Supervised Learning - Assignment 2

Dean Hope Robertson (RBRDEA003)

4/23/2018

Abstract

In this assignment students are tasked with building a classification model that is able to recognize weather a handwritten digit is an odd or even number. The students are required to download the training dataset **Train_Digits_20180302.csv.**, which contains approximately **2500** handwritten digits on which to train the classification model.

The students tasked with using their new found skills in the art of statistical learning, in order to train a scheme that best classifies the handwritten digits using a various methods covered in class. The trained classification scheme is then implemented on the test dataset, after which the classified results are then submitted for grading.

Introduction

The training dataset is structured as such, the first column holds the true value of the handwritten digit and the remaining columns from 2 to 785 represents the pixels that create a 28 by 28 pixelated image of a handwritten number.Each pixel holds a value that represents the grey scale value of the pixel, ranging from 0 for white pixels and 255 for black pixels. Each pixel holds a value that represents the grey scale value of the pixel, ranging from 0 for white pixels and 255 for black pixels. The following table shows the distribution of handwritten digits in the training dataset:

Table1 : Distribution of digits in training dataset.

Digit	Odd/Even	Count
9	Odd	261
8	Even	247
7	Odd	273
6	Even	247
5	Odd	232
4	Even	275
3	Odd	241
2	Even	243
1	Odd	244
0	Odd	237

As show in the table above, there is an even distribution of digits across the training data, with a total of 1249 **even** numbers and 1251 **odd** numbers. The true values of the digits are first converted into a odd and even binary dataset, 1' representing even numbers and 0's presenting the odd numbers. For this particular dataset, the response variable is qualitative categorical variable; defining this as a two-type classification problem.

The objective is to train and test a variety of classification models and compare the accuracy of each model and determine the most accurate classification model to predict the values of the test dataset. In order to train and test the classification models comprehensively, the training data is split 80:20 into subset training and test datasets.

Assessing Model Accuracy

The response variables for a classification scheme are not numerical, and thus very different to a regression problem. The most commonly used method of assessing the accuracy of the classification model is *error rate* given by the formula below:

$$ErrorRate = \frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y_i})$$

If the predicted response variable $\hat{y_i}$ is equal to the true response, the observation is classified correctly. When classification is correct, $I(y_i \neq \hat{y_i})$ is equal to zero, and when misclassified is equal to 1. An accurate classification scheme is one that returns a smallest error rate. In this assignment the classification rate is used to compare the performance of the classification models. **Classification rate** is defined as 1-error rate.

K-Nearest Neighbours

K - Nearest Neighbours (KNN) is one the most reputable methods when it comes to classification problems, as it attempts to implement conditional probability. The KKN algorithm attempts to classify an observation to a particular class with the highest probability, which is estimated by looking the classification of the other observations and its proximity to them.

$$Pr(Y = j)|X = x_0) = \frac{1}{K}\sum_{i \in N_o} I(y_i = y_i)$$

For the handwritten digit problem, the KNN classifier reviews the test observation (x_0) and identifies the K number of digits in the training data that are the closest to the test observation. The algorithm then estimates the condition probability of the class of that digit (odd or even) as a fraction of the K points closest to the observation whose class is odd or even (j).KNN then classifies that observation (x_0) to the class with the largest probability.

Determining K

The algorithm is greatly dependent on tuning parameter, K, as this directly corresponds to the number of 'neighbours' the algorithm takes into account when estimating the class probability.

When k=1, the algorithm will only take into account the single closet observation, which increases the flexibility of the model. This will result in a classifier with a low bias and a high variance. On the contrary, when scaling up the value of K to k=10, KNN will identify the closest 10 observations to the test observation, and estimate the class probability using the classes of all 10 points. Here the model becomes less flexible and reduces the risk of overfitting the training data at the price of a more bias classifier.

For the reasons above, it crucial that a variety of K-values are explored in order to return a model that will not only fit the training data accurately, but also be able to generalize and adapt to the test dataset. The K-nearest neighbour's classification can be utilized through the *class* or *fnn* packages in R, and the method is called using then following script :

```r
#apply K-Nearest Neighbours to the train and test data set
knn.pred = knn(train_data, test_data,train_resp, k=5, prob=TRUE)
```

In the above example, a randomly selected value of **k=** produces a model that returns a classification rate of **0.96** and **0.932** for training and test datset respectively.Ten-fold cross validation is then deployed to determine the optimal number for K, as can been in Figure 4 in Appendix A.Cross validation shows that k=3 results in the lowest error rate for training data. This is a reasonable value for K, for if K was equal to 1 has a risk of over fitting the training data. The final K-nearest neighbour's test results are as follows:

Table 2 : Confusion matrix for K-nearest neighbour classification

Predicted	Odd	Even
Odd	251	23
Even	7	219

$$CR(train) = 97.25\%$$
$$CR(test) = 94.00\%$$

Decision Trees

Tree-based methods used for classification involve segmenting the predictors in an effort to separate the space into regions of classification. Tree-methods are favoured for classification problems due to the fact that they are relatively easy to interpret, and handle qualitative variables well without the need to create dummy variables.

The tree-growing algorithm uses a *top-down, greedy* approach that is known as *recursive binary splitting* (James, Witten, Hastie and Tibshirani, 2013). The process begins at the top of the tree and spits up the predictor space in successive steps, dividing into to two new branches each time. The logic is referred to as greedy, seeing as it is very short-sighted in the sense that it evaluates what the 'best split' is at a particular step; rather than forecasting and picking a split that will results in a better classification in the future steps.

