# 503 HW 4

*Naomi Giertych*

*Due: 3/20/2018*

For this report, I chose to work with the standardized spam dataset. I standardized each column of the dataset to have mean 0 and unit variance. The standardized dataset, I believe, loses the least amount of information between all of the variables so would lead to a more generalizable result to test data.

The structure of the report is as follows: (1) investigate how sensitive SVM training and test errors are to choice of kernels and the cost and gamma tuning parameters, (2) investigate how sensitive neural network training and test errors are for the number of hidden layers and number of nodes, (3) investigate how sensitive training and test errors are for the size of a decision tree, (4) investigate how training and test errors change with respect to class-unbalanced data, and (5) investigate how ameliorating class-unbalanced data using bootstrap changes the training and test errors. Each of my investigations are concluded with a cross-validation of the parameters. **Please note that I report the final set of training and test errors for the final models at the end of the report.**

## 1. Applying SVM

The slack variable is related to the cost. It controls the amount of influence each support vector has on the margin. If the cost is high, then the margin is sensitive to points near the boundary that could be misclassified and my boundary could be really wiggly. If the cost is low, then the margin is more tolerant of points near the boundary that could be misclassified. In other words, it controls the trade-off between a smooth (more linear or planer) decision boundary and one that classifies training data correctly. Generally, I would expect a large cost to be very good at classifying the training data but not generalize well to the test dataset.

The gamma variable controls the amount of weight points near the boundary carry compared to points away from the boundary. If the gamma variable is large, points near the boundary carry a lot of weight and can greatly influence the details of the boundary, pulling it closer to them. So, like a large cost variable, we end up with a wiggly boundary that might not be generalizable to the test dataset. However, if the gamma variable is small, the boundary considers the “majority” of points on each side and adjusts accordingly. Therefore, we might perform about the same on the training and test datasets assuming that the training is representative of the test and the errors would be lower than with a low gamma.

Below I examine the choice of the slack variable (using the cost variable in R) and the gamma variable for each of the possible kernals (linear, radial [Gaussian], polynomial, and sigmoid). I chose to examine a cost of $e^{-1}$, 1 (default), and $e^2$ and a gamma of 0.001, 0.0178 (default (set to $\frac{1}{p}$ where $p$ is the dimension of the data)), and 1. (poly2 and poly3 are polynomials with 2 and 3 degrees respectively.)

Keeping the gamma variable fixed, we can see that a larger cost has smaller test errors for all of the possible kernals.

Keeping the cost variable fixed, we can see that the effect of changing the gamma variable as a less distinct result. The default gamma has the smallest test error for radial kernel, but a large gamma has smaller test errors for the polynomial kernel. (Gamma does not play a role in the linear kernel).

```
## [1] "Standardized Dataset Default Cost and Gamma"
```

| | training | test |
|---|---|---|
| linear | 0.0648843 | 0.0717080 |
| radial | 0.0515161 | 0.0632334 |
| poly2 | 0.1258559 | 0.1499348 |

| | training | test |
|---|---|---|
| poly3 | 0.2050864 | 0.2118644 |

```
## [1] "Standardized Dataset Small Cost and Default Gamma"
```

| | training | test |
|---|---|---|
| linear | 0.0671666 | 0.0697523 |
| radial | 0.0593414 | 0.0769231 |
| poly2 | 0.1953049 | 0.2033898 |
| poly3 | 0.2520378 | 0.2646675 |

```
## [1] "Standardized Dataset Large Cost and Default Gamma"
```

| | training | test |
|---|---|---|
| linear | 0.0619498 | 0.0684485 |
| radial | 0.0286925 | 0.0514993 |
| poly2 | 0.0508640 | 0.0801825 |
| poly3 | 0.0974894 | 0.1323338 |

```
## [1] "Standardized Dataset Default Cost and Small Gamma"
```

| | training | test |
|---|---|---|
| linear | 0.0648843 | 0.0717080 |
| radial | 0.0844473 | 0.0873533 |
| poly2 | 0.3909358 | 0.3970013 |
| poly3 | 0.3955005 | 0.4022164 |

```
## [1] "Standardized Dataset Default Cost and Large Gamma"
```

| | training | test |
|---|---|---|
| linear | 0.0648843 | 0.0717080 |
| radial | 0.0048908 | 0.1962190 |
| poly2 | 0.0048908 | 0.0801825 |
| poly3 | 0.0022824 | 0.0782269 |

Below I use a 10-fold cross-validation to select the tuning parameters for the standardized train dataset. Based on these results, it appears that a Gaussian kernel with a cost of $e^2$ and gamma of $\frac{1}{56}$ performs the best.

```
## [[1]]
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##       gamma     cost
##  0.01785714 7.389056
##
```

```
## - best performance: 0.05639863
##
## - Detailed performance results:
##         gamma      cost      error dispersion
## 1  0.00100000 0.3678794 0.12324945 0.01466684
## 2  0.01785714 0.3678794 0.07335803 0.01635116
## 3  1.00000000 0.3678794 0.24388452 0.02023170
## 4  0.00100000 1.0000000 0.08705265 0.01464918
## 5  0.01785714 1.0000000 0.06454940 0.01420165
## 6  1.00000000 1.0000000 0.12944051 0.01126891
## 7  0.00100000 2.7182818 0.07792681 0.01371024
## 8  0.01785714 2.7182818 0.05933661 0.01282543
## 9  1.00000000 2.7182818 0.12781291 0.01293034
## 10 0.00100000 7.3890561 0.07107471 0.01195401
## 11 0.01785714 7.3890561 0.05639863 0.01724982
## 12 1.00000000 7.3890561 0.12748611 0.01316324
##
##
## [[2]]
##
## Call:
## best.svm(x = class ~ ., data = train_set, gamma = def_gammas,
##     cost = def_costs, tuneconrol = tune.control(cross = 10))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  7.389056
##       gamma:  0.01785714
##
## Number of Support Vectors:  753
##
##  ( 403 350 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

## 2. Applying neural networks

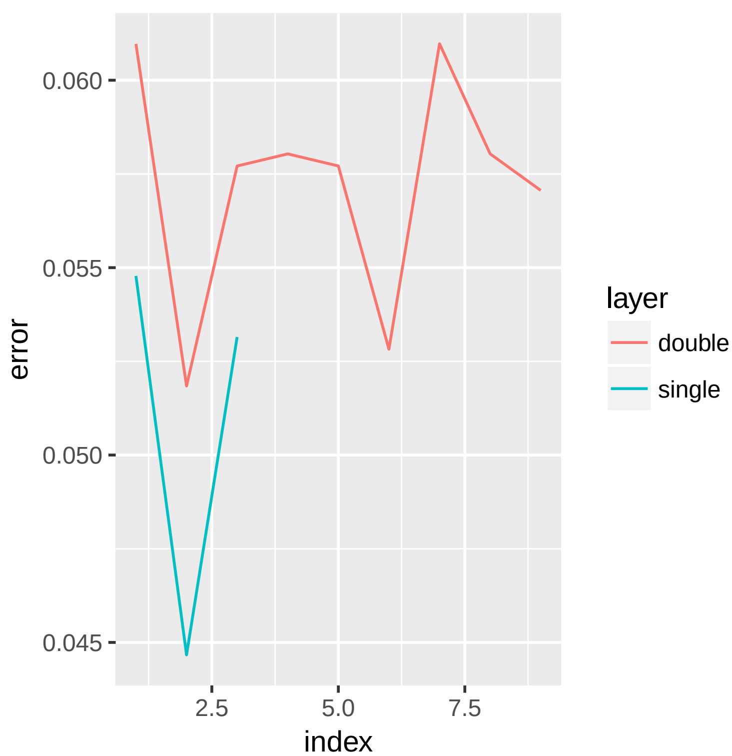
Below I examine the effect of the number of nodes and the number of layers on the training error of the dataset. I examined 1 layer with 5 and 10 nodes and 2 layers with 5 and 10 nodes each. Below is a table of the training error results.

| | 1 layer | 2 layers |
|---|---|---|
| 5 nodes | 0.0107597000 | 0.0228236061 |
| 10 nodes | 0.0071731334 | 0.0146723182 |

Based on this table, it appears that the neural network performs significantly better with 1 layer and many nodes as opposed to either 5 nodes or 10 nodes in each layer of a two layer neural network. This suggests that are data are linearly separable and that a few variables are good at classifying email into spam and not spam.

Next, I use cross-validation to select the tuning parameters for each of the training datasets. For a single layer neural network, I examined 5, 10, and 15 nodes, and for a double layer neural network, I examined all possible combinations of (5, 10, 15) and (5, 10, 15) as shown in the table below. Var1 corresponds to the number of nodes in the first layer and Var2 corresponds to the number of nodes in the second layer. The graph plots the 1 and 2 layer training errors; the index corresponds to the row in the table (so 1 layer should only have up to an index of 3). Based on this graph, I should pick a 1 layer neural network with 15 nodes (if I only wanted to use 1 layer) or a 2 layer neural network with 10 nodes in the first layer and 5 nodes in the second layer.

| Var1 | Var2 |
|---|---|
| 5 | 5 |
| 10 | 5 |
| 15 | 5 |
| 5 | 10 |
| 10 | 10 |
| 15 | 10 |
| 5 | 15 |
| 10 | 15 |
| 15 | 15 |

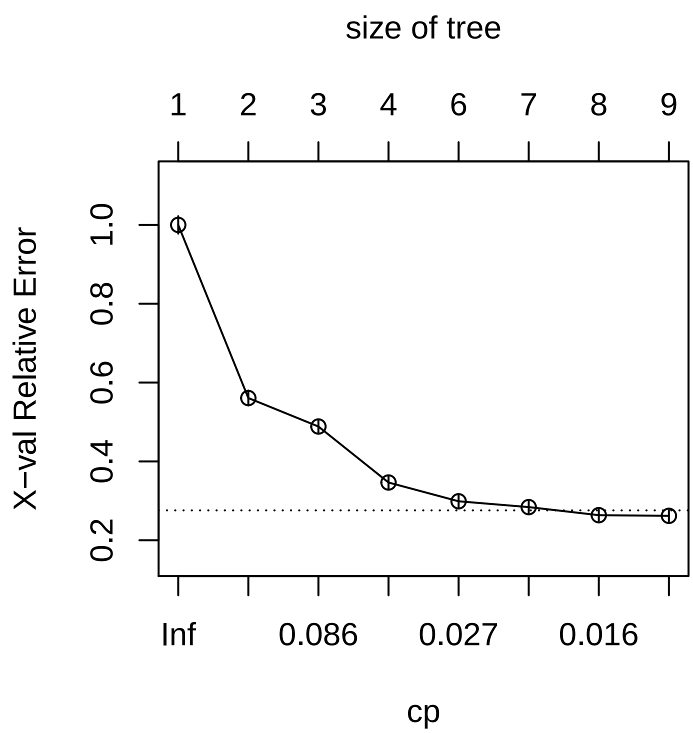



## 3. Applying decision trees

To adjust the size of the tree, I used the complexity parameter (cp) in rpart. This allows R to not attempt any split that does not decrease the lack of fit by a factor of the cp. I chose to investigate cp equal to 0, 0.01 (the default), and 0.05. The small (0) cp and the default perform exactly the same on the test data but the small cp performs better on the training data. The large cp performs significantly worse on the training and the test data. This suggests that a small number of variables are needed to correctly classify email into spam and not spam.

| | train error | test error |
|---|---|---|
| default | 0.0932507336 | 0.1043024772 |
| small | 0.0498858820 | 0.1043024772 |
| large | 0.1402021519 | 0.1427640156 |

I use cross-validation to determine the size of the tree to use as my final model. According to R documentation, a good choice of cp for pruning is the leftmost value for which the mean lies below the dashed line (which is drawn 1 standard error above the minimum of the curve). Examining the graph below, I shoud use a tree with 8 splits or with a cp of 0.016.




## 4. Class-Imbalance data

Next, I explore how imbalanced data could affect the final model choosen and the corresponding training and test errors. Similar to above, I focused on the standardized dataset. I similated unbalenced data by randomly selecting the spam email from the original dataset (training or test); I then standardized the dataset. For SVM, neural networks, and the decision trees, I use cross-validation to determine what my model would be for each of the 3:7, 2:8, and 1:9 training spam.

Below I perform cross validation for svm. For 3:7 unbalanced datasets, the best model was a Gaussian kernel with a cost of $e^1$ and a gamma of $\frac{1}{56}$. For the 2:8 and 1:9 unbalanced datasets, the best model was a Gaussian kernel with a cost of $e^2$ and a gamma of $\frac{1}{56}$.

