtable<String, Double[]>();
//      printAddressHashTable();

    }

    /**
     * Recreates the Hash table. Used after boundaries have changed.
     */
    public void reCreateTable(){
        table = quantizer.getAddressHashTable();
    }

    /**
     * Prints the hash table - address | probA | probB
     */
    public void printAddressHashTable(){
        for(String s : table.keySet()){
```

```
            System.out.println(s + "\t" + table.get(s)[0] + "\t" +
                table.get(s)[1]);
        }
    }

    /**
     * Predicts the class for a particular vector.
     * @param vector
     * @return
     */
    public Character predictCharHashTable(Double [] vector){
        String key = quantizer.getStringAddress(vector);
        Double [] probs = table.get(key);
        if(probs == null){ // Like tossing a die.
            return 'A';
        }
        if(probs[0] >= probs[1]){
            return 'A';
        }
        else{
            return 'B';
        }
    }

    /**
     * @return the relevant features of a vector.
     */
    public ArrayList<Integer> getRelevantFeatures() {
        return quantizer.relevantFeatures;
    }
}
```

Dataset.java

```
package edu.ml.project.utils;

import java.util.ArrayList;

import edu.ml.extraCredit.Component;

/**
 * Creates objects that contain all data sets
 * @author Juan Pablo Munoz
 * @author Carlos Jaramillo
 *
 */
public class Dataset {
    public ArrayList<Double[]> training;
    public ArrayList<Character> trainingLabels;
    public ArrayList<Double[]> validation;
```

```
    public ArrayList<Character> validationLabels;
    public ArrayList<Double[]> testData;
    public ArrayList<Character> testLabels;
    public int numberOfComponents = 4; // Default.
    public Component [] components;
}
```

Utils.java

```
package edu.ml.project.utils;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;

import edu.ml.extraCredit.Component;
import edu.ml.extraCredit.DecisionRule;

/**
 * Auxiliary methods.
 *
 * @author Juan Pablo Munoz
 * @author Carlos Jaramillo
 * @version 1.0
 *
 */
public class Utils {
    /**
     * Add all requested permutations to an ArrayList of Strings.
     * @param valuesToBeUsed
     * @param sizeOfPermutation
     * @param permutation
     * @param col
     */
    public static void permutations(char[] valuesToBeUsed, int
        sizeOfPermutation, String permutation, ArrayList<String> col){
        if(permutation.length() >= sizeOfPermutation){
//          System.out.println(result);
            col.add(permutation);
        }else{
            for(char value : valuesToBeUsed){
                if(permutation.length()==0){
                    permutations(valuesToBeUsed, sizeOfPermutation,
                        permutation+value, col);
                }
                else{
```

```
                permutations(valuesToBeUsed, sizeOfPermutation,
                    permutation+""+value, col); // "" in case we need
                    a separator
            }
        }
    }
}

/**
 * @param valuesToBeCombined
 * @param ar
 */
public static void combination(String valuesToBeCombined,
    ArrayList<String> ar) {
    combination("", valuesToBeCombined, ar);
}
/**
 * @param currentStr
 * @param combinedStr
 * @param ar
 */
private static void combination(String currentStr, String
    combinedStr, ArrayList<String>ar) {
    if (combinedStr.length() > 0) {
        ar.add(currentStr + combinedStr.charAt(0));
        combination(currentStr + combinedStr.charAt(0),
            combinedStr.substring(1), ar);
        combination(currentStr, combinedStr.substring(1), ar);
    }
}


/**
 * Returns all combinations of relevant features and the
    permutations of levels above a certain accuracy.
 * TODO: This function works only for number of intervals < 10
 * @param minAcc The minimun accuracy for a configuration to be
    considerate a candidate
 * @param c Character array of relevant features
 * @param dataset All the data
 * @param decider The Decision Rule object
 * @return The candidates, i.e., configurations that have resulted
    in a good accuracy.
 */
public static ArrayList<String> bruteForceOptimization(double
    minAcc, char [] c, Dataset dataset, DecisionRule decider) {
    ArrayList<String> candidates = new ArrayList<String>();
    ArrayList<String> permutations = new ArrayList<String>();
    ArrayList<Integer> relFeat = decider.getRelevantFeatures();
    Utils.permutations(c, relFeat.size(),"", permutations);
```

