Off-line Character Recognition

STAT 602 Final Project

Yuchi Hu

May 2, 2019

Contents

Abstract

Handwriting recognition is the ability of a computer to receive and interpret handwriting from paper, photographs, touch-screens, etc. There are two types of handwriting recognition: off-line recognition and on-line recognition. In off-line recognition, input is scanned from text on a piece of paper written in the past. In on-line recognition, input is obtained from text as it is written on devices such as tablets and smart phones. Inputs are then converted into bitmaps that can be interpreted by the computer. In this project, we will focus on off-line recognition.

1. Problem Statement

In off-line recognition, the image of the written text is optically scanned then converted into bitmaps that are usable within computer applications. Off-line recognition is difficult compared to on-line recognition since different people have different handwriting styles and on-line recognition has the advantage of sensing the pen-tip movements across the writing surface.

2. Background

We have been given two datasets:

- letters.unlabeled.csv: Is a csv file composed of 10,000 unlabeled handwritten characters.
- letters.labeled.csv: Is a csv file composed of 1,000 labeled handwritten characters.

The goal of this project is to use both datasets to construct multiple classifiers, then use the best classifier to predict the characters for a third dataset composed of 30,000 unlabeled handwritten characters.

3. Exploratory Data Analysis

First, we look at the structure of the labeled data (letters.labeled.csv) and unlabeled data (letters.unlabeled.csv).

```
## Labeled data dimensions: 1000 3138
```

```
## Unlabeled data dimensions: 10000 3137
```

Table 1: Snippet of the Labeled Data (First 10 Rows and First 7 Columns)

X	Letter	Pixel.1	Pixel.2	Pixel.3	Pixel.4	Pixel.5
1	t	0	0	0	0	0
2	r	0	0	0	0	0
3	d	0	0	0	0	0
4	t	0	0	0	0	0
5	e	0	0	0	0	0
6	d	0	0	0	0	0
7	v	0	0	0	0	0
8	p	0	0	0	0	0
9	3	0	0	0	0	0
10	z	0	0	0	0	0

Table 2: Snippet of the Unlabeled Data (First 10 Rows and First 7 Columns)

X	Pixel.1	Pixel.2	Pixel.3	Pixel.4	Pixel.5	Pixel.6
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0

The labeled data has 1,000 rows or observations and 3,138 columns. The first column is simply an index or observation number, which we can drop. The second column (*Letter*) is the label or class for each character, i.e. it is the response variable. Columns three to 3,138 (*Pixel.1* to *Pixel.3136*) represent the 3,136 pixels in the 56 by 56 bitmap. These pixels are the predictors and are binary, taking on a value of 0 if the pixel is white and a value of 1 if the pixel is black.

The unlabeled data has 10,000 rows or observations and 3,137 columns. Similar to the labeled data, the first column of the unlabeled data is an index, which we can drop; however, the unlabeled data does not contain *Letter* (the response variable).

```
## Number of NA values: 0
```

A neat thing about both the labeled and unlabeled data is that there are no missing values, which makes our analysis easier.

Figure 1 plots the pixels of the first 36 characters from the labeled data, and **Figures 2** and **3** plot the pixel means and standard deviations of each character, respectively. We see that there is a large amount of variance in the data.

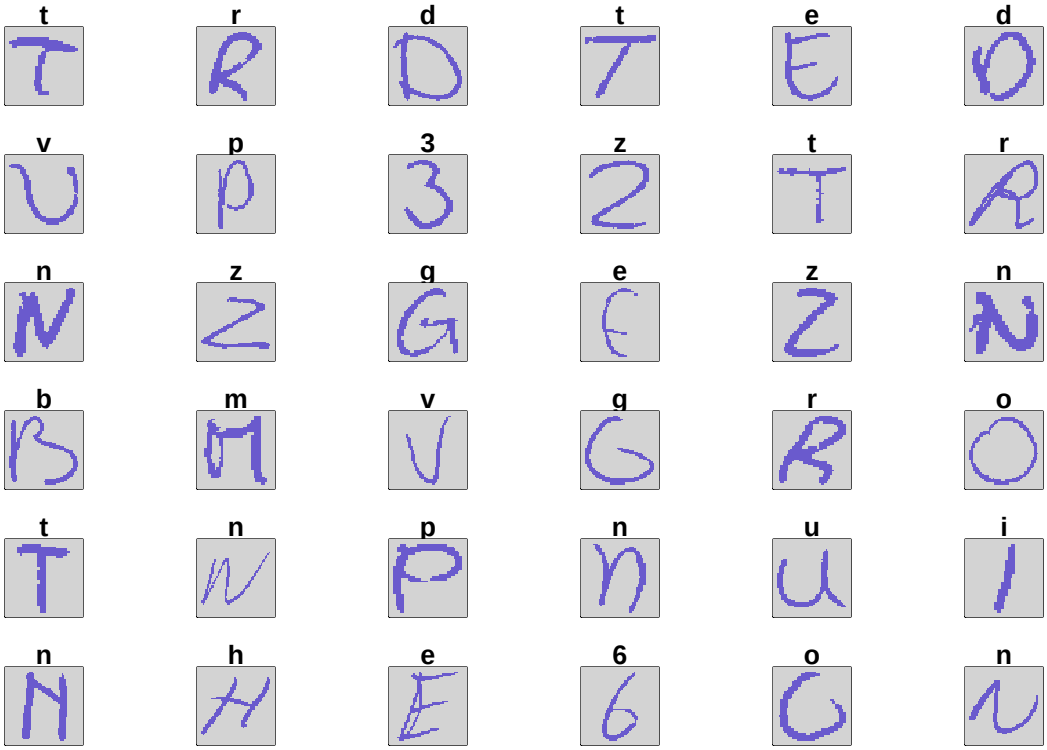

Figure 1: Plot of the pixels of the first 36 characters from the labeled data.