Regression trees determine the 'best split' based on a reduction in **RSS**, whilst classification trees utilizes alternative splitting criterion, such as the **Gini Index** and **Entropy**. The Gini Index is defined as seen below:

$$G = \sum_{K=1}^{K} \hat{p_{mk}}(1 - \hat{p_{mk}})$$

The Gini index measure variability of classes in a particular region, thus if the Gini index return a small value, then this means one class is favoured. The index can be referred to as the measure of node purity. Similarly, Entropy is also used to measure node purity and used to evaluate the quality of each split along the tree. *tree* function in the *tree* library is used to grow the intial tree which can be found in Appendix A, Figure 5.

Pruning the Tree

Growing the largest possible tree may produce good predictions on the training dataset, but is likely to overfit the training data and result in a model that does not generalize well, thus producing poor test results. On the other hand, a smaller tree could potentially lead to a more interpretable model with lower variance at the cost of negligible bias. A common approach is to grow a classification tree to the largest possible state, and then prune the tree back using *cost complexity pruning* to attain a subtree.

$$\sum_{K=1}^{|T|} \sum_{x_i \in R_m} (y_i - \hat{y_{rm}})^2 + \alpha|T|$$

Cost complexity pruning works in a similar way to **LASSO** where a **penalty** is added to the model which penalizes the complexity. The penalty consists of a tuning parameter multiplied by the absolute value of terminal nodes in the tree, which forces the tree to be small and avoid overfitting. In this way, the penalty creates a trade-off between optimal fit and the complexity of the tree. The tuning parameter is determined through cross–validation called by the following Rcode:

```r
#default is 10-fold cross validation
cv_tree = cv.tree(treefit, FUN = prune.misclass)
```

The results for cross validation can be found in Appendix A, Figure 6. Cross-valdation indicates that a optimal number of terminal nodes is **15**. The tree is then prune using the built-in *prune* function in the *tree* package. The results of the final pruned decision tree are as follows:

Table 3 : Confusion matrix for Decision Tree classification

Predicted	Odd	Even
Odd	210	35
Even	48	217

$$CR(train) = 88.75\%$$

$$CR(test) = 83.40\%$$

Bagging, Boosting, and Random Forests

While decision trees are easily interpretable, they are not as accurate in classification when compared to other classification methods. This is due to the fact that trees suffer from high variance and, a slight alteration in the data can have a significant effect on the structure of the tree. It is for this reason that methods such as bagging, boosting and random forests we devised, to reduce the variance of decision trees and improve their prediction performance.

Bagging

$$\hat{f_{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f^{*b}}(x)$$

Bagging makes use of bootstrapping by repeatedly taking samples from the training dataset and builds an individual decision tree for each training set. The algorithm then takes the average prediction of B number of trees, in order to obtain an estimated response. The bootstrapped trees are not pruned at all and have a high variance/ low bias. By taking the average of the bootstrapped trees, this greatly reduces the variance of the model.

The number of trees B is not a critical parameter; however a sufficiently sized B value results in a stabilized error rate. The default setting for n.trees of 500 was applied with an mtry = 784. The final bagged model's classification rates for the training and test dataset shown below:

Table 4 : Confusion matrix for Bagging model classification

Predicted	Odd	Even
Odd	227	25
Even	31	217

$$CR(train) = 100.00\%$$
$$CR(test) = 88.80\%$$

Random Forests

The drawback with bagged trees is that the bootstrapped trees are very correlated and lack variety. In essence, taking the average predictions from very correlated trees does not substantially reduce the variance of the model. **Random forests** attempt to address this problem by producing decorrelated trees from which to average from. In random forests, the algorithm does not consider every possible variable at each split, like produced in bagging; but instead only considers a random selection of m variables such as that $m < p$.

Typically m is chosen to be the sqrt root of the total number of predictors. While it might seem unorthodox to discard majority of the predictors in when building a decision tree; it allows other predictors (less prominent) to be considered at each split and thus create a diversified subtrees. Similar to bagging, there is no risk of overfitting when using random forests, therefore the selected *n.tree*=1000 is substantially large number and has little risk in resulting in an overfitting model.

The first interation of the random forest model returns a training and test classification rate of 1 and 0.918 respectively.The **tuneRF** function is then utilized to determine the optimal number for m, which can be found in Figure 7 in Appendix A.The optimal number for m is 13 variables as can be seen in the the figure below.

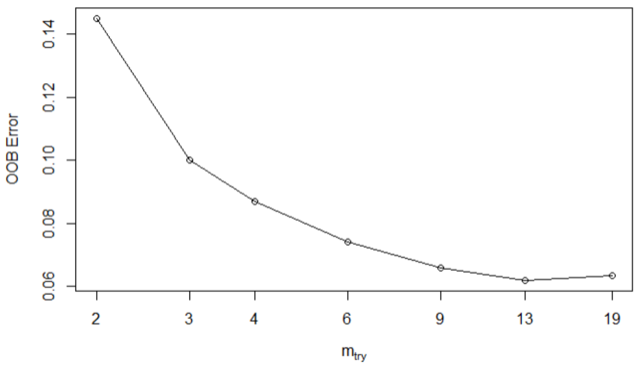

Figure 1: : Cross-Validation for Decision Tree

The final random forest classification rates for the training and test dataset shown below:

Table 5 : Confusion matrix for Random Forest classification

Predicted	Odd	Even
Odd	238	24
Even	20	218

$$CR(train) = 100.00\%$$

$$CR(test) = 91.20\%$$

Boosted Tree

The fundamental difference between **boosting** and the two methods mentioned above, is that instead of bootstrapping and combining all the separate decision trees, boosting grows each tree sequentially. Boosting does not grow trees independently of each other, but rather each tree is grown using the information from the previous trees in an additive sequence. Here, the boosting model learns slowly and does not make use of bootstrapping. The additive approach looks at the reducing the residuals by adding smaller trees into the current model to slowly improve the fit.