```
        for(String s : permutations){
            int count = 0;
            for(int i=0; i<dataset.numberOfComponents; i++){
                if(relFeat.contains(new Integer(i))){
                    Utils.initComponent(dataset.components[i],
                        Integer.parseInt("" + s.charAt(count)));
                    count++;
                }
            }
            decider.reCreateTable();
            double bruteForceExp =
                Utils.computeAccuracy(dataset.validation,
                dataset.validationLabels, decider);
            if(bruteForceExp > minAcc){
                candidates.add(s);
            }
        }
//      System.out.println("done with Brute Force gathering list of
    possible candidates");
        return candidates;
    }

    /**
     * Reads a file containing all the data. Randomizes the data and
         creates three sets: training, validation, and test.
     * @param file
     * @param percentageTraining
     * @param percentageValidation
     * @return All data sets
     */
    public static Dataset readAndRandomizeData(String file, double
        percentageTraining, double percentageValidation){
        BufferedReader reader;
        Dataset dataset = new Dataset();
        dataset.training = new ArrayList<Double[]>();
        dataset.trainingLabels = new ArrayList<Character>();
        dataset.validation = new ArrayList<Double[]>();
        dataset.validationLabels = new ArrayList<Character>();
        dataset.testData = new ArrayList<Double[]>();
        dataset.testLabels = new ArrayList<Character>();
        try {
            reader = new BufferedReader(new FileReader(file));
            Random r = new Random();
            String dataLine = reader.readLine(); //Disregard header
            String [] tokens = dataLine.split(",", 0); //split(dataLine,
                ',');
            // Use length of header to initialize components.
            dataset.numberOfComponents = tokens.length-1; // TODO: Move
                this out of here.
```

```
            dataset.components = new
                Component[dataset.numberOfComponents];
            for(int i = 0; i<dataset.numberOfComponents; i++){
                dataset.components[i] = new Component();
            }
            while((dataLine = reader.readLine()) != null){
                tokens = dataLine.split(",", 0); //split(dataLine, ',');
                Double [] features = new Double[tokens.length-1];
                for(int i = 0; i<tokens.length-1; i++){
                    features[i] = Double.valueOf(tokens[i]);
                }
                Character label =
                    tokens[tokens.length-1].toCharArray()[0];
                double randomVal = r.nextDouble();
                if(randomVal <percentageTraining){
                    dataset.training.add(features);
                    dataset.trainingLabels.add(label);
                    for(int i = 0; i<features.length; i++){
                        dataset.components[i].getDataPoints().add(features[i]);
                    }
                }
                else if(randomVal < (percentageTraining +
                    percentageValidation)){
                    dataset.validation.add(features);
                    dataset.validationLabels.add(label);
                }
                else{
                    dataset.testData.add(features);
                    dataset.testLabels.add(label);
                }
            }
            System.out.println("Report\n");
            System.out.println("Number of instances for training: " +
                dataset.training.size());
            System.out.println("Number of instances for validation: " +
                dataset.validation.size());
            System.out.println("test size: " + dataset.testData.size());
            System.out.println();
            reader.close();
        } catch (FileNotFoundException e1) {
            e1.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return dataset;
    }

    /**
     * Initializes the boundaries of a component.
     * @param component
```

```
     * @param boundaries
     */
    public static void initComponent(Component component, int
        boundaries){
        component.initialBoundaries(boundaries);
    }

    /**
     * Computes the accuracy of a Decision Rule in a particular data set.
     * @param cData
     * @param cLabels
     * @param decider
     * @return accuracy value
     */
    public static double computeAccuracy(ArrayList<Double[]> cData,
        ArrayList<Character> cLabels, DecisionRule decider){
        int correct = 0;
        int instances = 0;
        int labelI = 0;
        for(Double [] v : cData){
            if(decider.predictCharHashTable(v) == cLabels.get(labelI)){
                correct++;
            }
            instances++;
            labelI++;
        }
        return (double)correct/instances;
    }
}
```