```
## [1] "SVM Standardized 3:7 Ratio Data CV Results"

## [[1]]
##
```

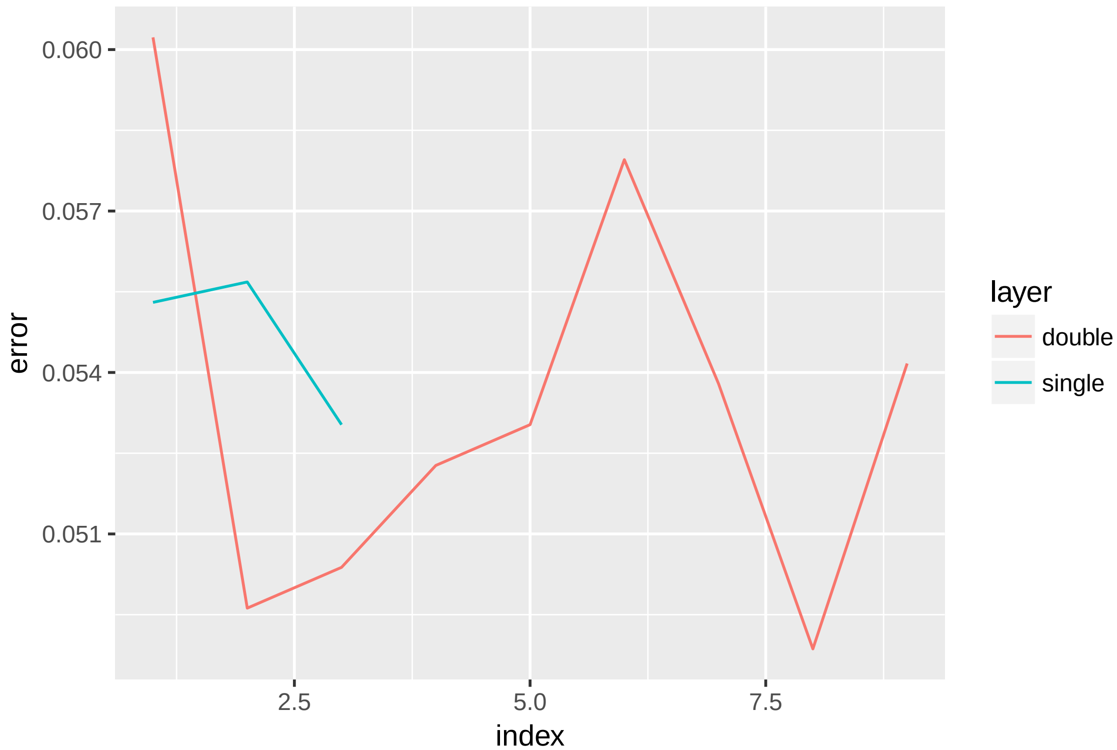
```
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##          gamma        cost
##  0.01785714286 2.718281828
##
## - best performance: 0.05416666667
##
## - Detailed performance results:
##            gamma         cost         error    dispersion
## 1  0.00100000000 0.3678794412 0.12954545455 0.01470295619
## 2  0.01785714286 0.3678794412 0.07234848485 0.01621381249
## 3  1.00000000000 0.3678794412 0.25037878788 0.01771730561
## 4  0.00100000000 1.0000000000 0.09734848485 0.01777121234
## 5  0.01785714286 1.0000000000 0.06477272727 0.01305460306
## 6  1.00000000000 1.0000000000 0.13712121212 0.01321843670
## 7  0.00100000000 2.7182818285 0.07500000000 0.01614977470
## 8  0.01785714286 2.7182818285 0.05416666667 0.01198497725
## 9  1.00000000000 2.7182818285 0.13333333333 0.01247382731
## 10 0.00100000000 7.3890560989 0.06969696970 0.01676965426
## 11 0.01785714286 7.3890560989 0.05492424242 0.01362817749
## 12 1.00000000000 7.3890560989 0.13333333333 0.01247382731
##
##
## [[2]]
##
## Call:
## best.svm(x = class ~ ., data = train_set, gamma = def_gammas,
##     cost = def_costs, tuneconrol = tune.control(cross = 10))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  2.718281828
##       gamma:  0.01785714286
##
## Number of Support Vectors:  692
##
##  ( 299 393 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1

## [1] "SVM Standardized 2:8 Ratio Data CV Results"

## [[1]]
##
## Parameter tuning of 'svm':
##
```

```
## - sampling method: 10-fold cross validation
##
## - best parameters:
##          gamma        cost
##  0.01785714286 7.389056099
##
## - best performance: 0.0480351545
##
## - Detailed performance results:
##            gamma         cost         error    dispersion
## 1  0.00100000000 0.3678794412 0.12245857591 0.02821005167
## 2  0.01785714286 0.3678794412 0.07139871623 0.01805616618
## 3  1.00000000000 0.3678794412 0.17437863860 0.02665181302
## 4  0.00100000000 1.0000000000 0.08957120466 0.02414832040
## 5  0.01785714286 1.0000000000 0.05928310196 0.01514679196
## 6  1.00000000000 1.0000000000 0.12203873712 0.02593033131
## 7  0.00100000000 2.7182818285 0.07355948649 0.01656941732
## 8  0.01785714286 2.7182818285 0.05106359158 0.01336291364
## 9  1.00000000000 2.7182818285 0.11814823108 0.02619548532
## 10 0.00100000000 7.3890560989 0.06014330497 0.01686612689
## 11 0.01785714286 7.3890560989 0.04803515450 0.01547879105
## 12 1.00000000000 7.3890560989 0.11858113151 0.02580771021
##
##
## [[2]]
##
## Call:
## best.svm(x = class ~ ., data = train_set, gamma = def_gammas,
##     cost = def_costs, tuneconrol = tune.control(cross = 10))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  7.389056099
##       gamma:  0.01785714286
##
## Number of Support Vectors:  549
##
##  ( 213 336 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1