There are a number of tuning hyper-parameters involved in this process:

- Number of trees (*n.tree*)
- Shrinkage paramters (λ)
- Number of splits (d)

Unlike bagging and random forests, it is possible to over fit the data using too many tree. The shrinkage parameter is essentially the learning rate of the boosting algorithm, and us typically a small value. The number of splits (d) controls the complexity of the model, also known as interaction depth. Ridgeway (2017, p.7) states that for a type-two classification, the *Bernoulli distribution* is the distribution that should be used. The shrinkage paramaters is the essentially the learning rate of the algorithm and should be a relatively small values.

The initial boosted model using a d=4, λ = 0.01 and *n.tree*=2000 returned a training and test classification rate of **0.991** and **0.924**, respectively. The gbm package function *gbm.perf*, shows the optimal number of iterations to utilize for boosting, reducing the risk of overfitting as can be seen below:

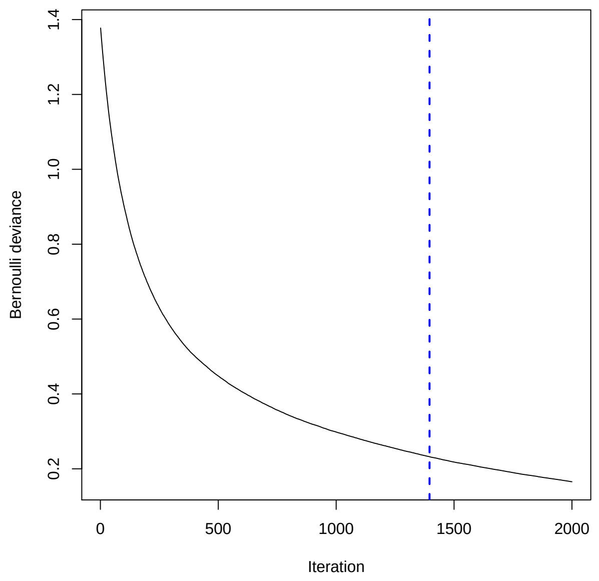

Figure 2: : OOB vs Number of Iterations (Boosting)

The above figure shows the relationship between the number of trees and the *out-of-bag error (OOB)*. In this case the optimal number of trees (B) is *1397*. Once obtaining the optimal number of trees, it is now possible to train the boosted model and determine the classification rates for the training and test dataset shown below:

Table 6 : Confusion matrix for Boosted model classification

Predicted	Odd	Even
Odd	235	23
Even	23	219

$$CR(train) = 98.00\%$$

$$CR(test) = 90.80\%$$

Support Vector Machines

Support vector classifiers attempts to find a hyperplane that separates the observations into their respective classes. The dimensions of the hyperplane are defined by the number of parameters in the space such that (p-1) = dimensions of the subspace. Of course not all problems can be solved linearly and support vector classifiers look to utilize quadratic and cubic terms to solve non-linearly separable observations. However given a large feature space one could end up with a huge number of polynomial features that can become very complex and computational inefficient.

To get around this, support vector classifiers expand into the use of **kernels**. The kernel function computes the inner products of pair observations in an effort to quantify the similarity of the pair of observations. Essentially this allows the support vector classifier to fit a higher dimension space (a sphere), and then project the classifications back to the original feature space. Kernels can do this for very large dimensional spaces which are perfect for this particular problem, seeing as there are approximately 784 predictor variables. **A support Vector** a machine is the combination of support vector classifier with non-linear kernels (e.g. radial) and takes the form:

$$f(x) = \beta o + \sum_{i \in S} \alpha i K(x, x_i)$$

The **e1071** package in R is the preferred library to utilize the support vector machine classification methodology. Using all 284 predictors for support vector machines can be very computational expensive especially taking into account the number of variables (pixels) which are equa to zero (see Table 1). Dimensionality reduction is required in the form of **principle component analysis** in order to decrease the number of variables in the dataset. This is done by executing the following R code:

```r
#need to peform Principle Compenent Analysis (PCA) on variables
pca_train = prcomp(train_set, scale=FALSE, center = T)
rotate<-pca_train$rotation[1:57]
pca_train<-as.matrix(scale(train_set,center = TRUE, scale = FALSE))%*%(rotate)
```

A variance vs no.components graph, figure 7 in Appendix A, shows how the percentage of variance explained changes with the total number of principle components. One can deduce that 85% of the variance can be explained with the first **57** principle components. It is on this basis that the optimal number of principle components is selected. The Support Vector Machine algorithm is then test on the training and test dataset with using a variety of kernels and default parameters selected. The default parameter **gamma** = 1/no. dimensions and **cost** = 1. The training and test data were as follows:

Table 7 : Confusion matrix for Support Vector Machine classification

Kernel	Train	Test
Linear	86.4%	82.8%
Poly-2	99.3%	95.0%
Poly-3	99.6%	93.2%
Radial	99.1%	94.6%
Sig	77.6%	77.4%

Based on the information above, the **polynomial** kernel with **degree 2** returned second highest training classification rate as well as the highest test classification rate. This shows that the polynomial vector machine was able to general and adapt the test data better than the other vector machines. Once the correct kernel is selected cross validation is used to determine the vector machine parameter C and coefO. This is done using the built-in tune function that utilizes 10-fold cross validation on the training set. The cross validation returns:

$C = 1$
$coefO = 0$

The polynomial support vector machine with a degree of two returns and cost =1 retruns the following results:

$$CR(train) = 99.3.00\%$$
$$CR(test) = 95.00\%$$

Neural Networks

A **Neural network** is a non-linear model that can be scaled up in a variety of ways, and was inspired by the human brain. Neural network models are capable of solving extremely advanced problems that push outside the realm of statistics. As can be seen in the figure below, the structure of neural network usually consists of a number of layers (multilayer NN); an input layer, a number of hidden layers and an output layer. There are two fundamentally different types of neural networks

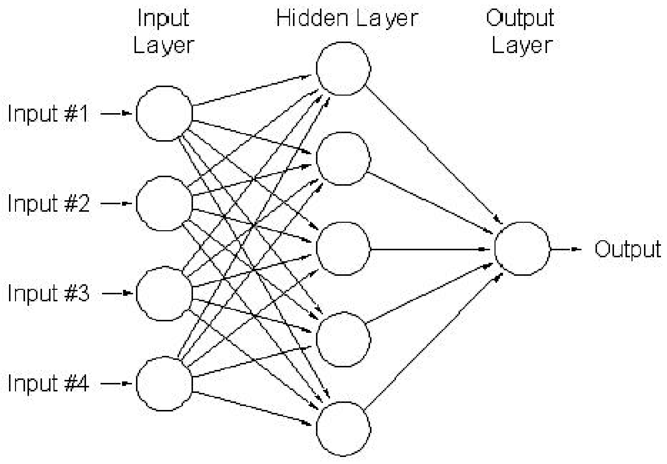

Figure 3: : Standard Neural Network,(www.expert.com)

Feedforward: A feedforward network only allows signals to move in one direction, from the input layer to the output layer. The hidden layers perform calculations, which then become the inputs for the preceding layers until finally reaching the output layer.

Feedback: A feedback network allows signals to travel in both directions via feedback loops. These feedback loops allow for calculations in multiple layers to be fed back into the network, creating a very dynamic and continuously changing fluid environment. The network will remain in this fluid state until it reaches an equilibrium point. Feedback networks mimic a sense of memory and continuous learning.

Network Architecture

Since the assignment focuses on supervised learning, the feedforward architecture is deployed to determine the neural network of best fit. The feedforward network will have an **input layer** of 784 neurons equal to the number of pixels (predictors), as well a number of **hidden layers**, and a **output layer**.

First layer: Each neuron has 784 parameter vectors called weights, meaning each predictor variable will have a corresponding weight at each individual neuron in the first of the hidden layers. The weights are multiplied with the predictor values (also known as activators) to compute a weighted sum for each pixel, at each neuron. The weighted sums are then processed by a non-linear activation function (generally a sigmoid function), to force the weighted sum into the range between 0 and 1. Learning takes place when weights are adapted to minimize the classification error in the training data set.

Hyper Parameters

Activation - the activation function, mentioned in the paragraph above, helps calculate the weighted sum of neurons and decide if a particular neuron should react if a particular variable is present. The activation functions can be linear or non-linear. Non-linear functions are more commonly used as the functions make the model more flexible and able to generalize better.

Nfolds - is the number of folds used in k-fold cross validation and is set to 10, which is the standard for k-fold cross validation throughout the assignment.

Epochs - specifies the number of iteration performed on the training dataset.

Distribution - specifies the distribution of the response.

Hidden - specifies the number and size of each hidden layer in the model.

Dropout ratio - The input and hidden dropout ratios declare the faction of the hidden/input to omit for the training process to increase generalization. In other words a portion of the input data is exclude intentionally. The default is set to 0.

Sparse - this parameter enables light data handling. This should be enabling for dataset with lots of zeros, like the handwritten number training dataset.

The choosen model is a single layer 300 neuron feedforward neural network using dropout activcation and a multinomial distribution. The number of neurons should not exceed you the number of predictor variables. Varying values for epochs and dropout ratios were explored before settling on an epoch of 21 iterations, and dropout ratio of 0.2. The results are as follows:

Table 8 : Confusion matrix for Neaural Network classification

Predicted	Odd	Even
Odd	242	12
Even	16	230

$$CR(train) = 99.8\%$$

$$CR(test) = 94.4\%$$

Conclusion

As stated in the beginning of the assignment, *classification rate* (CR) would be used as the benchmark for evaluating a model's performance. Throughout this assignment, seven models were trained, tested using a random 80/20 training to test dataset split. The results are as shown below:

Model	Train CR	Test CR
K-Nearest Neighbours	97.2%	94.0%
Discision Tree	88.75%	83.4%
Bagging	100%	88.8%
Random Forest	100%	91.2%
Boosting	98%	90.8%
Support Vector Machine	99.3%	95%
Neural Network	99.8%	94.4%

Based on the information above, the model with the lowest training *classification rate* is the standard decision tree. The highest training *classification rate* is secured by the bagging and random forest models, however the weaker test results indicate that the models may be over fitting to the training data and not able to generalize. Therefore the best preforming model is the single layer neural network withe a training *classification rate* of 99.8%. The neural network model is also able to generalize, seeing as the it managed to reproduce stellar results on the test dataset.