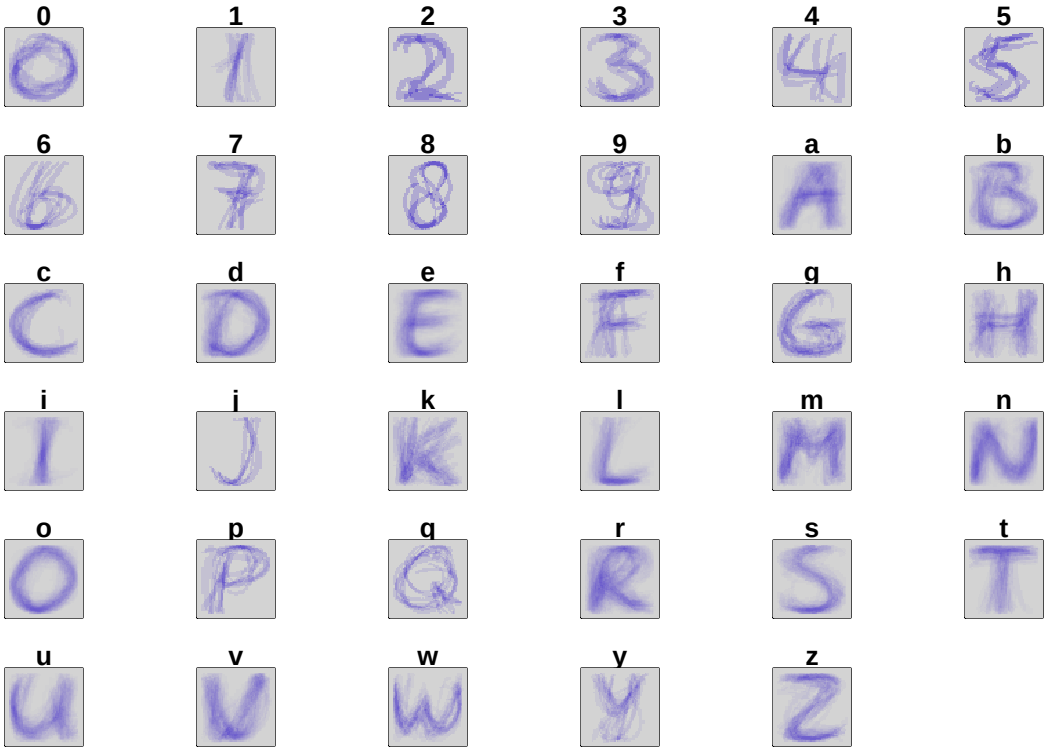

Figure 2: Plot of the pixel means of each character from the labeled data.

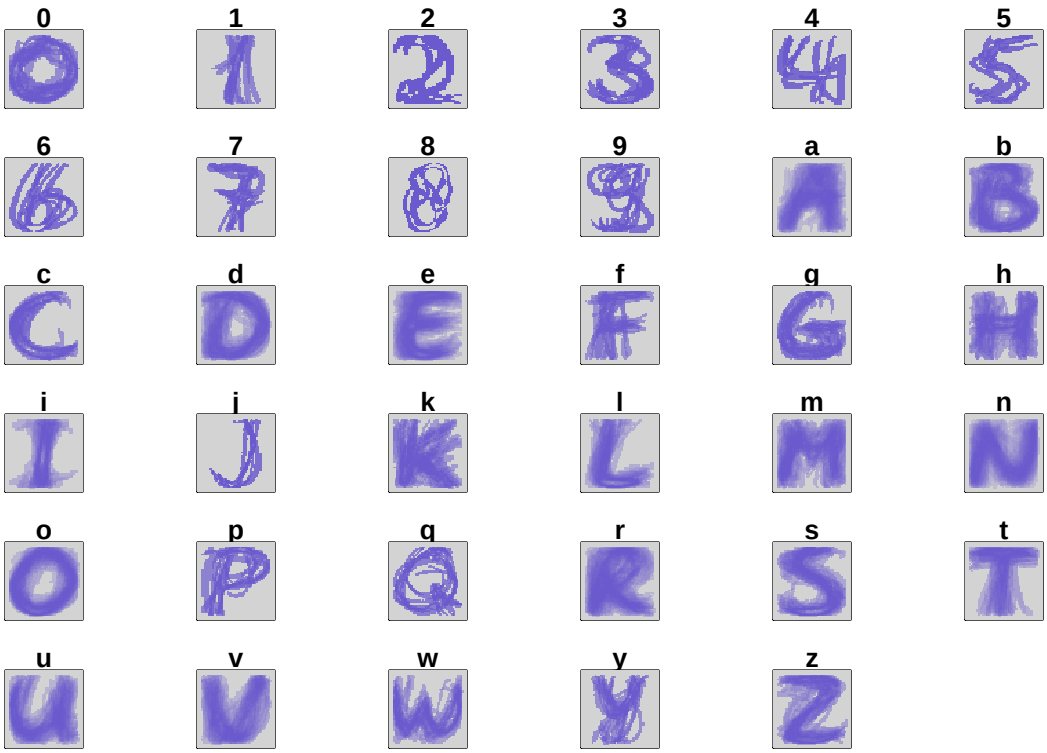

Figure 3: Plot of the pixel standard deviations of each character from the labeled data.