## [1] "SVM Standardized 1:9 Ratio Data CV Results"

## [[1]]
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
```

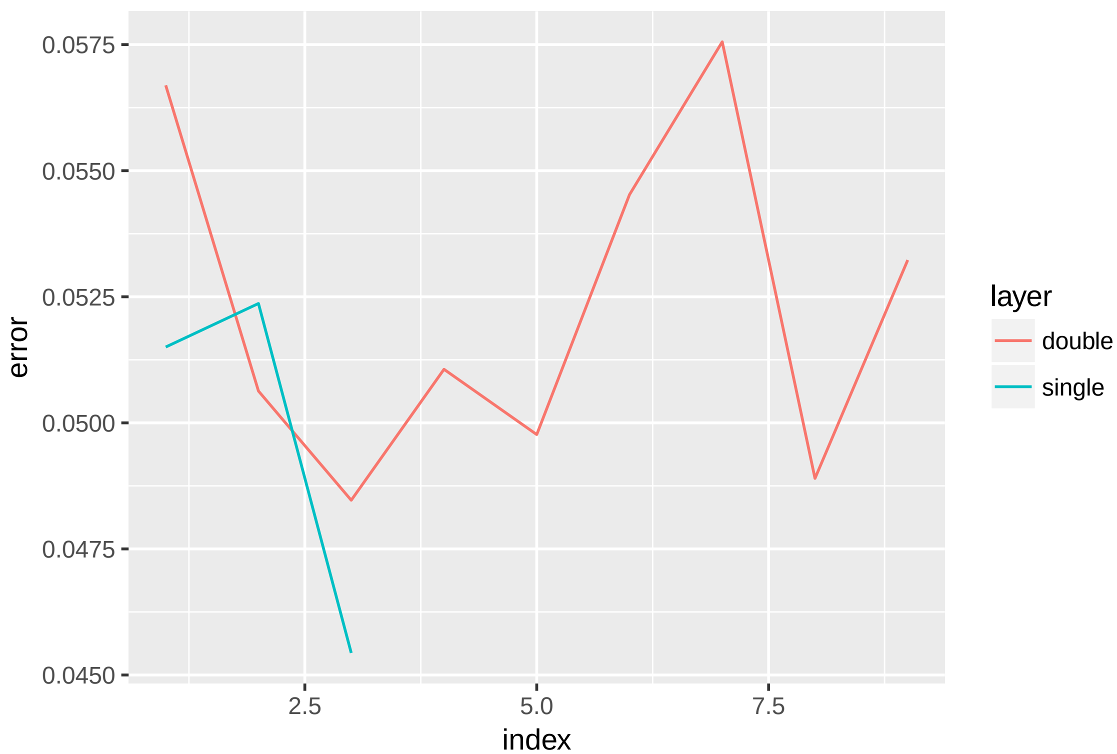
```
## - best parameters:
##          gamma        cost
##  0.01785714286 7.389056099
##
## - best performance: 0.03647170258
##
## - Detailed performance results:
##            gamma         cost         error     dispersion
## 1  0.00100000000 0.3678794412 0.07879232773 0.018841757794
## 2  0.01785714286 0.3678794412 0.05836372247 0.015012538567
## 3  1.00000000000 0.3678794412 0.09872839214 0.020567471917
## 4  0.00100000000 1.0000000000 0.06566185176 0.015410123986
## 5  0.01785714286 1.0000000000 0.04134264741 0.011955743935
## 6  1.00000000000 1.0000000000 0.07829741890 0.017002297003
## 7  0.00100000000 2.7182818285 0.05399479043 0.015973421735
## 8  0.01785714286 2.7182818285 0.03647407057 0.008324139821
## 9  1.00000000000 2.7182818285 0.07538006157 0.017735393430
## 10 0.00100000000 7.3890560989 0.04086905044 0.012425408401
## 11 0.01785714286 7.3890560989 0.03647170258 0.010025915967
## 12 1.00000000000 7.3890560989 0.07538006157 0.017735393430
##
##
## [[2]]
##
## Call:
## best.svm(x = class ~ ., data = train_set, gamma = def_gammas,
##     cost = def_costs, tuneconrol = tune.control(cross = 10))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  7.389056099
##       gamma:  0.01785714286
##
## Number of Support Vectors:  381
##
##  ( 129 252 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

Below I perform cross-validation for the neural network. Based on the graphs, I should chose a neural network with 2 layers and 15 nodes in first layer and 5 nodes in the second layer for the 3:7 ratios, a neural network with 1 layers and 10 nodes for the 2:8 ratio, and a neural network with 15 nodes in the first and 10 nodes in the second for the 1:9 ratio.

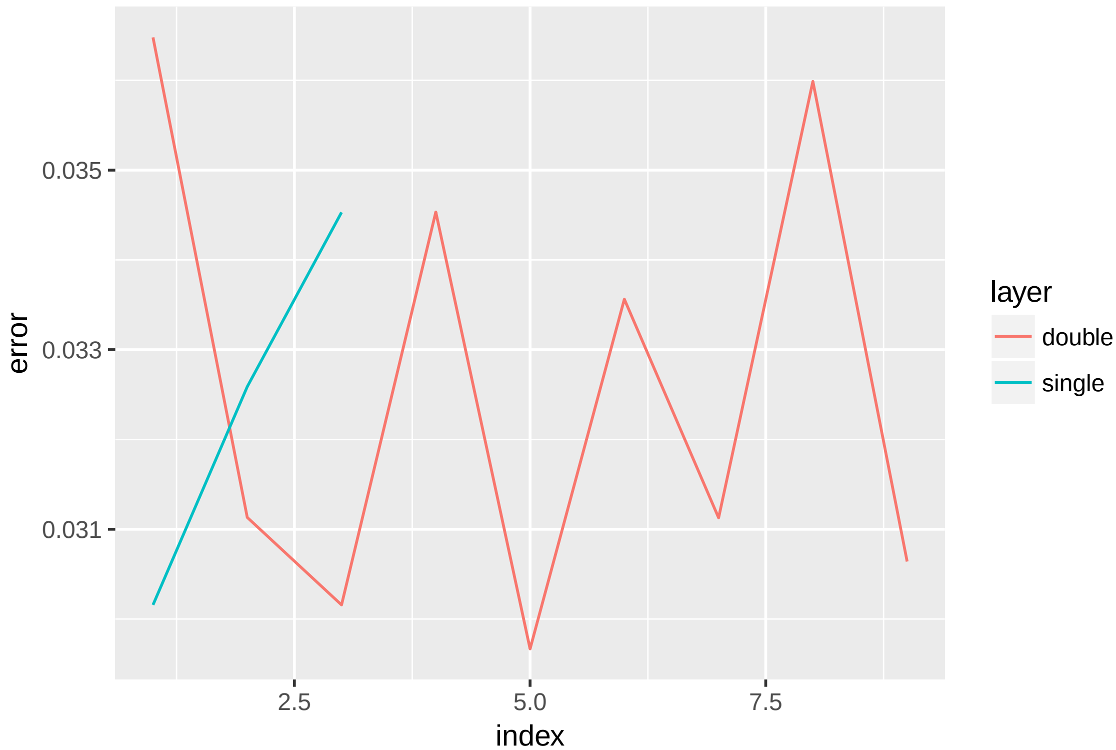
```
## [1] "Neural Network Standardized 3:7 Ratio Data CV Results"
```




## [1] "Neural Network Standardized 2:8 Ratio Data CV Results"

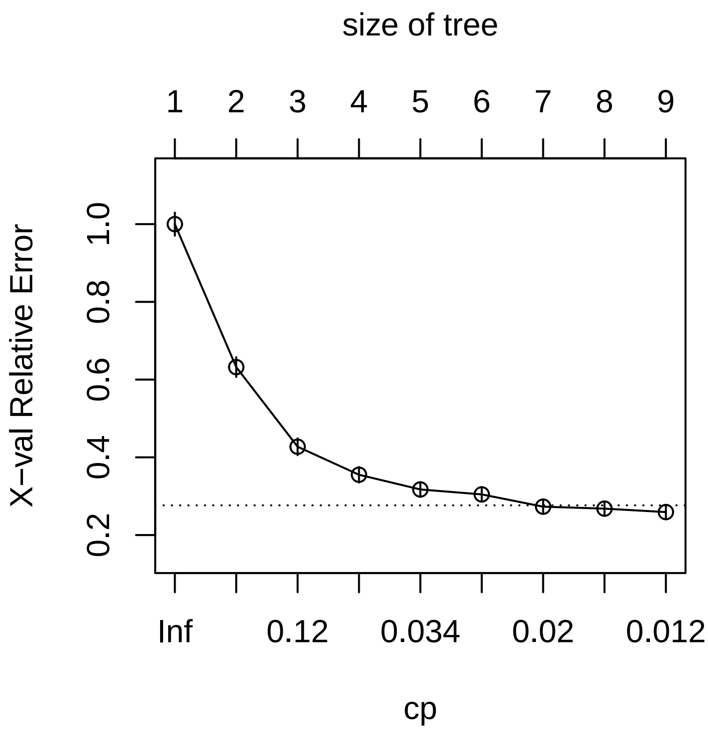



## [1] "Neural Network Standardized 1:9 Ratio Data CV Results"

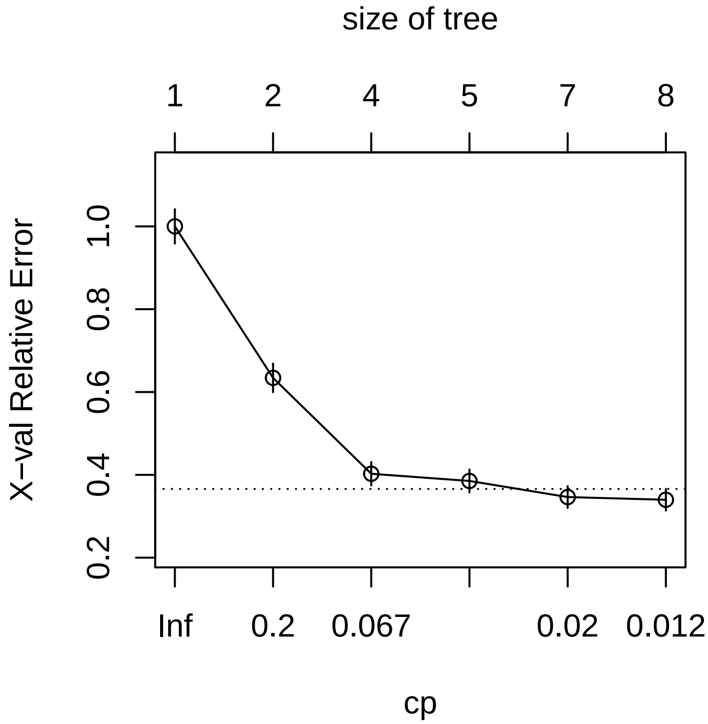



Finally, I perform cross-validation to determine the size of my tree for each of the unbalanced datasets. Based on the graphs, I should should have a tree with 7 splits (cp of 0.016) for the 3:7 ratio, a tree with 7 splits (cp of 0.018), and a tree with 8 splits (cp of 0.019) for the 1:9 ratio.