References

- Ridgeway G.(2017). Package 'gbm', Version 2.1.3, CRAN Repository,p.7
- James G, Witten D, Hastie T, and Tibshirani R.(2013), An Introduction to Statistical Learning, New York, Springer. p306
- Build Neural Network Model With MS Excel. Available at: www.xlpert.com/nn_Solve.htm [Accessed 21 Apr. 2018].

Appendix A

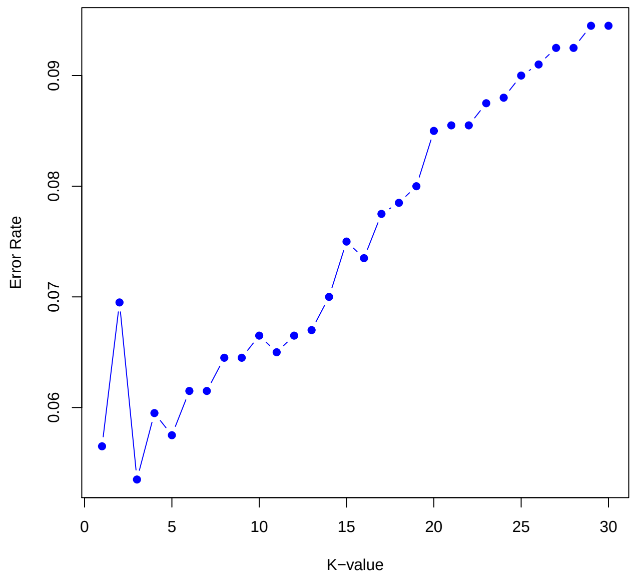

Figure 4: : Cross-Validation for K - Nearest Neighbours

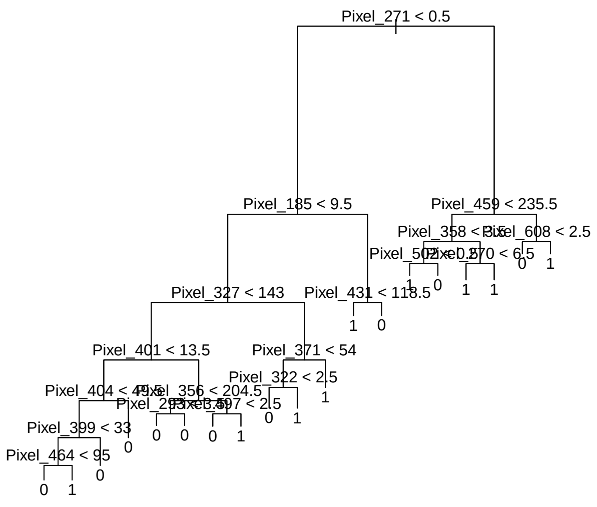

Figure 5: : Decision Tree - Initial

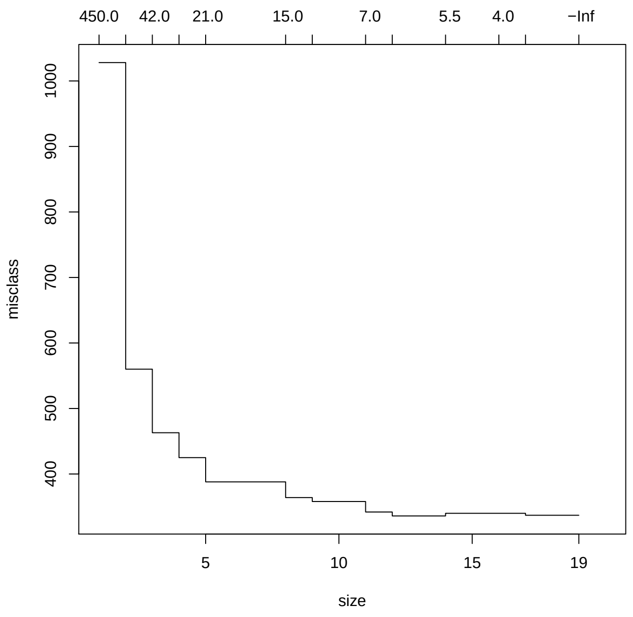

Figure 6: : Cross-Validation for Decision Tree

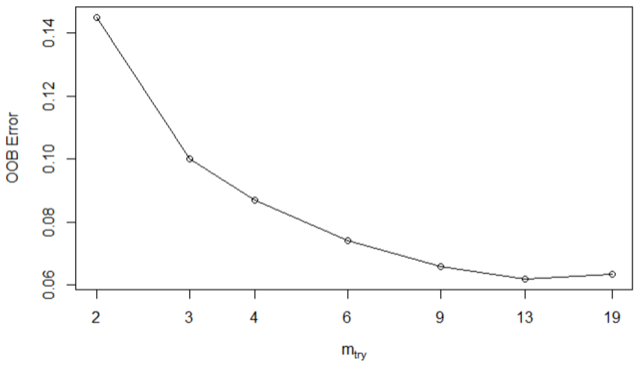

Figure 7: : Cross-Validation for Decision Tree

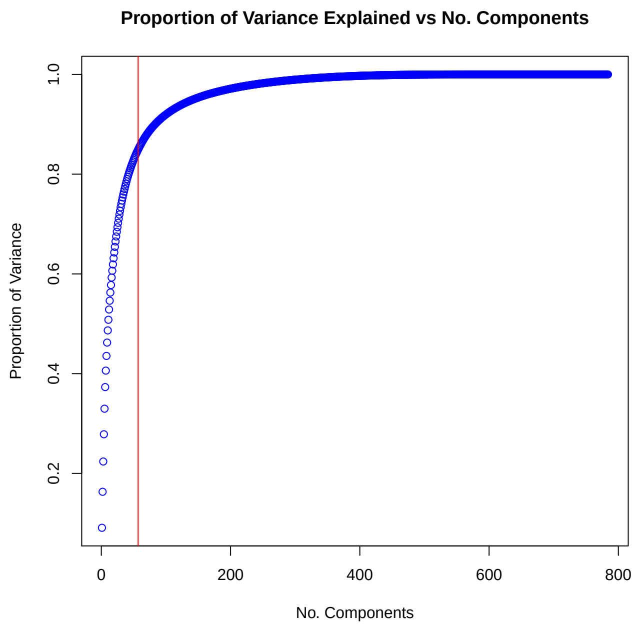

Figure 8: : Proportion of Variance Explained vs No. Principle Components

Appendix B

Introduction

```
rm(list = ls())
library(tibble)

#import training data
data = read.csv("Train_Digits_20180302.csv", header = T)
train<-as.matrix(data)

#replace digits with binary even/odd (1/0) + chnage colname
data$Digit = ifelse(data$Digit %% 2 == 0,1,0)
names(data)[1]<-paste("binary")

# Create a 28*28 matrix
m = matrix(unlist(train[1,-1]), nrow = 28, byrow = TRUE)
# reverses (rotates the matrix)
rotate <- function(x)t(apply(x, 1, rev))
# Show the digit in black and white
image(rotate(m),col= c("white", "black"))

# Plot some of images
par(mfrow=c(2,3))
lapply(1:6,
       function(x) image(
         rotate(matrix(unlist(train[x,-1]),nrow = 28, byrow = TRUE)),
         col=c("white","black"),
         xlab=train[x,1]
```

```
        )
)
```

K-Nearest Neighbour

```
rm(list = ls())
library(nnet)
library(tibble)
require(class)
require(e1071)

#import training data
data = read.csv("Train_Digits_20180302.csv", header = T)

#replace digits with binary even/odd (1/0) + chnage colname
data$Digit = ifelse(data$Digit %% 2 == 0,1,0)
names(data)[1]<-paste("binary")

train = 1:2000

#train and test variables for KNN
train_data = data[train, 2:785]#80%
test_data = data[-train,2:785] #20%

#train and test response for KNN
train_resp = data[train,]$binary
test_resp = data[-train,]$binary

#apply K-Nearest Neighbours to the train and test data set
knn.pred = knn(train_data, train_data,train_resp, k=5, prob=TRUE)

#(TRAIN)
#quantify the score
mean(knn.pred==train_resp)

#(TEST)
knn.pred = knn(train_data, test_data,train_resp, k=5, prob=TRUE)
mean(knn.pred==test_resp)