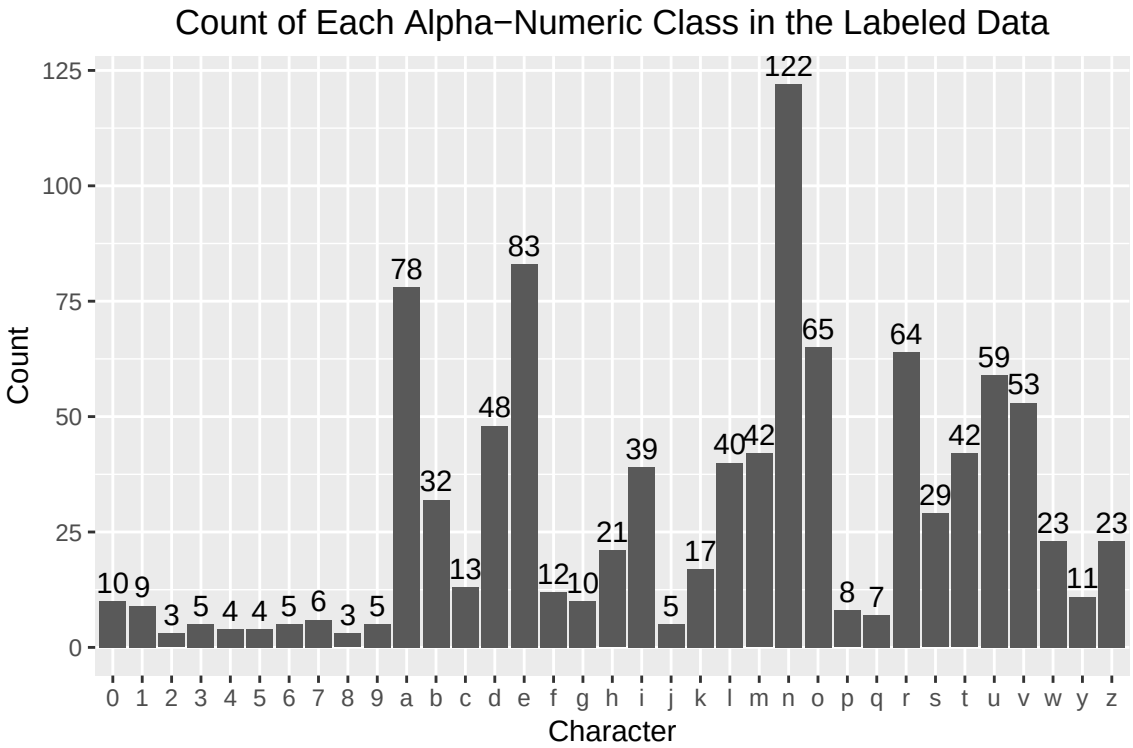

Figure 4: Count of each alpha-numeric class in the labeled data.

Figure 4 shows the count of each alpha-numeric class in the labeled data. We see that the classes are heavily unbalanced, ranging from 122 observations for the letter n and 3 observations each for the numbers 2 and 8. We should also note that there is no letter x in this data. Overall, there are far fewer numbers than letters (54 numbers vs. 946 letters).

4. Labeling the Unlabeled Data

First, we manually label the unlabeled data based on human judgment. Some of the 0's/o's and 1's/i's are impossible to tell apart, so for those characters, we initially label them as "0 or o" and "1 or i".

4.1 Labeling 0's and o's in the Unlabeled Data

We fit a random forest model to the 0's and o's from the labeled data. Then, we use that model to predict the labels for the characters initially labeled as "0 or o" from the unlabeled data. However, in order for the predictions to not be dominated by o's, we oversample and undersample the 0's and o's, respectively, from the labeled data. As shown in the output below, 77 of those characters were predicted to be 0's, and 672 were predicted to be o's.

```
##   0   o
##  77 672
```

Figure 5 shows the plot of the pixels of 36 of the characters intially labeled as "0 or o" from the unlabeled data. The random forest predicted labels are shown above each image. Interestingly, all of these characters look about the same, but random forest was able to classify them nevertheless.

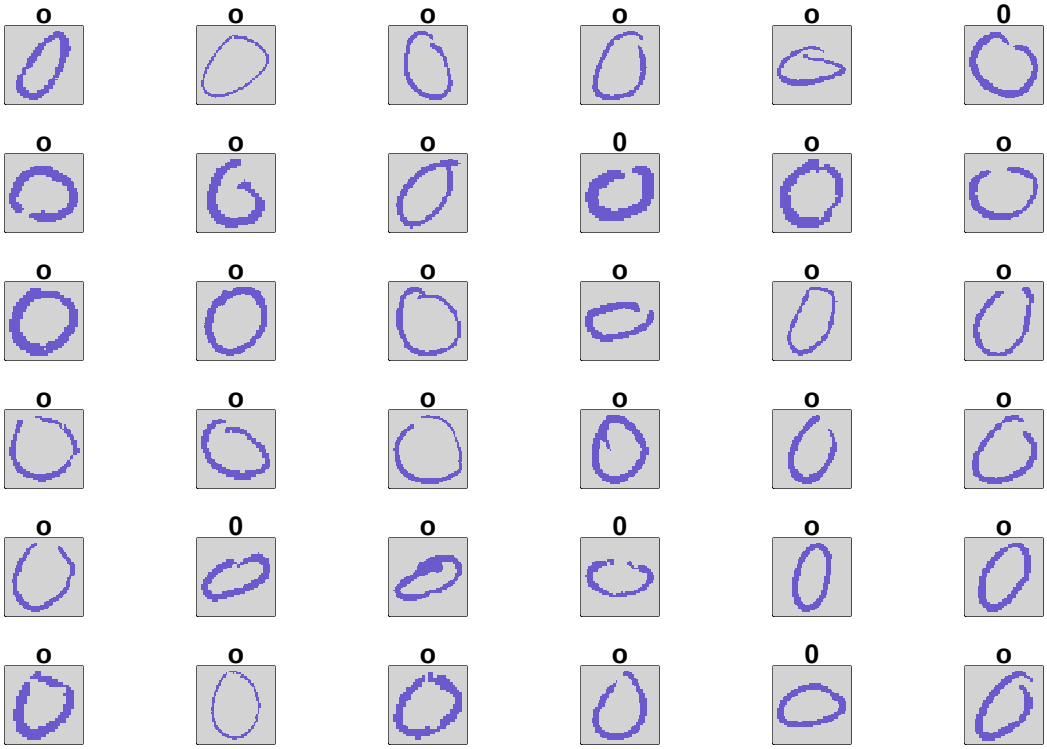

Figure 5: Plot of the pixels of 36 of the characters initially labeled as "0 or o" from the unlabeled data.

4.2 Labeling 1's and i's in the Unlabeled Data

We fit a random forest model to the 1's and i's from the labeled data. Then, we use that model to predict the labels for the characters initially labeled as "1 or i" from the unlabeled data. However, in order for the predictions to not be dominated by i's, we oversample and undersample the 1's and i's, respectively, from the labeled data. As shown in the output below, 71 of those characters were predicted to be 1's, and 477 were predicted to be i's.

```
##   1   i
##  71 477
```

Figure 6 shows the plot of the pixels of 36 of the characters intially labeled as "1 or i" from the unlabeled data. The random forest predicted labels are shown above each image. Interestingly, all of these characters look about the same, but random forest was able to classify them nevertheless.