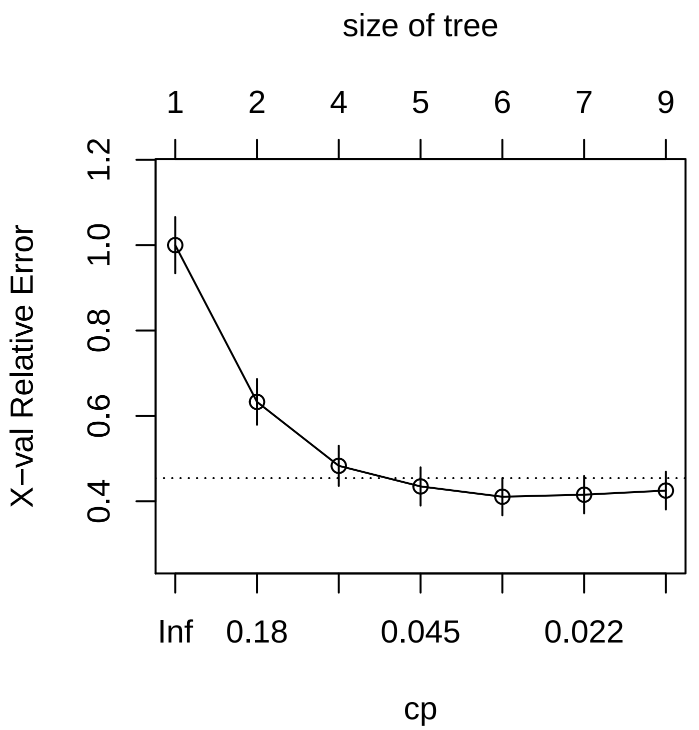
```
## [1] "Decision Tree Standardized 3:7 Ratio Data CV Results"
```




```
## [1] "Decision Tree Standardized 2:8 Ratio Data CV Results"
```




```
## [1] "Decision Tree Standardized 1:9 Ratio Data CV Results"
```




Below is a table of the training and test data errors for svm using the final models determined above. As we can see the training and test errors are the smallest for the dataset with a 1:9 ratio. This sort of makes sense however sense the largest amount of error that could occur in the dataset is 11% (just classifying everything as not spam). It is interesting how close all of the training and test errors regardless of the the ratio of spam and not spam emails in the dataset. This suggests that the training data is a good representation of the test data.

| | train error | test error |
|---|---|---|
| 4:6 ratio | 0.0286925334 | 0.0514993481 |
| 3:7 ratio | 0.0344696970 | 0.0622627183 |
| 2:8 ratio | 0.0181739507 | 0.0460869565 |
| 1:9 ratio | 0.0121595331 | 0.0460333007 |

Below is a table of the training and test data errors for neural networks using the final models determined above. Similar to the results of SVM, the training and test errors are the smallest for the dataset with a 1:9 ratio. Unlike SVM, the training errors are orders of magnitude smaller than the test errors. It is interesting that the test errors are about the same as for SVM: slightly larger for the original dataset and the 2:8 ratio, but smaller for the 3:7 and 1:9 unbalanced datasets.

| | train error | test error |
|---|---|---|
| 4:6 ratio | 0.0045647212 | 0.0827900913 |
| 3:7 ratio | 0.0087121212 | 0.0690964313 |
| 2:8 ratio | 0.0086542622 | 0.0434782609 |
| 1:9 ratio | 0.0043774319 | 0.0391772772 |

Below is a table of the training and test data errors for a decision tree using the final models determined above. Similar to the results of SVM, the training and test errors are the smallest for the dataset with a 1:9 ratio. Like SVM, the training errors are orders of magnitude smaller than the test errors. However, the training and test errors are much larger for the decision trees as compared to SVM and neural networks.

| | train error | test error |
|---|---|---|
| 4:6 ratio | 0.0984675579 | 0.1173402868 |
| 3:7 ratio | 0.0750000000 | 0.1047835991 |
| 2:8 ratio | 0.0597144093 | 0.0634782609 |
| 1:9 ratio | 0.0296692607 | 0.0597453477 |

Finally, I "fix" the unbalanced data by bootstrap sampling from the spam data in the unbalanced standardized dataset until the data was 50-50 split for spam and non-spam data. For instance, for the 3:7 ratio, I sampled with replacement from the 3/10ths of data that was labelled as spam until I reached the same number of observations as the non-spam data.

Next, I perform cross-validation for SVM to determine the best model for each of the rebalanced datasets. Based on the results, I should use a Gaussian kernel with a cost of $e^2$ and a gamma of 0.001 for the rebalanced 3:7 ratio data and for the rebalanced 2:8 ratio data, and a Gaussian kernel with a cost of $e^1$ and a gamma of 0.001 for the rebalanced 1:9 ratio data.

```
## [1] "SVM Standardized 3:7 Ratio Rebalanced Data CV Results"
```

```
## [[1]]
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma cost
##  0.001    1
##
```

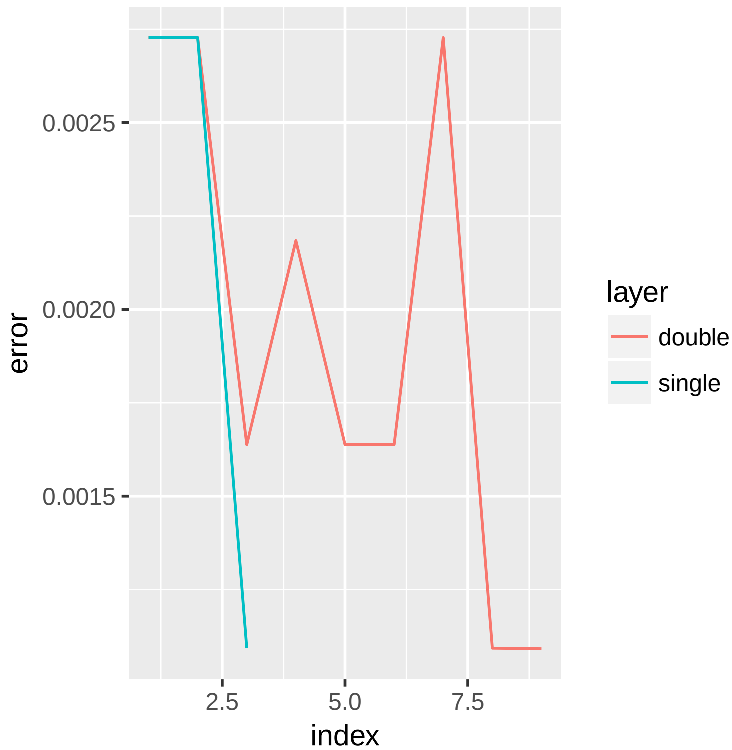
```
## - best performance: 0.005458541221
##
## - Detailed performance results:
##            gamma         cost          error     dispersion
## 1  0.00100000000 0.3678794412 0.012007008791 0.006198050882
## 2  0.01785714286 0.3678794412 0.011472440010 0.009091421767
## 3  1.00000000000 0.3678794412 0.092248752673 0.018998622182
## 4  0.00100000000 1.0000000000 0.005458541221 0.005151977193
## 5  0.01785714286 1.0000000000 0.008740199572 0.006415834333
## 6  1.00000000000 1.0000000000 0.093885127109 0.018526619532
## 7  0.00100000000 2.7182818285 0.006004989309 0.006014300862
## 8  0.01785714286 2.7182818285 0.008740199572 0.006415834333
## 9  1.00000000000 2.7182818285 0.092792230934 0.018733552804
## 10 0.00100000000 7.3890560989 0.006551437396 0.007195336752
## 11 0.01785714286 7.3890560989 0.008193751485 0.006441359154
## 12 1.00000000000 7.3890560989 0.092792230934 0.018733552804
##
##
## [[2]]
##
## Call:
## best.svm(x = class ~ ., data = train_set, gamma = def_gammas,
##     cost = def_costs, tuneconrol = tune.control(cross = 10))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.001
##
## Number of Support Vectors:  459
##
##  ( 232 227 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1

## [1] "SVM Standardized 2:8 Ratio Rebalanced Data CV Results"

## [[1]]
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  gamma        cost
##  0.001 7.389056099
##
## - best performance: 0.006007959135
##
```

```
## - Detailed performance results:
##            gamma         cost          error     dispersion
## 1  0.00100000000 0.3678794412 0.013640413400 0.010364236595
## 2  0.01785714286 0.3678794412 0.009289617486 0.008562957576
## 3  1.00000000000 0.3678794412 0.056750415776 0.016030184001
## 4  0.00100000000 1.0000000000 0.010367664528 0.009082345433
## 5  0.01785714286 1.0000000000 0.006557377049 0.007194323373
## 6  1.00000000000 1.0000000000 0.052928248990 0.014475282514
## 7  0.00100000000 2.7182818285 0.007100855310 0.007309453847
## 8  0.01785714286 2.7182818285 0.006554407223 0.006717879995
## 9  1.00000000000 2.7182818285 0.052928248990 0.014475282514
## 10 0.00100000000 7.3890560989 0.006007959135 0.005434377781
## 11 0.01785714286 7.3890560989 0.006554407223 0.006717879995
## 12 1.00000000000 7.3890560989 0.052928248990 0.014475282514
##
##
## [[2]]
##
## Call:
## best.svm(x = class ~ ., data = train_set, gamma = def_gammas,
##     cost = def_costs, tuneconrol = tune.control(cross = 10))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  7.389056099
##       gamma:  0.001
##
## Number of Support Vectors:  155
##
##  ( 81 74 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1

## [1] "SVM Standardized 1:9 Ratio Rebalanced Data CV Results"

## [[1]]
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##          gamma cost
##  0.01785714286    1
##
## - best performance: 0.001092896175
##
## - Detailed performance results:
##            gamma         cost          error     dispersion
```