# tune knn to find optimal k value
knn_cross <- tune.knn(x = data[train, 2:785],y = as.factor(data[train,1]),k = 1:30,tunecontrol = tune.c

#(TEST)
knn_cross$performances

matplot(knn_cross$performances[,1],knn_cross$performances[,2], pch = 19, col = "blue", type = "b", ylab

#apply K-Nearest Neighbours to the train and test data set
knn.pred = knn(train_data, train_data,train_resp, k=3, prob=TRUE)
mean(knn.pred==train_resp)

#Select new kvalue of k=3
```

```
knn.pred = knn(train_data, test_data,train_resp, k=3, prob=TRUE)
mean(knn.pred==test_resp)

#Show the table of results
table(knn.pred,test_resp)
```

Decision Trees

```
rm(list = ls())
require(tree)
#import training data
data = read.csv("Train_Digits_20180302.csv", header = T)

#replace digits with binary even/odd (1/0) + chnage colname
data$Digit = ifelse(data$Digit %% 2 == 0,1,0)
names(data)[1]<-paste("binary")

train = 1:2000

train_tree = data[train,]
test_tree = data[-train,]

#check if response variable is a factor variable
is.factor(train_tree$binary)

#declare the response variable as a "factor" vraiable
train_tree$binary = as.factor(train_tree$binary)
test_tree$binary = as.factor(test_tree$binary)

#apply the decision tree alogrithm
treefit= tree(binary~., data = train_tree)

#(TRAIN)
tree.pred = predict(treefit, train_tree, type = "class")
mean(tree.pred==train_tree$binary)

#(TESt)
tree.pred = predict(treefit, test_tree, type = "class")
mean(tree.pred==test_tree$binary)

#plot the training tree
plot(treefit);text(treefit, pretty=0)
```

Pruning the Tree

```
#Cross Validation
#default is 10-fold cross validation
cv_tree = cv.tree(treefit, FUN = prune.misclass)

#plot the missclasses vs number of terminal nodes
#classes drop significantly down to 15 and then begins to rise -  optimal size 15
```

```
pdf('CV_Tree.pdf')
plot(cv_tree,xlim=c(1,20))
dev.off()

plot(cv_tree$size ,cv_tree$dev, xlim=c(0,20), type="b", xlab="# of terminal nodes", ylab="CV (misclassi

#prune the tree on the full training data
prune.tree = prune.misclass(treefit, best =15)

#Plot the new scaled down tree
plot(prune.tree); text(prune.tree, pretty=0)

#predict with new pruned tree
tree.pred_2 = predict(prune.tree, test_tree, type = "class")

#(TRAIN)
tree.pred_2 = predict(prune.tree, train_tree, type = "class")
mean(tree.pred==train_tree$binary)

#(TEST)
#predict with new pruned tree
tree.pred_2 = predict(prune.tree, test_tree, type = "class")
mean(tree.pred_2==test_tree$binary)

#Show the table of results
table(tree.pred_2,test_tree$binary)
```

