Student's name: Duong Nguyen

PID: 1646288

Professor: Dr. Christoph Eick

# ASSIGNMENT 3

For this assignment, I decided to choose tictactoe dataset and use Support Vector Machine, Decision Trees and Random Forest.

Even though I would use 10-fold cross validation to train the data and tune the parameters, I saved 15% of the data to test my models and 85% to train the models.

## DECISION TREE:

First, without tuning any parameters, I tried to use ctree on train set and test set to get the "base" tree model, to later compare if tuning parameters would do me any good.

The results I got from simply applying ctree() on the train set then test it on test set:

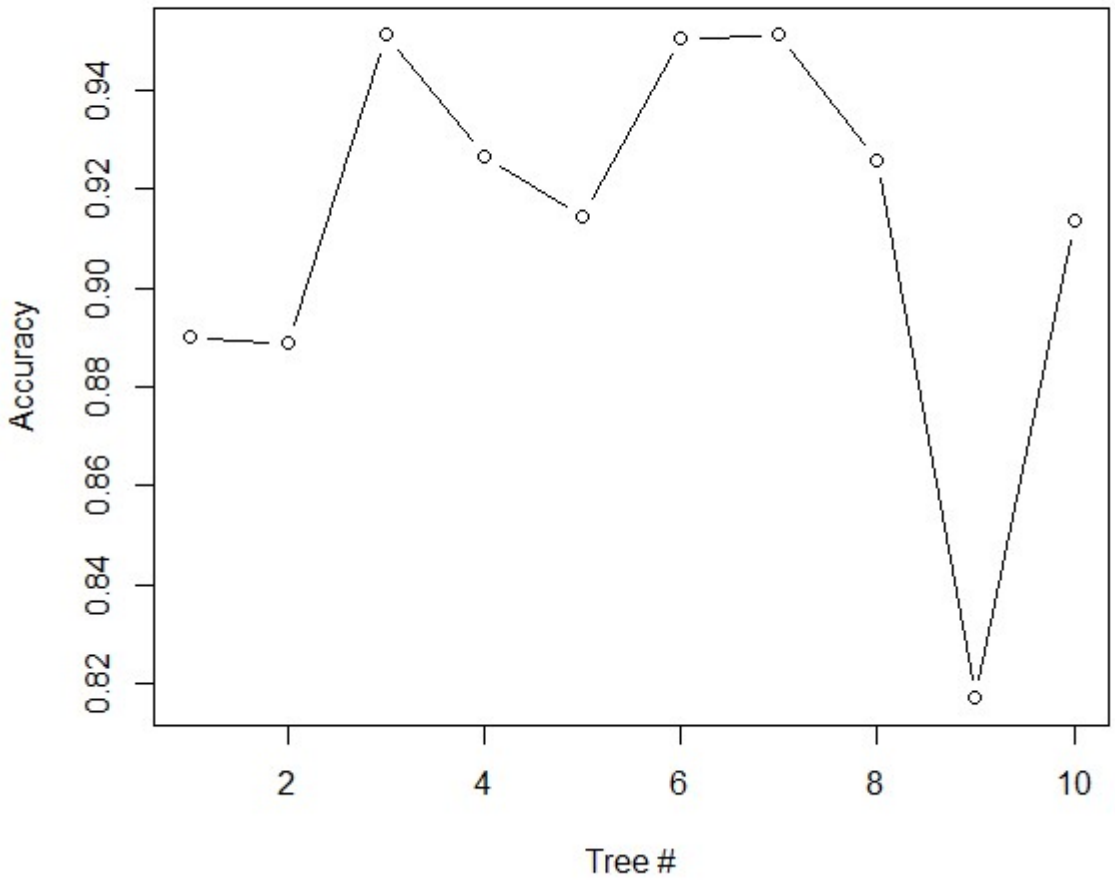
```
Accuracy of tree model on train set =  0.8321078
Accuracy of tree model on test set =  0.7816901
```

As we see from above, the base model does a pretty good job.

I then used train() from package caret to tune the parameters and cross-validate the ctree model. The parameters I chose to tune was mincriterion with the values of 0.2,0.3,0.5,.35, the result I got:

```
Conditional Inference Tree

816 samples
  9 predictor
  2 classes: 'negative', 'positive'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 734, 735, 734, 735, 734, 735, ...
Resampling results across tuning parameters:

  mincriterion  Accuracy   Kappa
  0.20          0.9093646  0.8002875
  0.30          0.9130232  0.8089250
  0.35          0.9130232  0.8089250
  0.50          0.9069256  0.7962769

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mincriterion = 0.35.
```

The accuracy scores among 10 trees built showed in a plot:




On average, the mean accuracy looks much better when compare to the base ctree, in fact, trees number 3, 6 and 7 perform equally well while tree number 3 gave the best performance. The mean accuracy scores:

```
Average accuracy scores of ctree with parameters tuned and 10-fold cross vali
dations:  0.9154472
```

And if I applied the best model after tuning and cross-validating, the accuracies for train set and test set also improved greatly:

```
Accuracy of tree model on train set =  0.9387255
Accuracy of tree model on test set =  0.9014085
```

RANDOM FOREST:

Just like last time, I first ran randomForest model on the train set then applied it on train set as well as test set to get my "base scores" for the model. Below is the results I got:

```
    Number of trees: 500
No. of variables tried at each split: 3
```

```
        OOB estimate of  error rate: 1.47%
Confusion matrix:
         negative positive class.error
negative      272       11 0.038869258
positive        1      532 0.001876173
```

```
Accuracy of Random Forest model on train set =  1
Accuracy of Random Forest model on test set =  0.9929577
```

Wow! Random forest achieved perfect accuracy on train set and almost perfect on test set. One may think we should stop here and use the model since it is perfect, so probably there is no need for parameters tuning. However, the random forest model above is executed with all the default parameters (500 trees and 3 mtry each time splitting the nodes). Perhaps if we do parameters tuning and cross validations, we can find "forests" with less trees that can provide the similar accuracy. And it turns out, there are better forests out there which are less computationally expensive after model was tuned.