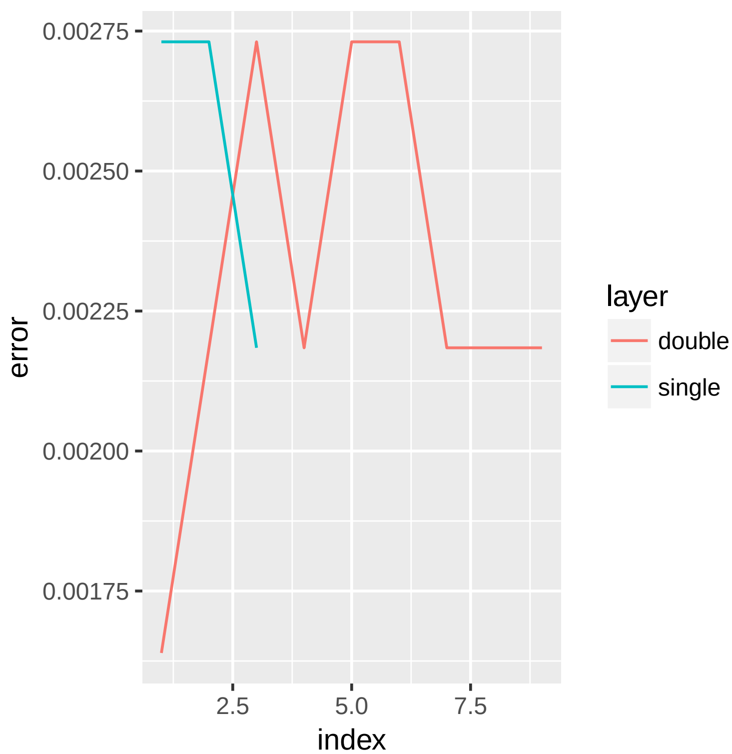
```
## 1  0.00100000000 0.3678794412 0.008728320266 0.005846569561
## 2  0.01785714286 0.3678794412 0.002729270611 0.002876917452
## 3  1.00000000000 0.3678794412 0.027833214540 0.014410442585
## 4  0.00100000000 1.0000000000 0.007091945830 0.004479568796
## 5  0.01785714286 1.0000000000 0.001092896175 0.002304027439
## 6  1.00000000000 1.0000000000 0.002729270611 0.003861648881
## 7  0.00100000000 2.7182818285 0.003816227132 0.004487163446
## 8  0.01785714286 2.7182818285 0.001092896175 0.002304027439
## 9  1.00000000000 2.7182818285 0.002729270611 0.003861648881
## 10 0.00100000000 7.3890560989 0.004362675220 0.004299597111
## 11 0.01785714286 7.3890560989 0.001092896175 0.002304027439
## 12 1.00000000000 7.3890560989 0.002729270611 0.003861648881
##
##
## [[2]]
##
## Call:
## best.svm(x = class ~ ., data = train_set, gamma = def_gammas,
##     cost = def_costs, tuneconrol = tune.control(cross = 10))
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
##       gamma:  0.01785714286
##
## Number of Support Vectors:  186
##
##  ( 88 98 )
##
##
## Number of Classes:  2
##
## Levels:
##  0 1
```

Next, I perform cross-validation for SVM to determine the best model for each of the rebalanced datasets. Based on the results, I should use a 2 layer neural network with 10 nodes in each layer for the 3:7 rebalanced data, a 2 layer neural network with 5 nodes in both for the 2:8 rebalanced data, and 1 layer with 5 nodes for the 1:9 rebalanced dataset.

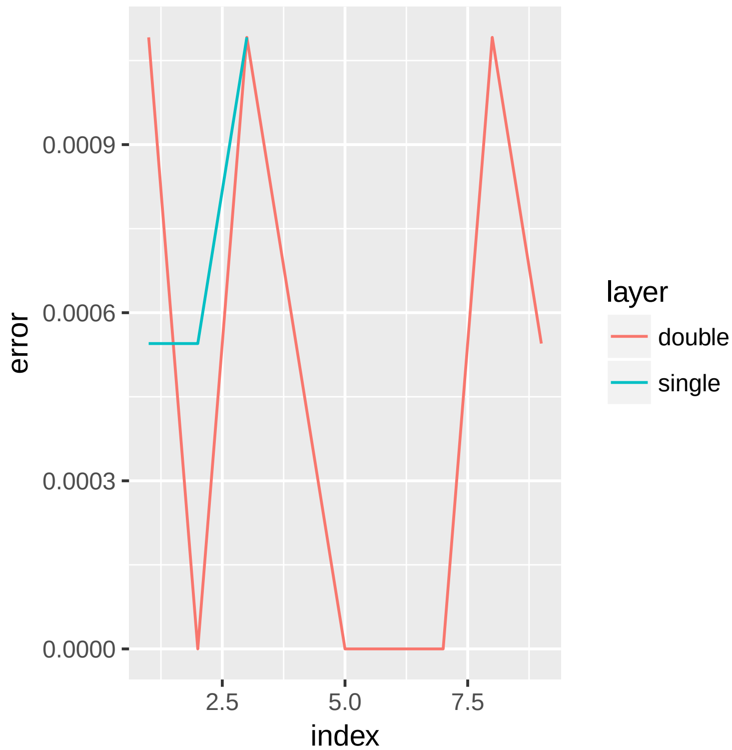
```
## [1] "Neural Network Standardized 3:7 Ratio Rebalanced Data CV Results"
```




```
## [1] "Neural Network Standardized 2:8 Ratio Rebalanced Data CV Results"
```

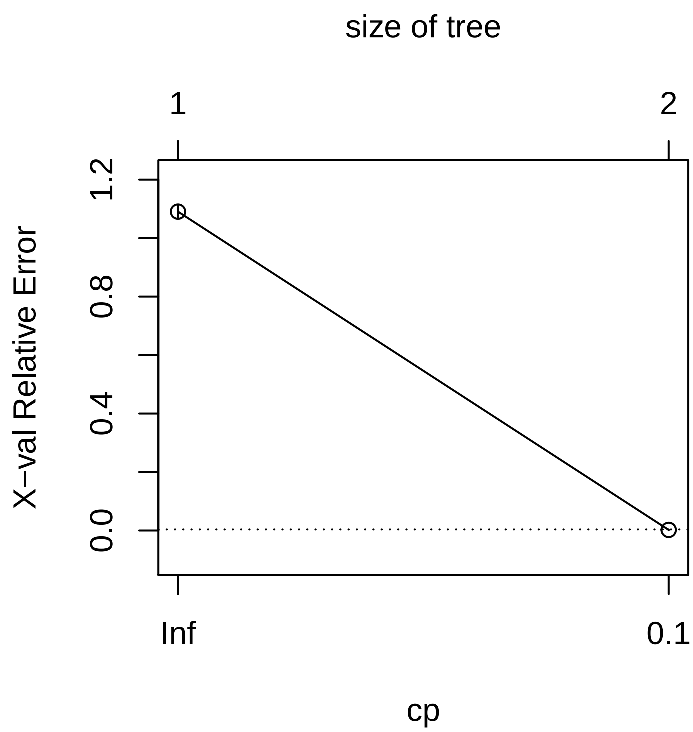



```
## [1] "Neural Network Standardized 1:9 Ratio Rebalanced Data CV Results"
```

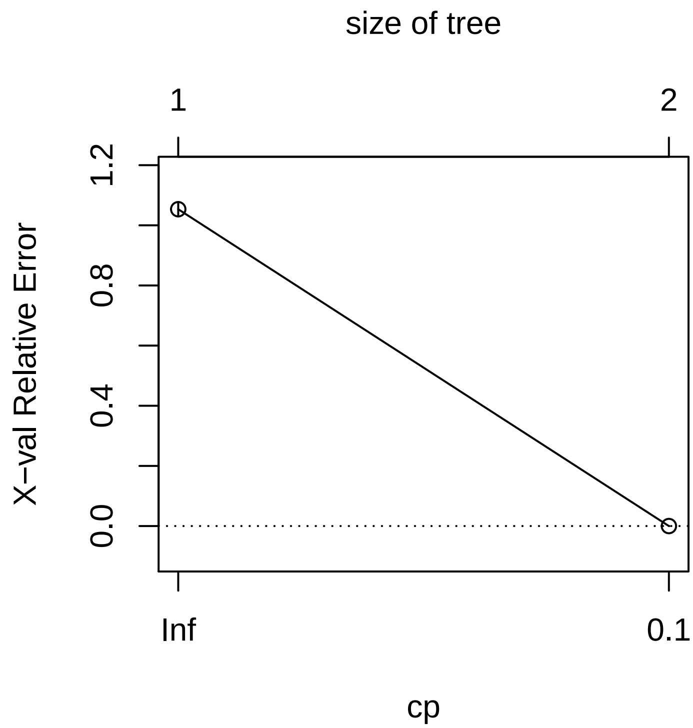



Next, I perform cross-validation for decision trees to determine the best model for each of the rebalanced datasets. Based on the results, I should use a decision tree with 2 splits and a cp of 0.1.

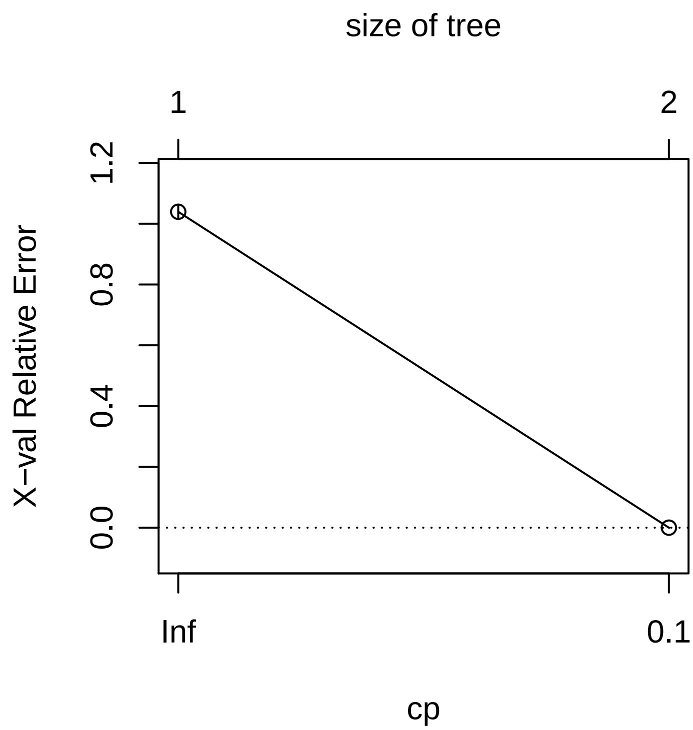
```
## [1] "Decision Tree Standardized 3:7 Ratio Rebalanced Data CV Results"
```




```
## [1] "Decision Tree Standardized 2:8 Ratio Rebalanced Data CV Results"
```




```
## [1] "Decision Tree Standardized 1:9 Ratio Rebalanced Data CV Results"
```




Below is a table of the training and test data errors for SVM using the final models for the rebalanced data determined above. The training error is 0 and my test error is signficantly smaller than the test errors of SVM for the original and unbalanced data above.

| | train error | test error |
|---|---|---|
| 3:7 ratio | 0 | 0.0027292576 |

| | train error | test error |
|---|---|---|
| 2:8 ratio | 0 | 0.0021834061 |
| 1:9 ratio | 0 | 0.0403930131 |