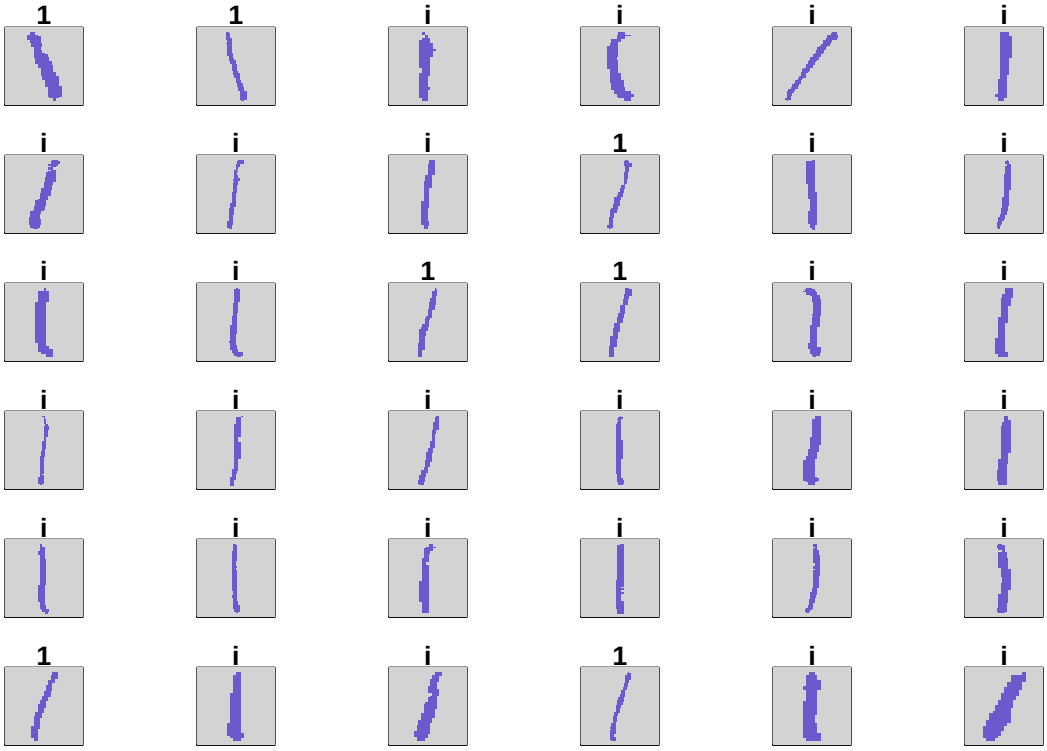

Figure 6: Plot of the pixels of 36 of the characters initially labeled as "1 or i" from the unlabeled data.

4.3 Distribution of the Labels of the Unlabeled Data (After Labeling)

No doubt our technique for labeling would result in errors; this is inherent in the identical appearance of some 0's/o's, 1's/i's, etc. **Figure 7** shows the count of each alpha-numeric class in the unlabeled data (after labeling). We see that the distribution of the labels of the unlabeled data is similar to that of the labeled data; however, we probably overestimated the number of 0's and 1's. Again, this is due to the identical appearance of some 0's/o's and 1's/i's.

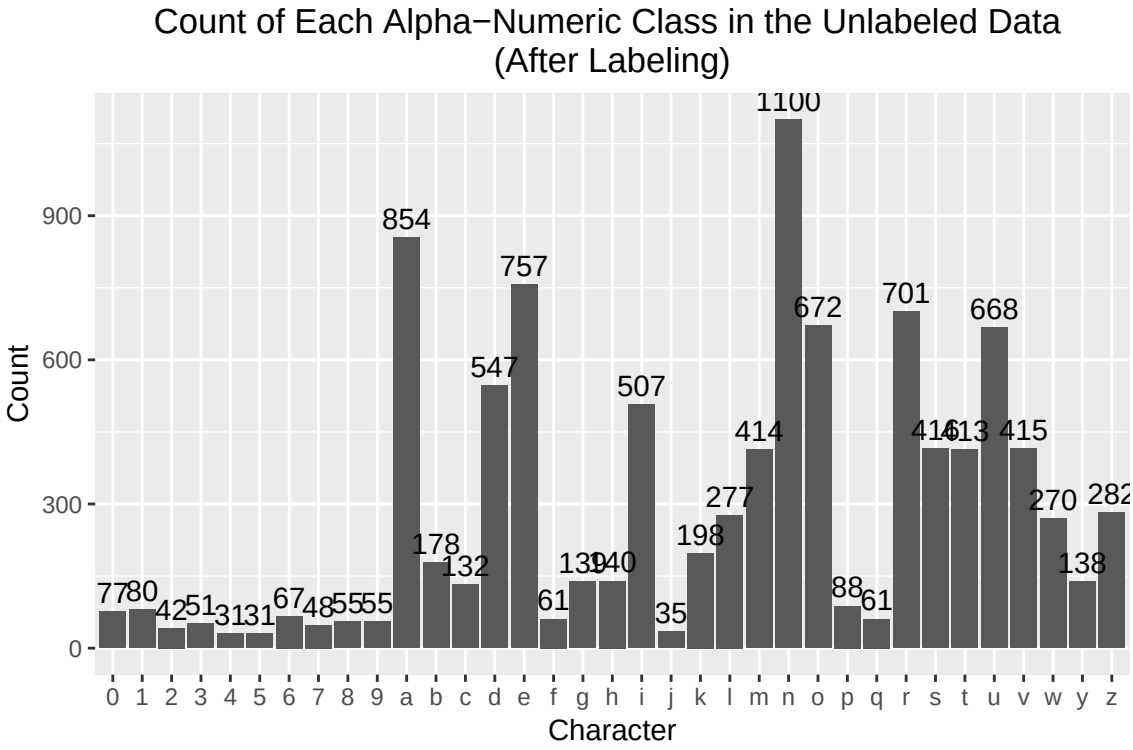

Figure 7: Count of each alpha-numeric class in the unlabeled data (after labeling).

