**Final**

Problem 1

1. Use regression trees to construct the predictive model.
   1. Split the data set into training and test sets. Use the partition obtained to answer all questions of problem 1.This step should be done once and used on the different models. This means that you should use the same training and test sets to construct your classifiers. Hence, it will be easier to compare the models.

First I imported the Superconductivity Data Set using the url of the file. I then imported train\_test\_split from sklearn.model\_selection. Using this command I was able to split the training and testing set using a 75% split. See code below:

# -\*- coding: utf-8 -\*-

"""

Created on Mon Jun 21 16:17:22 2021

@author: 16319

"""

import pandas as pd

import numpy as np

import matplotlib as mpl

import matplotlib.pyplot as plt

import graphviz

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier, export\_graphviz

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from sklearn.metrics import confusion\_matrix, mean\_squared\_error, classification\_report

from sklearn.preprocessing import scale

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/00464/superconduct.zip"

data = pd.read\_csv('train.csv')

data = data.dropna()

data.astype('int64').dtypes

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import KFold

X = scale(data.iloc[:,0:81])

y = scale(data.iloc[:,81].astype('int64'))

X\_train,X\_test,y\_train,y\_test=train\_test\_split(X,y,test\_size=0.25, random\_state=0)

* 1. Use k-fold cross validation on the training set to select the best value(s)of the tuning parameter(s) needed.

To do this I imported GridSearchCV from sklearn.model\_selection and DecisionTreeRegressor from sklearn.tree. I set the tuning parameters equal to max\_depth and min\_samples\_split. I used 10 folds using neg\_mean\_squared\_error as the scoring metric. See code below:

max\_depth\_range = np.arange(1,11,1)

mss\_range = np.arange(2,11,1)

tuned\_parameters\_lr = [{'max\_depth': max\_depth\_range,

'min\_samples\_split': mss\_range, }]

lr = GridSearchCV(DecisionTreeRegressor(), tuned\_parameters\_lr, cv=KFold(n\_splits=10), scoring = 'neg\_mean\_squared\_error')

lr.fit(X\_train.astype('int64'), y\_train)

print(lr.best\_params\_)

Results:

{'max\_depth': 10, 'min\_samples\_split': 6}

* 1. Report the performance of each model on the training set, by reporting the training MSE or RMSE.

Once I had the best parameters I used them to fit my model and make predictions. Below are the results of this model on the training set. See code below:

linear = DecisionTreeRegressor(max\_depth = 10, min\_samples\_split=6)

linear.fit(X\_train.astype('int64'),y\_train)

y\_pred\_lr\_train = linear.predict(X\_train)

from sklearn.metrics import mean\_squared\_error

print("MSE train:", mean\_squared\_error(y\_train,y\_pred\_lr\_train))

Results:

MSE train: 1166.2104074791698

* 1. Report the performance of each model on the testing set, by reporting the training MSE or RMSE.

Using the model fitted using the training data I predicted the values of the test set. See code below:

y\_pred\_lr\_test = linear.predict(X\_test)

print("MSE test:", mean\_squared\_error(y\_test,y\_pred\_lr\_test))

Results:

MSE test: 3489.924348706606

1. Use bagging regression to construct the predictive model. Report also the bagging important variables
   1. NA
   2. Use k-fold cross validation on the training set to select the best value(s)of the tuning parameter(s) needed.

I used GridSearchCV again to choose the best tuning paramaters. I had the command search the values of max\_features, max\_samples, and n\_estimators using 10 folds. This was scored by neg\_mean\_sqaured\_error. See code below:

n\_est\_range = np.arange(10,21,1)

max\_samples\_range = np.arange(1,11,1)

max\_features\_range = np.arange(1,11,1)

tuned\_parameters\_lrbag = [{ 'n\_estimators': n\_est\_range,

'max\_features': max\_features\_range,

'max\_samples\_range': max\_samples\_range

}]

lr\_bag = GridSearchCV(BaggingRegressor(), tuned\_parameters\_lrbag, cv=KFold(n\_splits=10), scoring = 'neg\_mean\_squared\_error')

lr\_bag.fit(X\_train.astype('int64'),y\_train)

print(lr\_bag.best\_params\_)

Results:

{'max\_features': 9, 'max\_samples': 10, 'n\_estimators': 18}

* 1. Report the performance of each model on the training set, by reporting the training MSE or RMSE.