Below is a table of the training and test data errors for neural networks using the final models for the rebalanced data discussed above. The training error is 0 and my test error is signficantly smaller than the test errors of neural networks for the original and unbalanced data above (with the exception of the 2:8 rebalanced dataset). Similar to the unbalanced data, the test errors are lower for the neural networks than for SVM.

| | train error | test error |
|---|---|---|
| 3:7 ratio | 0 | 0.0005458515 |
| 2:8 ratio | 0 | 0.0032751092 |
| 1:9 ratio | 0 | 0.0152838428 |

Below is a table of the training and test data errors for a decision tree using the final models for the rebalanced data discussed above. These errors are much higher than the decision tree errors for the unbalanced data and for any of the errors for the unbalanced or rebalanced datasets.

| | train error | test error |
|---|---|---|
| 3:7 ratio | 0.6174242424 | 0.6074411541 |
| 2:8 ratio | 0.7412375595 | 0.7426086957 |
| 1:9 ratio | 0.7923151751 | 0.7845249755 |

Overall, I would say that rebalancing the data using bootstrap for the underrepresented class helps improved the classification performance for SVM and neural network but not for decision trees.

```r
library(knitr)
knitr::opts_chunk$set(echo = FALSE, warning = FALSE, fig.align = "center", fig.height = 4, fig.width = 4

setwd("~/Documents/UMich/Classes/2018 Spring/STATS 503/Homework/HW4/")
library(knitr)
library(rpart)
library(e1071)
library(neuralnet) # multiple layers
library(nnet) # only one layer
library(ggplot2)

set.seed(2987554)
set.seed(2987554)
##########################
# read in the training and test data
##########################

# excel can be magical sometimes
col_names <- c('word_freq_make',    'word_freq_address',    'word_freq_all',    'word_freq_3d',
               'word_freq_our', 'word_freq_over',   'word_freq_remove', 'word_freq_internet',
               'word_freq_order',   'word_freq_mail',   'word_freq_receive',    'word_freq_will',
               'word_freq_people',  'word_freq_report', 'word_freq_addresses',
               'word_freq_free',    'word_freq_business',   'word_freq_email',  'word_freq_you',
               'word_freq_credit',  'word_freq_your',   'word_freq_font',   'word_freq_000',
```

```
               'word_freq_money',   'word_freq_hp', 'word_freq_hpl',    'word_freq_george',
               'word_freq_650', 'word_freq_lab',    'word_freq_labs',   'word_freq_telnet',
               'word_freq_857', 'word_freq_data',   'word_freq_415',    'word_freq_85',
               'word_freq_technology',  'word_freq_1999',   'word_freq_parts',  'word_freq_pm',
               'word_freq_direct',  'word_freq_cs', 'word_freq_meeting','word_freq_original',
               'word_freq_project', 'word_freq_re', 'word_freq_edu',    'word_freq_table',
               'word_freq_conference',  'char_freq_;',  'char_freq_(',  'char_freq_[',
               'char_freq_!',   'char_freq_$',  'char_freq_#',  'capital_run_length_average',
               'capital_run_length_longest',    'capital_run_length_total', 'class')
train_spam <- read.table("spam-train.txt", sep = ",", col.names = col_names)
train_spam$class <- as.factor(train_spam$class)

test_spam <- read.table("spam-test.txt", sep = ",", col.names = col_names)
test_spam$class <- as.factor(test_spam$class)

##########################
# create separate datasets
##########################

# training dataset
  stand_train_spam <- as.data.frame(scale(train_spam[,-58]))
  # add class variable
    stand_train_spam$class <- train_spam$class

# train datasets
  stand_test_spam <- as.data.frame(scale(test_spam[,-58]))
    # add class variable
    stand_test_spam$class <- test_spam$class

set.seed(2987554)
##############################################################################
# apply SVM to each type of training and train dataset
##############################################################################

svm_fun <- function(train_set, test_set, def_cost, def_gamma, scale_tog){

  ######################################
  # linear kernal
  ######################################

  svmlinear <- svm(class ~ ., data=train_set,
                 kernel="linear", cost=def_cost, scale = scale_tog)
  # what is cost referring to? is this the amount of slack? or is that epsilon? #what is gamma

  ### Summary of results
  summary(svmlinear)

  ### Prediction on train data set
  svmlinear_train_error <- 1-sum(diag(table(train_set$class,
                         predict(svmlinear,train_set))))/
                         nrow(train_set)

  ### Prediction on test data set
```

```r
svmlinear_test_error <- 1-sum(diag(table(test_set$class,
                        predict(svmlinear,test_set))))/
                        nrow(test_set)

linear <- c(svmlinear_train_error, svmlinear_test_error)

#######################################
# radial kernel: exponential minus gamma
#######################################

svmradial = svm(class ~ ., data=train_set,
                     kernel="radial", cost=def_cost, gamma = def_gamma, scale = scale_tog)

### Prediction on train data set
svmradial_train_error <- 1-sum(diag(table(train_set$class,
                        predict(svmradial,train_set))))/
                        nrow(train_set)

### Prediction on test data set
svmradial_test_error <- 1-sum(diag(table(test_set$class,
                        predict(svmradial,test_set))))/
                        nrow(test_set)

radial <- c(svmradial_train_error, svmradial_test_error)

#######################################
# polynomial kernel
#######################################
poly <- c()
for (i in 2:3) {
  svmpoly = svm(class ~ ., data=train_set,
                   kernel="polynomial", cost=def_cost, degree = i, gamma = def_gamma, scale = scale_t

  ### Summary of results
  summary(svmpoly)

  ### Prediction on train data set
  svmpoly_train_error <- 1-sum(diag(table(train_set$class,
                          predict(svmpoly,train_set))))/
                          nrow(train_set)

  ### Prediction on test data set
  svmpoly_test_error <- 1-sum(diag(table(test_set$class,
                          predict(svmpoly,test_set))))/
                          nrow(test_set)

  poly_t <- c(svmpoly_train_error, svmpoly_test_error)
  poly <- rbind(poly, poly_t)
}
 rownames(poly) <- c("poly2", "poly3")

#######################################
# combine all of the errors
```

```
  #######################################
  errors <- rbind(linear, radial, poly)
  colnames(errors) <- c("training", "test")
  return(kable(errors))
}

#######################################
# Standardized Dataset
#######################################

  # default mode
    default_cost = 1
    default_gamma = 1/56
    print("Standardized Dataset Default Cost and Gamma")
    scale_toggle = TRUE
    svm_fun(stand_train_spam, stand_test_spam, default_cost, default_gamma, scale_toggle)

  # change cost variable to be small
    cost = exp(-1)
    print("Standardized Dataset Small Cost and Default Gamma")
    svm_fun(stand_train_spam, stand_test_spam, cost, default_gamma, scale_toggle)

  # change cost variable to be large
    cost = exp(2)
    print("Standardized Dataset Large Cost and Default Gamma")
    svm_fun(stand_train_spam, stand_test_spam, cost, default_gamma,scale_toggle)

  # change gamma variable to be small
    gamma <- 0.001
    print("Standardized Dataset Default Cost and Small Gamma")
    svm_fun(stand_train_spam, stand_test_spam, default_cost, gamma, scale_toggle)

  # change gamma variable to be large
    gamma <- 1
    print("Standardized Dataset Default Cost and Large Gamma")
    svm_fun(stand_train_spam, stand_test_spam, default_cost, gamma, scale_toggle)

set.seed(2987554)
###############################################################################
# built in tuning function
# this is the same as CV
###############################################################################

svm_cv_fun <- function(train_set, def_costs, def_gammas) {
  tune.out = tune.svm(class ~ ., data = train_set, cost = def_costs, gamma = def_gammas,
                      tuneconrol = tune.control(cross = 10))
  tune.out_sum <- summary(tune.out)
  bestmod =tune.out$best.model # get's best tuning parameter
  bestmod_sum <- summary(bestmod)
  return(list(tune.out_sum, bestmod_sum))
}

# define potential costs and gammas
```

```
costs = exp(-1:2)
gammas = c(0.001, 1/56, 1)

svm_cv_fun(stand_train_spam, costs, gammas)

set.seed(2987554)
#Create formula for neural networks
# c1 and c2 are dummy variables that classify the emails into non-spam and spam respectively
spam_formula <- formula(paste("c1 + c2", paste(colnames(train_spam)[-58], collapse = " + "), sep = " ~

#Create separate dummy variables for each class
nn_dataset_fun <- function(dataset) {
  spam_classind <- as.data.frame(class.ind(dataset$class))

  #Rename columns to reflect formula
  colnames(spam_classind) <- c("c1", "c2")

  nn_train_spam <- cbind(dataset, spam_classind)
  return(nn_train_spam)
}

# training datasets
  nn_stand_train_spam <- nn_dataset_fun(stand_train_spam)

# test datasets
  nn_stand_test_spam <- nn_dataset_fun(stand_test_spam)

set.seed(2987554)
################################################################################
# train neural network function
################################################################################

# function to compute the error
pred = function(nn, dat) {
  yhat = compute(nn, dat)$net.result
  yhat = apply(yhat, 1, which.max)
  yhat <- (yhat == 2)*1
  return(yhat)
}

nn_fun <- function(nn_set) {

  # Train neural network using 5 neurons in 1 layer
  neuralnet_train15 <- neuralnet(spam_formula, nn_set, hidden = 5, linear.output = F)