5. Dealing with the Unbalanced Classes in the Unlabeled Data

Since the classes in the unlabeled data are heavily unbalanced, any model we train will be biased towards the more common classes, resulting in a deceptively high accuracy. To combat this problem, we apply oversampling to the minority classes and undersampling to the majority classes. SMOTE (Synthetic Minority Over-Sampling Technique) is an algorithm that artificially generates new examples of the minority classes by considering their K-nearest neighbors in feature space; this method is known as oversampling. In addition, the majority classes are undersampled, leading to a more balanced dataset [7]. We apply oversampling and undersampling to the unlabeled data using **SmoteClassif** from the **UBL** package. The resulting distribution of the classes in the unlabeled data is shown below:

```
##   0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f   g   h
## 286 286 285 285 286 286 285 285 286 286 286 286 286 286 286 286 286 285
##   i   j   k   l   m   n   o   p   q   r   s   t   u   v   w   y   z
## 286 285 286 286 286 286 286 286 286 286 286 286 286 286 286 286 286
```

We see that the data is now balanced.

6. Choosing the Training and Validation Sets

Unlike most situations in which we randomly split a dataset into training and validation sets, we will use the unlabeled data (after labeling and applying oversampling and undersampling) as the training set and the labeled data as the validation set to get an unbiased estimate of the test error. From here on out, when we

mention the training set, we are referring to the unlabeled data, and when we mention the validation set, we are referring to the labeled data. The training set has 10,004 observations, and the validation set has 1,000 observations.

7. Principal Component Analysis (PCA)

We can use PCA to reduce dimensionality in order to visualize the data in two-dimensional space. In situations like this in which the number of predictor variables is very large, PCA can also be used to reduce dimensionality in order to deal with multicollinearity. As an unsupervised learning technique, PCA can extract patterns from data with only a set of features and no associated response variable. We use **prcomp()** to perform PCA on the training set.

PCA allows us to summarize the original variables with a smaller number of variables that collectively explain most of the variability in the data [1]. The first principal component is a linear combination of the predictors that has the largest variance. Each succeeding principal component has the largest variance out of all linear combinations with the constraint that it is orthogonal to the preceding principal components. In this way, we can use the principal component scores as the predictors instead of the original predictors/pixels.

```
## PVE by first five principal components: 6.82% 5.38% 3.59% 2.78% 2.62%
```

We see that the first principal component explains only 6.8% of the variance in the data, the second principal component explains only 5.4% of the variance, and so forth.

```
## PVE of the first 80 components: 70.62%
```

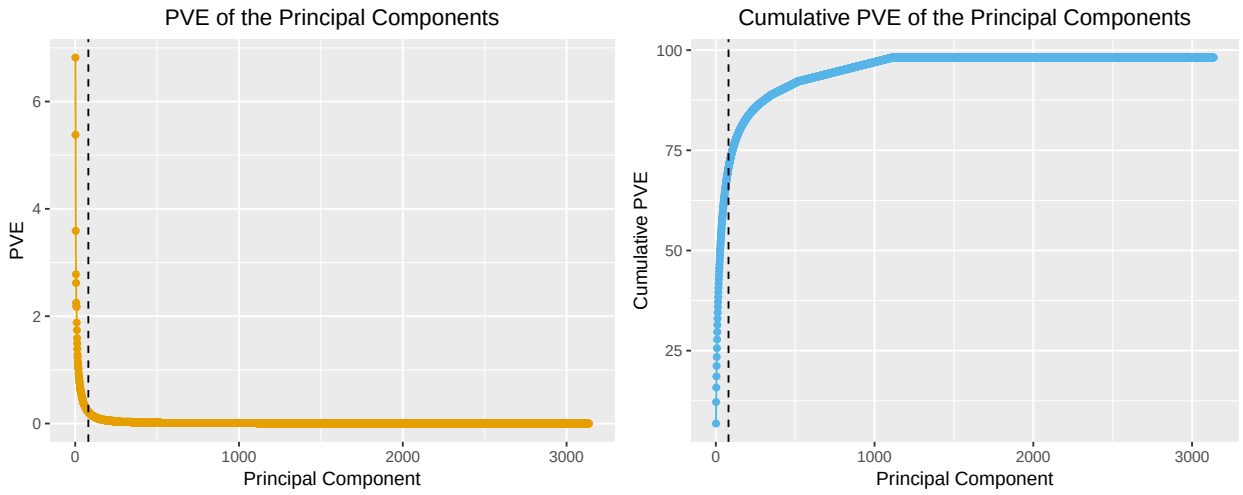

Figure 8: The PVE of the principal components of the training set (left) and the cumulative PVE of the principal components of the training set (right).

Figure 8 shows the proportion of variance explained (PVE) and the cumulative PVE of the principal components of the training set.

The first 80 components explain only about 70% of the variance in the data. However, the left-hand plot of **Figure 8** (scree plot) shows there is an "elbow" after approximately 80 components; that is, there is a marked decrease in the PVE by further components. Thus, we can use the first 80 component scores as the new predictors for the training and validation sets. Compared to the original 3,316 predictors/pixels, we have greatly reduced the dimensionality.

8. Classification Methods

We will fit various classification models to the training set with *Letter* as the response and the first 80 principal component scores as the predictors. We then test and compare these models by evaluating their performance on the validation set (validation set approach). Since the classes in the training set are unbalanced, we will also calculate the test error rates for each character. The models we will consider are the classification tree, random forest, linear discriminant analysis, K-nearest neighbors, and support vector machine. Recall that we are now using the first 80 principal component scores as the predictors for the training and validation sets.

8.1 Classification Tree (rpart)

The classification tree is an easy to explain and easily interpretable method to predict a qualitative response. Tree-based methods involve dividing the predictor space into a number of regions. Each observation is predicted to belong to the most commonly occurring class of training observations in the region to which it belongs. The classification tree is grown through recursive binary splitting, in which the Gini index is used as the criterion for making the binary splits. The Gini index is a measure of node purity – a small value indicates that a node contains mostly observations from a single class [1].

We use **rpart()** from the **rpart** package to construct a classification tree on the training set to predict the character labels. We draw the tree using **rpart.plot()** from the **rpart.plot** package.