Using these parameters I fit the model using the training set and predicted the values based on the training set predictors. See code below:

linear\_bag = BaggingRegressor(max\_features = 9, max\_samples = 10, n\_estimators = 18)

linear\_bag.fit(X\_train,y\_train)

y\_pred\_lr\_bag\_train = linear\_bag.predict(X\_train)

print("MSE train:", mean\_squared\_error(y\_train,y\_pred\_lr\_bag\_train))

Results:

MSE train: 607.9542514285362

* 1. Report the performance of each model on the testing set, by reporting the training MSE or RMSE.

I repeated this process using the model fitted on the training set in order to predict the values based on the predictors in the test set. See code below:

y\_pred\_lr\_bag\_test = linear\_bag.predict(X\_test)

print("MSE test:", mean\_squared\_error(y\_test,y\_pred\_lr\_bag\_test))

Results:

MSE test: 1462.7433405514475

* 1. Report the important variables

To plot the important variables I first had to calculate the important variables using the feature\_importances\_ attribute of the bagging model. Then I was able to plot by sorting the values by Importance, by row, and by ascending order. See code below (many variables so a cluttered graph was created):

Importance\_linear\_bag = pd.DataFrame({'Importance': linear.feature\_importances\_\*100})

Importance\_linear\_bag.sort\_values(by = 'Importance',

axis = 0,

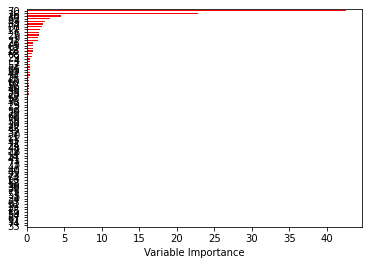
ascending = True). plot(kind = 'barh',

color = 'r', )

plt.xlabel('Variable Importance')

plt.gca().legend\_ = None

Results:



1. Use random forest regression to construct the predictive model. Report also the random forest important variables.
   1. NA
   2. Use k-fold cross validation on the training set to select the best value(s)of the tuning parameter(s) needed.

Using GridSearchCV and RandomForestRegressor I was able to run a cross validation on the parameters max\_features and n\_estimators. See code below:

n\_est\_rf\_range = np.arange(10,100,10)

tuned\_parameters\_lrrf = [{'max\_features': max\_features\_range,

'n\_estimators': n\_est\_rf\_range

}]

lr\_rf = GridSearchCV(RandomForestRegressor(), tuned\_parameters\_lrrf, cv=KFold(n\_splits=3), scoring = 'neg\_mean\_squared\_error')

lr\_rf.fit(X\_train.astype('int64'),y\_train)

print(lr\_rf.best\_params\_)

Results:

{'max\_features': 4, 'n\_estimators': 90}

* 1. Report the performance of each model on the training set, by reporting the training MSE or RMSE.

Using these parameters I was able to fit the model using the training set predictors and then ran a prediction based on the training set data. See code below:

linear\_rf = RandomForestRegressor(max\_features = 4, n\_estimators = 90)

linear\_rf.fit(X\_train,y\_train)

y\_pred\_lr\_rf\_train = linear\_rf.predict(X\_train)

print("MSE train:", mean\_squared\_error(y\_train,y\_pred\_lr\_rf\_train))

Results:

MSE train: 24.814814445015234

* 1. Report the performance of each model on the testing set, by reporting the training MSE or RMSE.