  # Train neural network using 10 neurons in 1 layer
  neuralnet_train110 <- neuralnet(spam_formula, nn_set, hidden = 10, linear.output = F)

  # Train neural network using 5 neurons in 2 layers
  neuralnet_train25 <- neuralnet(spam_formula, nn_set, hidden = c(5,5), linear.output = F)

  # Train neural network using 10 neurons in 2 layers
  neuralnet_train210 <- neuralnet(spam_formula, nn_set, hidden = c(10,10), linear.output = F)
```

```r
  # Calculate errors
  # 1 layer, 5 nodes
    nn_error_pred15 <- pred(neuralnet_train15, nn_set[, -c(58,59,60)])
    error15 <- mean(nn_error_pred15 != nn_set$class)

  # 1 layer, 10 nodes
    nn_error_pred110 <- pred(neuralnet_train110, nn_set[, -c(58,59,60)])
    error110 <- mean(nn_error_pred110 != nn_set$class)

  # 2 layers, 5 nodes
    nn_train_pred25 <- pred(neuralnet_train25, nn_set[, -c(58,59,60)])
    error25 <- mean(nn_train_pred25 != nn_set$class)

  # 2 layers, 10 nodes
    nn_train_pred210 <- pred(neuralnet_train210, nn_set[, -c(58,59,60)])
    error210 <- mean(nn_train_pred210 != nn_set$class)

  # combine training and test errors to create a table
  nn_error5 <- cbind(error15, error25)
  nn_error10 <- cbind(error110, error210)
  nn_error <- rbind(nn_error5, nn_error10)
  rownames(nn_error) <- c("5 nodes", "10 nodes")
  colnames(nn_error) <- c("1 layer", "2 layers")

  return(kable(nn_error))

}

nn_fun(nn_stand_train_spam)

set.seed(2987554)
# Cross validate the hidden number of nodes for any layer neural network.
# What is the error rate (training and test) for the best model according to cross validation?

nn_cv_error_calc <- function(train_df, cv_formula, num_nodes, num_folds = 5) {

  cv_folds <- split(1:nrow(train_df), 1:num_folds)

  cv_error <- mean(sapply(cv_folds, function(fold) {
      cv_nn <- neuralnet(cv_formula, train_df[-fold,], hidden = num_nodes, linear.output = F)
      mean(train_df$class[fold] != pred(cv_nn, train_df[fold, c(-58, -59, -60)]))}))
  return(cv_error)
}

# make grid for 2 layers
grid_n_l <- expand.grid(seq(5,15,5),seq(5,15,5))
kable(grid_n_l)

nn_cv_error_plot <- function(train_set, nodes_grid) {

  # one layer
  nodes_unique <- as.data.frame(unique(nodes_grid$Var1))
  test_cv_error1 <- sapply(1:nrow(nodes_unique), function(i) {nn_cv_error_calc(train_set, spam_formula,
```

```
  test_cv_error1 <- as.data.frame(test_cv_error1)
  colnames(test_cv_error1) <- "error"
  test_cv_error1$layer <- "single"
  test_cv_error1$index <- 1:nrow(nodes_unique)

  # two layers
  test_cv_error2 <- c()
  test_cv_error2 <- cbind(test_cv_error2, sapply(1:nrow(nodes_grid),
      function(i) {nn_cv_error_calc(train_set, spam_formula,
        c(as.numeric(nodes_grid[i,1]), as.numeric(nodes_grid[i,2])))}))
  test_cv_error2 <- as.data.frame(test_cv_error2)
  colnames(test_cv_error2) <- "error"
  test_cv_error2$layer <- "double"
  test_cv_error2$index <- 1:nrow(nodes_grid)

  errors_cv <- rbind(test_cv_error1, test_cv_error2)

  ggplot(errors_cv) + geom_line(aes(x = index, y = error, colour = layer))
}

nn_cv_error_plot(nn_stand_train_spam, grid_n_l)

set.seed(2987554)
# Classification tree prediction error for training and test datasets
# do we need to play with the cp parameter?

tree_train_fun <- function(train_set, def_cp){
  tree_train <- rpart(class~., data=train_set, control = rpart.control(cp = def_cp))
}

tree_fun <- function(tree_train, train_set, test_set){

  train_predict = predict(tree_train ,train_set, type="class")
  train_error <- mean(train_predict != train_set$class)

  test_predict = predict(tree_train ,test_set, type="class")
  test_error <- mean(test_predict != test_set$class)

  error <- c(train_error, test_error)

  return(error)
}

# Cp controls the complexity of the tree. Small cp suggests large trees, large cp suggests small trees
cp_test <- function(cps, train_set, test_set){
  errors <- c()
  for(i in 1:3){
    stand_tree_fun <- tree_train_fun(train_set, cps[i])
    errors <- rbind(errors, tree_fun(stand_tree_fun, train_set, test_set))
  }
  rownames(errors) <- c("default", "small", "large")
  colnames(errors) <- c("train error", "test error")
  return(errors)
```

```
}

cps <- c(0.01, 0, 0.05)

# standardized dataset
  # play with size of tree
  stand_tree <- cp_test(cps, stand_train_spam, stand_test_spam)
  kable(stand_tree)

set.seed(2987554)
  # perform cross validation and then prune the tree if necessary
  # what does the dashed line mean?
  stand_tree_cv <- tree_train_fun(stand_train_spam, 0.01)
  plotcp(stand_tree_cv)

set.seed(2987554)
##############################################
# creating unbalanced training data
##############################################

  # 3:7 ratio
  classes = lapply(levels(train_spam$class), function(x) which(train_spam$class==x))
  train = lapply(classes, function(class) sample(class, .65*length(class), replace = F))
  train3 <- unlist(train[2])
  class_0 <- subset(train_spam, train_spam$class == 0)
  train_spam37 <- rbind(train_spam[train3,], class_0)

  # 2:8 ratio
  classes = lapply(levels(train_spam$class), function(x) which(train_spam$class==x))
  train = lapply(classes, function(class) sample(class, .38*length(class), replace = F))
  train2 <- unlist(train[2])
  train_spam28 <- rbind(train_spam[train2,], class_0)

  # 1:9 ratio
  classes = lapply(levels(train_spam$class), function(x) which(train_spam$class==x))
  train = lapply(classes, function(class) sample(class, .17*length(class), replace = F))
  train1 <- unlist(train[2])
  train_spam19 <- rbind(train_spam[train1,], class_0)

##############################################
# processed training datasets of unbalanced data
##############################################

######################
# create initial datasets
######################
# 3:7 ratio
  stand_train_spam37 <- as.data.frame(scale(train_spam37[,-58]))
  # add class variable
    stand_train_spam37$class <- train_spam37$class

# 2:8 ratio
  stand_train_spam28 <- as.data.frame(scale(train_spam28[,-58]))
```

```
  # add class variable
    stand_train_spam28$class <- train_spam28$class

# 1:9 ratio
  stand_train_spam19 <- as.data.frame(scale(train_spam19[,-58]))
  # add class variable
    stand_train_spam19$class <- train_spam19$class

##############################################
# creating unbalanced test data
##############################################

  # 3:7 ratio
  classes = lapply(levels(test_spam$class), function(x) which(test_spam$class==x))
  test = lapply(classes, function(class) sample(class, .65*length(class), replace = F))
  test3 <- unlist(test[2])
  class_0 <- subset(test_spam, test_spam$class == 0)
  test_spam37 <- rbind(test_spam[test3,], class_0)

  # 2:8 ratio
  classes = lapply(levels(test_spam$class), function(x) which(test_spam$class==x))
  test = lapply(classes, function(class) sample(class, .38*length(class), replace = F))
  test2 <- unlist(test[2])
  test_spam28 <- rbind(test_spam[test2,], class_0)

  # 1:9 ratio
  classes = lapply(levels(test_spam$class), function(x) which(test_spam$class==x))
  test = lapply(classes, function(class) sample(class, .17*length(class), replace = F))
  test1 <- unlist(test[2])
  test_spam19 <- rbind(test_spam[test1,], class_0)

##############################################
# processed test datasets of unbalanced data
##############################################

# 3:7 ratio
  stand_test_spam37 <- as.data.frame(scale(test_spam37[,-58]))
  # add class variable
    stand_test_spam37$class <- test_spam37$class

# 2:8 ratio
  stand_test_spam28 <- as.data.frame(scale(test_spam28[,-58]))
  # add class variable
    stand_test_spam28$class <- test_spam28$class

# 1:9 ratio
  stand_test_spam19 <- as.data.frame(scale(test_spam19[,-58]))
  # add class variable
    stand_test_spam19$class <- test_spam19$class

set.seed(2987554)
##############################################################################
# perform cross validation on all the training datasets from the unbalanced data
```

```
##############################################################################

# svm

  # standardized
      print("SVM Standardized 3:7 Ratio Data CV Results")
      svm_cv_fun(stand_train_spam37, costs, gammas)

      print("SVM Standardized 2:8 Ratio Data CV Results")
      svm_cv_fun(stand_train_spam28, costs, gammas)

      print("SVM Standardized 1:9 Ratio Data CV Results")
      svm_cv_fun(stand_train_spam19, costs, gammas)

set.seed(2987554)
# neural nets

  # standardized
      nn_stand_train37 <- nn_dataset_fun(stand_train_spam37)
      print("Neural Network Standardized 3:7 Ratio Data CV Results")
      nn_cv_error_plot(nn_stand_train37, grid_n_l)

      nn_stand_train28 <- nn_dataset_fun(stand_train_spam28)
      print("Neural Network Standardized 2:8 Ratio Data CV Results")
      nn_cv_error_plot(nn_stand_train28, grid_n_l)

      nn_stand_train19 <- nn_dataset_fun(stand_train_spam19)
      print("Neural Network Standardized 1:9 Ratio Data CV Results")
      nn_cv_error_plot(nn_stand_train19, grid_n_l)
set.seed(2987554)
# decision trees
  # perform cross validation and then prune the tree if necessary
  # what does the dashed line mean?

  # standardized
    print(paste("Decision Tree Standardized 3:7 Ratio Data CV Results"))
    stand_tree_cv <- tree_train_fun(stand_train_spam37, 0.01)
    plotcp(stand_tree_cv)

    print(paste("Decision Tree Standardized 2:8 Ratio Data CV Results"))
    stand_tree_cv <- tree_train_fun(stand_train_spam28, 0.01)
    plotcp(stand_tree_cv)

    print(paste("Decision Tree Standardized 1:9 Ratio Data CV Results"))
    stand_tree_cv <- tree_train_fun(stand_train_spam19, 0.01)
    plotcp(stand_tree_cv)

set.seed(2987554)