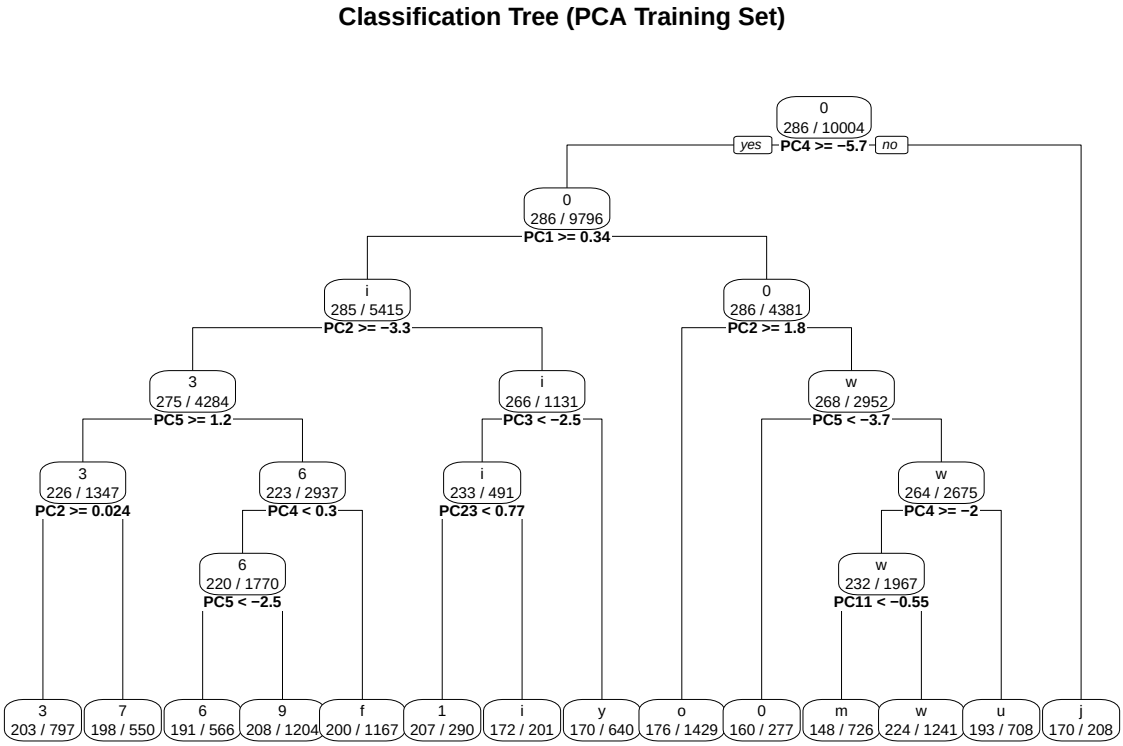

Figure 9: Classification Tree of the training set.

Figure 9 shows the fitted classification tree. We see that there are 11 terminal nodes. The label at the bottom of each internal node indicates the variable being split and the value of the split point. For example,

the split at the top of the tree results in two branches. The left-hand branch corresponds to $PC4 \geq -5.7$, and the right-hand branch corresponds to $PC4 <$ -5.7. The letter in each node is the mode or most commonly occurring class for the observations that fall there. The fraction in each node is the number of correct classifications over the number of observations in the node. For example, in the first terminal node, the mode is the number 3, and there are 203 correct classifications out of 797 observations in that node.

Printing the cp table of the fitted tree produces some summary statistics.

```
##
## Classification tree:
## rpart(formula = Letter ~ ., data = train.pca, method = "class")
##
## Variables actually used in tree construction:
## [1] PC1  PC11 PC2  PC23 PC3  PC4  PC5
##
## Root node error: 9718/10004 = 0.97141
##
## n= 10004
##
##         CP nsplit rel error  xerror      xstd
## 1 0.023410      0   1.00000 1.01050 0.0013829
## 2 0.019088      3   0.92684 0.93692 0.0029434
## 3 0.018419      5   0.88866 0.90626 0.0033404
## 4 0.018008      6   0.87024 0.89957 0.0034172
## 5 0.016293      7   0.85223 0.86211 0.0037972
## 6 0.014561     10   0.80335 0.81632 0.0041701
## 7 0.014406     12   0.77423 0.81148 0.0042046
## 8 0.010000     13   0.75983 0.79656 0.0043060
```

We see that only 7 out of the 80 principal components were used in tree construction. The lowest cross-validation error occurs at 13 splits or 14 terminal nodes, which means we do not need to prune the tree.

```
## VSA test error: 0.911
```

The validation set error rate of the classification tree is 0.911. There is significant room for improvement.

The individual test errors are:

```
##    0    1    2    3    4    5    6    7    8    9    a    b    c    d    e
## 0.90 1.00 1.00 0.80 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00 1.00
##    f    g    h    i    j    k    l    m    n    o    p    q    r    s    t
## 0.83 1.00 1.00 1.00 1.00 1.00 1.00 0.74 1.00 0.18 1.00 1.00 1.00 1.00 1.00
##    u    v    w    y    z
## 0.98 1.00 0.13 1.00 1.00
```

8.2 Random Forest

By aggregating multiple trees, using a method like random forest, we can construct more accurate prediction models. Random forest constructs multiple trees and makes predictions based on the mode or most commonly occurring class of the individual trees.

We use **randomForest()** from the **randomForest** package to perform random forest on the training set. In random forest, only a random subset of the predictors is considered for each split; by default, **randomForest()** uses $m \approx \sqrt{p}$ when building a random forest of classification trees. In this case, $m = 9$ since there

are 80 predictors/principal components. This is in contrast to bagging, which considers all of the predictors for each split. Therefore, if there are strong predictors in the data, all of the trees will look similar to each other, leading to highly correlated predictions. Using a subset of the predictors decorrelates the trees, which makes the average of the resulting trees less variable and hence more reliable [1].

```
## VSA test error: 0.374
```

The validation set error rate of random forest is 0.374. This is a significant improvement over the single classification tree.

The individual test errors are:

```
##    0    1    2    3    4    5    6    7    8    9    a    b    c    d    e
## 0.60 1.00 1.00 1.00 1.00 1.00 1.00 1.00 0.67 1.00 0.15 0.12 0.46 0.42 0.10
##    f    g    h    i    j    k    l    m    n    o    p    q    r    s    t
## 1.00 0.60 0.86 0.23 1.00 1.00 0.48 0.48 0.36 0.38 0.88 0.43 0.14 0.17 0.43
##    u    v    w    y    z
## 0.31 0.38 0.26 1.00 0.13
```

8.3 Linear Discriminant Analysis (LDA)

Linear discriminant analysis models the distributions of the predictors separately for each of the classes, and then it uses Bayes' theorem to estimate the probability of each class [2]. A test observation is assigned to the class with the largest probability. To use LDA, we assume that the observations within each class are drawn from a normal distribution with a class-specific mean vector and a common covariance matrix.
We use **lda()** from the **MASS** package to fit a LDA model to the training set.

```
## VSA test error: 0.34
```

The validation set error rate of LDA is 0.340.