I could have used the train() function from caret package, however I did not do so because the tuneGrid supplied to train() only allows the model to tune mtry but not ntree, therefore, to avoid doing all the tuning manually, I instead chose to tune my parameters and got the "best" model using the best.tune() function from e1071 package which allows me to tune both mtry and ntree, and carry out 10-fold cross validations as well. The model achieved the results below:

```
   Number of trees: 20
No. of variables tried at each split: 9

        OOB estimate of  error rate: 3.68%
Confusion matrix:
         negative positive class.error
negative      267       16  0.05653710
positive       14      519  0.02626642
```

The OOB(out-of-bag) error rate is pretty low. 3.68% is the OOB error rate produced by the "best" forest among the 10 forests the cross-validation gave. The average accuracy for 10 forests is:

```
Mean accuracy scores of Random Forest model with tuned parameters and cross-v
alidation:  0.9294423
```

If we applied this to train set and test set, we'll get:

```
Accuracy of Random Forest model on train set =  1
Accuracy of random forest model on test set =  0.9859155
```

If we look closely, the number of trees used is only 20, significantly less than 500 trees used in the original untuned random forest model. Therefore, even though it took a little bit longer to tune and train the model, at the end, if we chose to use the "best" forest we got after tuning, the chosen model would take much less time to run later because the forest is much smaller.

## SUPPORT VECTOR MACHINE:

Before running the SVM algorithm, I converted all character data points into numeric by binarizing them, since SVM does not work with all data types but numeric.

And once again, I started with training my model using all default parameters to see what “base” SVM model could give me. I got the below result:

```
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  radial
       cost:  1
      gamma:  0.03703704

Number of Support Vectors:  533
```

```
Accuracy of SVM model on train set =  0.9914216
Accuracy of SVM model on test set =  0.9859155
```

SVM with default parameters gave very good results!

However, let’s try tuning the parameters and test the model using cross-validation.

I used best.tune() to figure out which kernel would give me the most possible optimal result since train() function asks for known method (either svmPoly, svmLinear,etc) beforehand. It turned out Polynomial kernel would likely to give me the best result.

Next, I used train() and passed “svmPoly” to its method and let the function tune the other parameters. I got a surprisingly good results when apply the model on train set and test set. Both achieve accuracy of 1!

```
Accuracy of SVM model on train set =  1
Accuracy of SVM model on train set =  1
```

With 10-fold cross validation:

```
Average accuracy of SVM with tuned parameters =  0.9963415
Standard deviation of SVM model with tuned parameters =  0.00823108
```

As we see, the standard deviation of all the models is really small, implying that the models are really stable. On average, SVM with tuned parameters would predict correctly 99.6% of the time.

To wrap it up, using varImp() function which returns the importance of attributes in each model, I wanted to see which attributes play the most important role in predicting the outcome:

```
> varImp(best_tree)
ROC curve variable importance

              Importance
middle_middle    100.000
top_middle        33.280
middle_left       26.066
```

```
bot_middle        21.795
middle_right      19.833
top_left           7.678
top_right          3.413
bot_right          2.429
bot_left           0.000
```

```
> varImp(best_rf)
               Overall
top_left      46.27299
top_middle    38.86541
top_right     40.66412
middle_left   32.27527
middle_middle 53.12147
middle_right  30.27880
bot_left      46.72197
bot_middle    37.91884
bot_right     45.62329
```

```
> varImp(best_svm)
ROC curve variable importance

  only 20 most important variables shown (out of 27)

                Importance
middle_middle.o     100.00
middle_middle.x      97.11
top_left.o           42.27
bot_left.o           34.89
bot_right.o          34.83
top_right.o          33.50
top_middle.x         31.98
top_left.x           31.59
middle_left.x        29.42
middle_right.x       28.87
bot_middle.x         26.93
bot_right.x          25.79
top_right.x          25.56
bot_left.x           24.40
middle_right.o       20.17
top_middle.b         16.98
middle_left.o        14.25
bot_middle.o         13.21
middle_left.b        12.29
top_middle.o         12.11
```

When inspecting the importance of attributes, we saw that the most important attributes is the one in the central.

A quick few lines of code shows that when ‘x’ is in placed in the center, the result is likely “positive” and negative if ‘o’ is placed in the center

```
> table(tictactoe[tictactoe$result == 'positive', 'middle_middle'])

  b   o   x
112 148 366
> table(tictactoe[tictactoe$result == 'negative', 'middle_middle'])

  b   o   x
 48 192  92
```

In summary, as I observed, SVM took the longest to train while ctree took the shortest time to train, obviously because ctree is just a single decision tree, while SVM has to tune many parameters (type of kernel, gamma, C, epsilon,etc). Random Forest outperforms regular tree algorithm because it is more robust because it does not suffer from instability like a single decision tree. SVM worked the best for this dataset at the end. The reason is SVM usually works well when it comes to classify between two classes, plus, sparse data set (18 out of 27 predictors are 0’s since all the attributes were binarized) is also a reason why SVM outperforms other models. Also, if it would leave up to me to decide, while the problem is binary classification and the dataset is sparse and not too large, I would most likely to choose SVM as my best model.