##############################################################################
# create functions to calculate training and test errors using tuning parameters from CV
##############################################################################

colnamesf <- c("train error", "test error")
```

```r
rownamesf <- c("4:6 ratio", "3:7 ratio", "2:8 ratio", "1:9 ratio")

##############################################
# function to calculate svm errors
##############################################

# function to calculate svm training and test errors
svm_error_calc <- function(train_set, test_set, def_kernal, def_cost, def_gamma){

  svm_cv = svm(class ~ ., data=train_set,
                    kernel=def_kernal, cost=def_cost, gamma = def_gamma)

  ### Prediction on train data set
  svm_cv_train_error <- 1-sum(diag(table(train_set$class,
                          predict(svm_cv,train_set))))/
                          nrow(train_set)

  ### Prediction on test data set
  svm_cv_test_error <- 1-sum(diag(table(test_set$class,
                          predict(svm_cv,test_set))))/
                          nrow(test_set)
  return(c(svm_cv_train_error, svm_cv_test_error))
}

##############################################
# function to calculate nn errors
##############################################

# final training and test errors using CV terms neural network function

nn_error_fun <- function(train_set, test_set, nodes) {

  # function to compute the error
    pred = function(nn, dat) {
      yhat = compute(nn, dat)$net.result
      yhat = apply(yhat, 1, which.max)
      yhat <- (yhat == 2)*1
      return(yhat)
    }

  neuralnet_train_cv <- neuralnet(spam_formula, train_set, hidden = nodes, linear.output = F)

  # Calculate training error
    nn_train_pred_cv <- pred(neuralnet_train_cv, train_set[, -c(58,59,60)])
    error_train <- mean(nn_train_pred_cv != train_set$class)

  # Calculate test error
    nn_test_pred_cv <- pred(neuralnet_train_cv, test_set[, -c(58,59,60)])
    error_test <- mean(nn_test_pred_cv != test_set$class)

  error_cv <- cbind(error_train, error_test)

  return(error_cv)
```

```
}

set.seed(2987554)
###############################################
# getting final set of training and test errors, unfixed ratios data
###############################################

# svm

  stand_svm46_errors <- svm_error_calc(stand_train_spam, stand_test_spam,
                         "radial", exp(2), default_gamma)

  stand_svm37_errors <- svm_error_calc(stand_train_spam37, stand_test_spam37,
                         "radial", exp(1), default_gamma)

  stand_svm28_errors <- svm_error_calc(stand_train_spam28, stand_test_spam28,
                         "radial", exp(2), default_gamma)

  stand_svm19_errors <- svm_error_calc(stand_train_spam19, stand_test_spam19,
                         "radial", exp(2), default_gamma)

  stand_svm_errors <- rbind(stand_svm46_errors, stand_svm37_errors,
                          stand_svm28_errors, stand_svm19_errors)
  colnames(stand_svm_errors) <- colnamesf
  rownames(stand_svm_errors) <- rownamesf
  kable(stand_svm_errors)
# nn
  stand_nn46_errors <- nn_error_fun(nn_stand_train_spam, stand_test_spam, 15)
  stand_nn37_errors <- nn_error_fun(nn_stand_train37, stand_test_spam37, c(15,5))
  stand_nn28_errors <- nn_error_fun(nn_stand_train28, stand_test_spam28, 10)
  stand_nn19_errors <- nn_error_fun(nn_stand_train19, stand_test_spam19, c(15,10))

  stand_nn_errors <- rbind(stand_nn46_errors, stand_nn37_errors, stand_nn28_errors, stand_nn19_errors)
  colnames(stand_nn_errors) <- colnamesf
  rownames(stand_nn_errors) <- rownamesf
  kable(stand_nn_errors)



# trees

  tree_train_form <- tree_train_fun(stand_train_spam, 0.016)
  stand_tree46_errors <- tree_fun(tree_train_form, stand_train_spam, stand_test_spam)

  tree_train_form <- tree_train_fun(stand_train_spam37, 0.016)
  stand_tree37_errors <- tree_fun(tree_train_form, stand_train_spam37, stand_test_spam37)

  tree_train_form <- tree_train_fun(stand_train_spam28, 0.018)
  stand_tree28_errors <- tree_fun(tree_train_form, stand_train_spam28, stand_test_spam28)

  tree_train_form <- tree_train_fun(stand_train_spam19, 0.019)
  stand_tree19_errors <- tree_fun(tree_train_form, stand_train_spam19, stand_test_spam19)
```

```
  stand_tree_errors <- rbind(stand_tree46_errors, stand_tree37_errors,
                             stand_tree28_errors, stand_tree19_errors)
  colnames(stand_tree_errors) <- colnamesf
  rownames(stand_tree_errors) <- rownamesf
  kable(stand_tree_errors)


set.seed(2987554)
##############################################
# bootstraps of unbalanced training data
##############################################

  n <- nrow(class_0)

  # fix ratios
  fix_ratios <- function(ratio_data, class_0){
    class1 <- subset(ratio_data, ratio_data$class == 1)
    bootstrap_sample <- lapply(1, function(i) (sample(1:nrow(class1), n, replace = T)))
    bootstrap_sample <- unlist(bootstrap_sample)
    class1_boot <- class1[bootstrap_sample,]
    fixed_data <- rbind(class1_boot, class_0)
    return(fixed_data)

  }

  # 3:7
  stand_train_spamf37 <- fix_ratios(stand_train_spam37, class_0)

  # 2:8
  stand_train_spamf28 <- fix_ratios(stand_train_spam28, class_0)

  # 1:9
  stand_train_spamf19 <- fix_ratios(stand_train_spam19, class_0)


##############################################
# bootstraps of unbalanced test data
##############################################

  # 3:7
  stand_test_spamf37 <- fix_ratios(stand_test_spam37, class_0)

  # 2:8
  stand_test_spamf28 <- fix_ratios(stand_test_spam28, class_0)

  # 1:9
  stand_test_spamf19 <- fix_ratios(stand_test_spam19, class_0)

set.seed(2987554)
############################################################################
# perform cross validation on all the training datasets from the rebalanced data
############################################################################
```

```r
# svm

  # standardized
      print("SVM Standardized 3:7 Ratio Rebalanced Data CV Results")
      svm_cv_fun(stand_train_spamf37, costs, gammas)

      print("SVM Standardized 2:8 Ratio Rebalanced Data CV Results")
      svm_cv_fun(stand_train_spamf28, costs, gammas)

      print("SVM Standardized 1:9 Ratio Rebalanced Data CV Results")
      svm_cv_fun(stand_train_spamf19, costs, gammas)

set.seed(2987554)
# neural nets

  # standardized
      nn_stand_train_spamf37 <- nn_dataset_fun(stand_train_spamf37)
      print("Neural Network Standardized 3:7 Ratio Rebalanced Data CV Results")
      nn_cv_error_plot(nn_stand_train_spamf37, grid_n_l)

      nn_stand_train_spamf28 <- nn_dataset_fun(stand_train_spamf28)
      print("Neural Network Standardized 2:8 Ratio Rebalanced Data CV Results")
      nn_cv_error_plot(nn_stand_train_spamf28, grid_n_l)

      nn_stand_train_spamf19 <- nn_dataset_fun(stand_train_spamf19)
      print("Neural Network Standardized 1:9 Ratio Rebalanced Data CV Results")
      nn_cv_error_plot(nn_stand_train_spamf19, grid_n_l)

set.seed(2987554)
# decision trees
  # perform cross validation and then prune the tree if necessary
  # what does the dashed line mean?

  # standardized
    print(paste("Decision Tree Standardized 3:7 Ratio Rebalanced Data CV Results"))
    stand_tree_cv <- tree_train_fun(stand_train_spamf37, 0.01)
    plotcp(stand_tree_cv)

    print(paste("Decision Tree Standardized 2:8 Ratio Rebalanced Data CV Results"))
    stand_tree_cv <- tree_train_fun(stand_train_spamf28, 0.01)
    plotcp(stand_tree_cv)

    print(paste("Decision Tree Standardized 1:9 Ratio Rebalanced Data CV Results"))
    stand_tree_cv <- tree_train_fun(stand_train_spamf19, 0.01)
    plotcp(stand_tree_cv)


colnamesf <- c("train error", "test error")
rownamesf <- c("3:7 ratio", "2:8 ratio", "1:9 ratio")

set.seed(2987554)
##############################################
# getting final set of training and test errors, fixed ratios data
```

```r
##############################################

# svm

  stand_svm37_errors <- svm_error_calc(stand_train_spamf37, stand_test_spamf37,
                        "radial", exp(2), default_gamma)

  stand_svm28_errors <- svm_error_calc(stand_train_spamf28, stand_test_spamf28,
                        "radial", exp(2), default_gamma)

  stand_svm19_errors <- svm_error_calc(stand_train_spamf19, stand_test_spamf19,
                        "radial", exp(1), default_gamma)

  stand_svm_errors <- rbind(stand_svm37_errors,
                            stand_svm28_errors, stand_svm19_errors)
  colnames(stand_svm_errors) <- colnamesf
  rownames(stand_svm_errors) <- rownamesf
  kable(stand_svm_errors)



# nn
  stand_nn37_errors <- nn_error_fun(nn_stand_train_spamf37, stand_test_spamf37, c(10,10))
  stand_nn28_errors <- nn_error_fun(nn_stand_train_spamf28, stand_test_spamf28, c(5,5))
  stand_nn19_errors <- nn_error_fun(nn_stand_train_spamf19, stand_test_spamf19, 5)

  stand_nn_errors <- rbind(stand_nn37_errors, stand_nn28_errors, stand_nn19_errors)
  colnames(stand_nn_errors) <- colnamesf
  rownames(stand_nn_errors) <- rownamesf
  kable(stand_nn_errors)


# trees

  tree_train_form <- tree_train_fun(stand_train_spamf37, 0.1)
  stand_tree37_errors <- tree_fun(tree_train_form, stand_train_spam37, stand_test_spam37)

  tree_train_form <- tree_train_fun(stand_train_spamf28, 0.1)
  stand_tree28_errors <- tree_fun(tree_train_form, stand_train_spam28, stand_test_spam28)

  tree_train_form <- tree_train_fun(stand_train_spamf19, 0.1)
  stand_tree19_errors <- tree_fun(tree_train_form, stand_train_spam19, stand_test_spam19)

  stand_tree_errors <- rbind(stand_tree37_errors, stand_tree28_errors, stand_tree19_errors)
  colnames(stand_tree_errors) <- colnamesf
  rownames(stand_tree_errors) <- rownamesf
  kable(stand_tree_errors)
```