The individual test errors are:

```
##    0    1    2    3    4    5    6    7    8    9    a    b    c    d    e
## 0.50 0.67 1.00 0.80 0.50 0.75 0.80 0.50 0.00 0.80 0.31 0.19 0.00 0.38 0.22
##    f    g    h    i    j    k    l    m    n    o    p    q    r    s    t
## 0.67 0.30 0.38 0.31 0.60 0.76 0.15 0.38 0.36 0.40 0.25 0.29 0.25 0.24 0.48
##    u    v    w    y    z
## 0.24 0.43 0.17 0.45 0.35
```

8.4 K-Nearest Neighbors (KNN)

K-nearest neighbors is a non-parametric classifier. In the KNN algorithm, a test observation is assigned to the class most common among its K nearest neighbors (K closest training observations) in the feature space, where K is a positive integer. For example, with K=1, a test observation is simply assigned to the class of the single closest training observation.

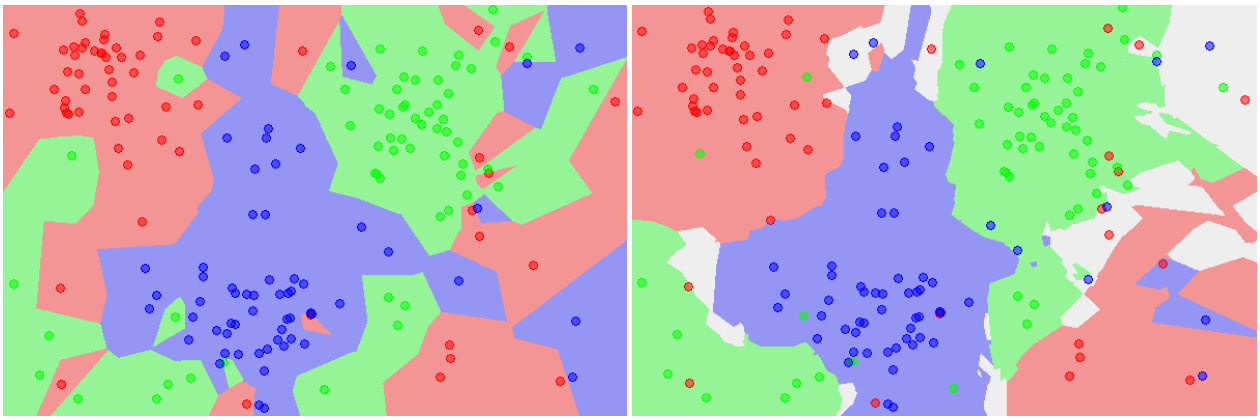

Figure 10: The 1NN (left) and 5NN classification maps (right) [3][4].

Figure 10 illustrates 1NN and 5NN involving three classes (red, green, and blue). Each pixel is classified according to the class of the nearest neighbor (1NN) or the most common class among five nearest neighbors (5NN). White regions correspond to ties, e.g. two green, two red, and one blue among five nearest neighbors.

We use **knn()** from the **class** package to perform KNN on the training set. A simple rule of thumb is to use $K = \sqrt{n}$, where n is the number of training observations [5]. In this case, $K = \sqrt{10000} = 100$. However, we should experiment with different values of K and choose the value that produces the lowest validation error. For a sequence from 2 to 100, we find that the best value of K is 3.

```
## Best value of K: 3
```

```
## VSA test error: 0.237
```

The validation set error rate of KNN (K=3) is 0.237.

The individual test errors are:

```
##    0    1    2    3    4    5    6    7    8    9    a    b    c    d    e
## 0.30 0.89 0.67 0.20 0.50 0.25 0.60 0.33 0.33 0.60 0.17 0.41 0.08 0.29 0.20
##    f    g    h    i    j    k    l    m    n    o    p    q    r    s    t
## 0.58 0.20 0.38 0.05 0.40 0.35 0.02 0.31 0.22 0.32 0.38 0.29 0.33 0.28 0.05
##    u    v    w    y    z
## 0.14 0.15 0.09 0.36 0.26
```

8.5 Support Vector Machine (SVM)

A support vector machine is a classifier defined by a separating hyperplane. In two-dimensional space, this hyperplane is a line that separates the classes. Given a labeled training set, a SVM constructs an optimal hyperplane that creates the largest separation, or margin, between the classes. A test observation is then classified based on which side of the hyperplane it lies.

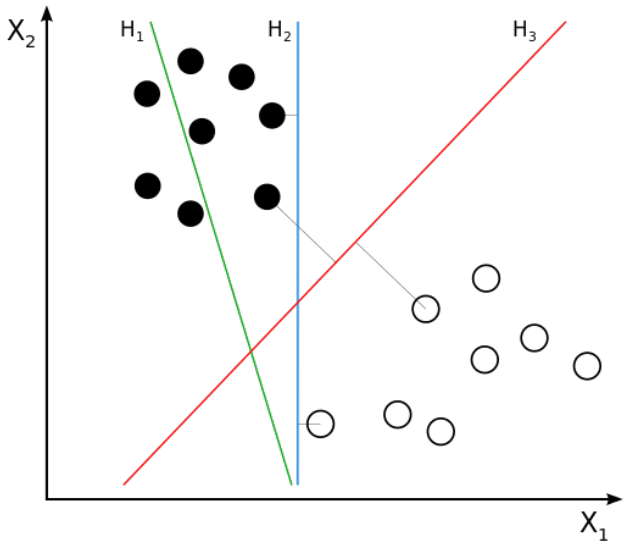

Figure 11: Example of how a SVM would choose a separating hyperplane [6].

Figure 11 shows how a SVM would choose a separating hyperplane for two classes of observations (black and white circles). H_1 does not separate the classes. H_2 separates the classes but does not create the largest margin. H_3 separates the classes with the largest margin and hence is the maximal margin hyperplane.