Bagging

```
#import the randome forest package
require(randomForest)
data = read.csv("Train_Digits_20180302.csv", header = T)
train = 1:2000

data$Digit = ifelse(data$Digit %% 2 == 0,1,0)
names(data)[1]<-paste("binary")

train_tree = data[train, 2:785]
test_tree = data[-train, 2:785]

train_rf = data[train,]
train_rf$binary = as.factor(train_rf$binary)

response <- as.factor(data[train,1])
response2 <- as.factor(data[-train,1])

#baggging -- when the number of variables at each split is equal to the mac number of varibles
rf_fit = randomForest(binary~. , data =train_rf, mtry=784)

#(TRAINING)
rf_pred = predict(rf_fit,train_tree, type = "class")
mean(rf_pred==response)
```

```r
#(TEST)
rf_pred = predict(rf_fit,test_tree, type = "class")
mean(rf_pred==response2)
```

Random Forest

```r
rm(list = ls())
#import the randome forest package
require(randomForest)
data = read.csv("Train_Digits_20180302.csv", header = T)
train = 1:2000

data$Digit = ifelse(data$Digit %% 2 == 0,1,0)
names(data)[1]<-paste("binary")

train_tree = data[train, 2:785]
test_tree = data[-train, 2:785]

train_rf = data[train,]
train_rf$binary = as.factor(train_rf$binary)

response <- as.factor(data[train,1])
response2 <- as.factor(data[-train,1])

#randomforests with 200 trees and mtry= sqrt(784) -- 28
rf_fit = randomForest(binary~. , data =train_rf, ntree=1000)

#(TRAINING)
rf_pred = predict(rf_fit,train_tree, type = "class")
mean(rf_pred==response)

#(TEST)
rf_pred = predict(rf_fit,test_tree, type = "class")
mean(rf_pred==response2)

# for(mtry in 1:28){
#   fit = randomForest(binary~. , data =train_rf, ntree=100)
#   oob.err[mtry] = mean(fit$err.rate[,1]) #calling Out-of-bag Error
#   pred = predict(fit,test_tree, type = "class") #predict on test data
#   test.err[mtry]=mean(pred==test_tree$binary)
#   cat(mtry," ")
# }

#plot the out of bag Error for different mtry
#matplot(1:mtry, oob.err, pch = 19, col = "blue", type = "b", ylab = "Out of Bag Error", main="Random F

#plot
#matplot(1:mtry, test.err, pch = 19, col = "blue", type = "b", ylab = " Classification Rate", main="Ran

#Tune the RF
tuneRF(train_tree, response,mtryStart=2, ntree=1000,stepFactor=1.5,improve=0.05, plot=TRUE)
```

```r
#randomforests with 200 trees and mtry= sqrt(784) -- 28
rf_fit = randomForest(binary~. , data =train_rf, ntree=1000, mtry=13)

#(TRAINING)
rf_pred = predict(rf_fit,train_tree, type = "class")
mean(rf_pred==response)

#(TEST)
rf_pred = predict(rf_fit,test_tree, type = "class")
mean(rf_pred==response2)

#Show the table of results
table(rf_pred,response2)
```

Boosted Tree

```r
rm(list = ls())
require(gbm)
data = read.csv("Train_Digits_20180302.csv", header = T)

#replace digits with binary even/odd (1/0) + chnage colname
data$Digit = ifelse(data$Digit %% 2 == 0,1,0)
names(data)[1]<-paste("binary")

train = 1:2000
train_data = data[train, 2:785]
test_data = data[-train, 2:785]

response <- as.factor(data[train,1])
response2 <- as.factor(data[-train,1])

boost_fit = gbm(binary~., data=data[train,], distribution = "bernoulli", n.trees = 2000, shrinkage = 0.

summary(boost_fit)

predmat = predict(boost_fit, newdata =data[train,], n.trees = 2000, type = "response")

#classify the probability vector back into classification types (TRAIN)
result = ifelse(predmat >= 0.5 ,1,0)
mean(result==response)


#classify the probability vector back into classification types (TEST)
predmat = predict(boost_fit, newdata =data[-train,], n.trees = 2000, type = "response")
result = ifelse(predmat >= 0.5 ,1,0)
mean(result==response2)


gbmWithCrossValidation = gbm(formula = binary ~ .,
                             distribution = "bernoulli",
                             data = data[train,],
                             n.trees = 1000,
                             shrinkage = 0.01,
                             interaction.depth = 4,
                             cv.folds = 5,
                             n.cores = 1)

#used to determine the optimal number of trees -1397
gbm.perf(boost_fit)
#new model
boost_fit = gbm(binary~., data=data[train,], distribution = "bernoulli", n.trees = 1397, shrinkage = 0.0

#(Training)
predmat = predict(boost_fit, newdata =data[train,], n.trees = 1397, type = "response")
result = ifelse(predmat >= 0.5 ,1,0)
mean(result==response)

#(test)
predmat = predict(boost_fit, newdata =data[-train,], n.trees = 1397, type = "response")
result = ifelse(predmat >= 0.5 ,1,0)
mean(result==response2)
```

SVM

```
rm(list = ls())
require(e1071)
data = read.csv("Train_Digits_20180302.csv", header = T)

#replace digits with binary even/odd (1/0) + chnage colname
data$Digit = ifelse(data$Digit %% 2 == 0,1,0)
names(data)[1]<-paste("binary")

train = 1:2000
train_data = data[train, 2:785]
test_data = data[-train, 2:785]

#defining the factors
response <- as.factor(data[train,1])
response2 <- as.factor(data[-train,1])

#need to peform Principle Compenent Analysis (PCA) on variables (TRAIN)
pca_train = prcomp(train_data, scale=FALSE, center = T)

#Determine the number of components required
eigs = pca_train$sdev^2
summ= rbind(
  SD = sqrt(eigs),
  Proportion = eigs/sum(eigs),
  Cumulative = cumsum(eigs)/sum(eigs))
#57 components
which(summ[3,]<=0.85)

plot(summ[3,], main="Proportion of Variance Explained vs No. Components", xlab = "No. Components", ylab
```

```r
abline(v=57, col='red')

#only select
rotate<-pca_train$rotation[,1:57]
pca_train<-as.matrix(scale(train_data,center = TRUE, scale=FALSE))%*%(rotate)

#PCA onn test data
pca_test<-as.matrix(scale(test_data,center = TRUE, scale=FALSE))%*%(rotate)

#perform SVM on the training data
svm_fit = svm(pca_train,response, kernel="polynomial", degree =2)

#Training data
svm_pred <-predict(svm_fit,pca_train)
mean(svm_pred==response)

#Test Data
svm_pred <-predict(svm_fit,pca_test)
mean(svm_pred==response2)

#tune the model
tune_out = tune(svm,train.x=pca_train ,train.y=response , kernel="polynomial", degree =2,ranges = list(
#gamma= c(0.5,1,2,3)
summary(tune_out)

#enhance model with CV paramters
svm_fit = svm(pca_train,response, kernel="polynomial", degree = 2, cost=1)
svm_pred <-predict(svm_fit,pca_test)
mean(svm_pred==response2)
```

Neural Network

```r
require(h2o)
data = read.csv("Train_Digits_20180302.csv", header = T)
#split up the data (80:20)
split=1:2000
train = data[split,]
test = data[-split,]

#cnvert to binary 0-odd and 1-even
train$Digit = ifelse(train$Digit %% 2 == 0,1,0)
test$Digit = ifelse(test$Digit %% 2 == 0,1,0)

#declare responses as factors
train$Digit = as.factor(train$Digit)
test$Digit = as.factor(test$Digit)

#Initialize the H20 package and classify training/tets data
localH2O = h2o.init()
train_h2o = as.h2o(train)
test_h2o = as.h2o(test)
```

```r
NN_model = h2o.deeplearning(x = 2:785,
                   y = 1,
                   training_frame = train_h2o,
                   validation_frame = test_h2o,
                   distribution = "multinomial",
                   activation = "RectifierWithDropout", #activation algorithm
                   input_dropout_ratio = 0.2, # % of inputs dropout (default is zero)
                   hidden = c(300), # one layer of 200 nodes
                   sparse = TRUE,
                   nfolds = 10,
                   epochs = 21)

#Results into confusion matrix
NN_confusion<-h2o.confusionMatrix(NN_model)
#call matrix for Training Results
NN_confusion

h2o.performance(NN_model)
1-h2o.confusionMatrix(NN_model)$Error[3]
model@parameters

#s - proc.time()
plot(model)
test_h2o = as.h2o(test)
h2o_y_test <- h2o.predict(NN_model, test_h2o)
df_y_test = as.data.frame(h2o_y_test)
df_y_test = data.frame(ImageId = seq(1,length(df_y_test$predict)), Label = df_y_test$predict)
results = as.numeric(unlist(df_y_test[,2]))-1
table(results,test$Digit)
mean(results==test$Digit)

h2o.shutdown(prompt = F)
```

Submission

```r
require(h2o)
data = read.csv("Train_Digits_20180302.csv", header = T)
#split up the data (80:20)
split=1:2000
train = data[split,]
test = data[-split,]

#cnvert to binary 0-odd and 1-even
train$Digit = ifelse(train$Digit %% 2 == 0,1,0)
test$Digit = ifelse(test$Digit %% 2 == 0,1,0)

#declare responses as factors
train$Digit = as.factor(train$Digit)
test$Digit = as.factor(test$Digit)

#Initialize the H2O package and classify training/tets data
localH2O = h2o.init()
```

```r
train_h2o = as.h2o(train)
test_h2o = as.h2o(test)

NN_model = h2o.deeplearning(x = 2:785,
                   y = 1,
                   training_frame = train_h2o,
                   validation_frame = test_h2o,
                   distribution = "multinomial",
                   activation = "RectifierWithDropout", #activation algorithm
                   input_dropout_ratio = 0.2, # % of inputs dropout (default is zero)
                   hidden = c(300), # one layer of 300 nodes
                   sparse = TRUE,
                   nfolds = 10,
                   epochs = 21)

#Results into confusion matrix
NN_confusion<-h2o.confusionMatrix(NN_model)
#call matrix for Training Results
NN_confusion

h2o.performance(NN_model)
1-h2o.confusionMatrix(NN_model)$Error[3]
model@parameters

#Now pull in Vula Test Results
test_final<-read.csv("Test_Digits_20180302.csv")
h2o_output = as.h2o(test_final)
h2o_output_predict <- h2o.predict(NN_model, h2o_output)
out_test = as.data.frame(h2o_output_predict)
out_test = data.frame(Pred = out_test$predict)
head(out_test)

out_test$Pred = ifelse(out_test$Pred == 0,"odd","even")
head(out_test)

write.csv(out_test, file = "Digits_Pred_RBRDEA003.csv", row.names=F, quote = F)
h2o.shutdown(prompt = F)
```