We fit a support vector classifier (linear SVM) to the training set with various values of cost (0.001, 0.01, 0.1, 1, 5, 10, 100). We use **tune()** from the **e1071** library to perform 10-fold cross-validation on the set of models under consideration. To save time, we use a random 10% of the training set for tuning. The cross-validation errors for these models can be accessed using **summary()** (not shown due to computation time).

We find that cost=0.1 results in the lowest CV error rate at 0.344.

Next, we fit an SVM with radial basis kernels with various values of cost (1, 5, 10, 50, 100, 125, 150) and gamma (0.001, 0.01, 0.1, 1). We use **tune()** to perform 10-fold cross-validation on the set of models under consideration. The cross-validation errors for these models can be accessed using **summary()** (not shown due to computation time).

We find that cost=5 and gamma=0.01 result in the lowest CV error rate at 0.301.

Finally, we fit an SVM with polynomial basis kernels with various values of cost (1, 5, 10, 50, 100, 125, 150) and degree (2, 3, 4, 5). The cross-validation errors for these models can be accessed using **summary()** (not shown due to computation time).

We find that cost=50 and degree=2 result in the lowest CV error rate at 0.402.

Among the SVM's, the SVM with radial basis kernels performed the best; hence, we will use that model to compute the validation set test error. The summary of the fitted radial SVM using cost=5 and gamma=0.01 is shown below:

```
##
## Call:
## svm(formula = Letter ~ ., data = train.pca, kernel = "radial",
##     cost = 5, gamma = 0.01)
##
##
## Parameters:
##    SVM-Type:  C-classification
```

```
##  SVM-Kernel:  radial
##        cost:  5
##       gamma:  0.01
##
## Number of Support Vectors:  6686
##
##  ( 260 274 258 151 278 274 257 270 255 230 271 259 118 102 110 140 101 101 115 119 156 139 267 143 1
##
##
## Number of Classes:  35
##
## Levels:
##  0 1 2 3 4 5 6 7 8 9 a b c d e f g h i j k l m n o p q r s t u v w y z

## VSA test error: 0.216
```

The validation set error rate of the SVM with radial basis kernels (cost=5, gamma=0.01) is 0.216.

The individual test errors are:

```
##    0    1    2    3    4    5    6    7    8    9    a    b    c    d    e
## 0.40 0.67 1.00 0.80 1.00 1.00 0.80 0.33 0.67 1.00 0.12 0.03 0.31 0.17 0.07
##    f    g    h    i    j    k    l    m    n    o    p    q    r    s    t
## 0.75 0.30 0.38 0.10 1.00 0.41 0.25 0.21 0.16 0.26 0.75 0.29 0.08 0.07 0.21
##    u    v    w    y    z
## 0.15 0.26 0.13 0.27 0.22
```

8.6 Summary of Classification Methods

The test error rate estimates from the validation set approach (VSA) for the various classification methods are compared in **Table 3**.

Table 3: Test Error Rate Comparison for the Classification Methods

Method	VSA
Classification Tree	0.911
Random Forest	0.374
LDA	0.34
KNN (K=3)	0.237
Radial SVM (cost=5, gamma=0.01)	0.216

Both KNN and SVM with radial basis kernels performed well. SVM had the lowest validation set test error, but KNN had lower individual test errors for the sparse characters such as the digits. On the other hand, the single classification tree by far performed the worst. Thus, we recommend KNN if the individual test errors are important; otherwise, if a lower overall error rate is the priority, we recommend SVM. Since the classes in the training set are heavily unbalanced, the overall error rate is not a good performance measure; thus, we recommend KNN for most circumstances.

9. Conclusion

In this project, we were given a labeled dataset with 1,000 labeled handwritten characters and an unlabeled dataset with 10,000 unlabeled handwritten characters. We manually labeled the unlabeled data based on human judgment. In most cases, 0's/o's and 1's/i's are impossible to tell apart, so we labeled these as "0 or o" and "1 or i" respectively. Then, we used random forest to try to separate the 0's from the o's and the 1's from the i's. To deal with the heavily unbalanced classes in the unlabeled data, we applied oversampling to the minority classes and undersampling to the majority classes.

We then used the unlabeled data (after labeling and applying oversampling/undersampling) as the training set and the labeled data as the validation set to get an unbiased estimate of the test error. Based on the proportion of variance explained (PVE) or "scree" plot, we chose 80 as the number of principal components to use; thus, we have reduced the dimensionality from 3,316 predictors or pixels to 80 predictors or principal component scores.

With the first 80 principal component scores as the new predictors, we fit the following models on the training set: classification tree, random forest, LDA, KNN, and support vector machines (linear, radial, and polynomial kernels). We then used the validation set approach to estimate the test error rate of each model. For each model, we also calculated the test error rate of each character. It was found that KNN with K = 3 performed the best in terms of the individual test error rates, and SVM with radial basis kernels (cost=5, gamma=0.01) performed the best in terms of the overall error rate. Since the classes in the training set are unbalanced, the overall error rate is not a good performance measure; thus, we recommend KNN as the best model.

Finally, we will use our chosen classifier (KNN with K = 3) to predict the labels of the 30,000 unlabeled characters. However, this is just the first step in constructing an automated handwriting recognition system. For future work, we could focus on improving discernibility between "0" and "O", "1" and "I", "5" and "S", etc. Since the characters in the data are heavily unbalanced, we can improve our results by obtaining more examples for the sparse characters such as the digits. And lastly, we should explore semi-supervised learning techniques, which make use of both labeled and unlabeled data.

References

[1] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An Introduction to Statistical Learning: with Applications in R.* New York, NY: Springer.

[2] Peixeiro, M. (2018, December 11). *Classification – Linear Discriminant Analysis.* Retrieved from https://towardsdatascience.com/classification-part-2-linear-discriminant-analysis-ea60c45b9ee5

[3] By Agor153 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24350615

[4] By Agor153 - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24350617

[5] Thirumuruganathan, S. (2010, May 17). A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithm [Blog post]. Retrieved from https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/

[6] User:ZackWeinberg, based on PNG version by User:Cyc [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)]

[7] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). *Smote: Synthetic minority over-sampling technique.* Journal of Artificial Intelligence Research, 16:321-357.