Using the model fitted from the training data I predicted the values based on the testing set predictors. See code below:

y\_pred\_lr\_rf\_test = lr\_rf.predict(X\_test)

print("MSE test:", mean\_squared\_error(y\_test,y\_pred\_lr\_rf\_test))

Results:

MSE test: 2263.6692258176754

* 1. Report variable importance.

Using the feature\_importances\_ attribute I was able to produce the variable importance on the predictors of this data set. See code below:

Importance\_linear\_rf = pd.DataFrame({'Importance': linear\_rf.feature\_importances\_\*100})

Importance\_linear\_rf.sort\_values(by = 'Importance',

axis = 0,

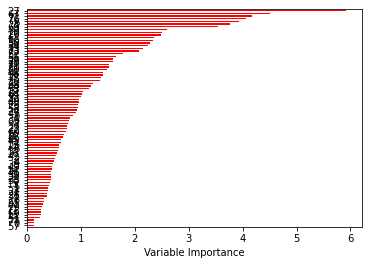
ascending = True). plot(kind = 'barh',

color = 'r', )

plt.xlabel('Variable Importance')

plt.gca().legend\_ = None

Results:



1. Use boosting regression to construct the predictive model. Report alsothe boosting important variables.
   1. NA
   2. Use k-fold cross validation on the training set to select the best value(s)of the tuning parameter(s) needed.

Using GridSearchCV and GradientBoostingRegressor functions I was able to test the parameters learning\_rate and n\_estimators. See code below:

learning\_rate\_range = np.arange(0.1,1,0.1)

tuned\_parameters\_lrboost = [{ 'n\_estimators': n\_est\_range,

'learning\_rate': learning\_rate\_range

}]

lr\_boost = GridSearchCV(GradientBoostingRegressor(), tuned\_parameters\_lrboost, cv=KFold(n\_splits=3), scoring = 'neg\_mean\_squared\_error')

lr\_boost.fit(X\_train.astype('int64'),y\_train)

print(lr\_boost.best\_params\_)

Results:

{'learning\_rate': 0.7000000000000001, 'n\_estimators': 20}

* 1. Report the performance of each model on the training set, by reporting the training MSE or RMSE.

Using these parameters I was able to fit the model based on the training set data and then predicted the response values based on the training data. See code below:

linear\_boost = GradientBoostingRegressor(learning\_rate = 0.7, n\_estimators = 20)

linear\_boost.fit(X\_train,y\_train)

y\_pred\_lr\_boost\_train = linear\_boost.predict(X\_train)

print("MSE train:", mean\_squared\_error(y\_train,y\_pred\_lr\_boost\_train))

Results:

MSE train: 147.73998468715249

* 1. Report the performance of each model on the testing set, by reporting the training MSE or RMSE.

Using the model fitted using the training data I predicted the values based off of the predictors in the testing set. See code below:

y\_pred\_lr\_boost\_test = linear\_boost.predict(X\_test)

print("MSE test:", mean\_squared\_error(y\_test,y\_pred\_lr\_boost\_test))

Results:

MSE test: 2142.1142175495447

* 1. Report the variable importance.

Using the feature\_importances\_ attribute I was able to produce the variable importance on the predictors of this data set. See code below:

Importance\_linear\_boost = pd.DataFrame({'Importance': linear\_boost.feature\_importances\_\*100})

Importance\_linear\_boost.sort\_values(by = 'Importance',

axis = 0,

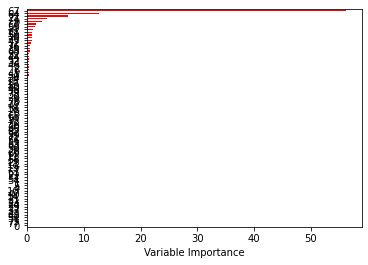
ascending = True). plot(kind = 'barh',

color = 'r', )

plt.xlabel('Variable Importance')

plt.gca().legend\_ = None

Results:



1. Use linear regression to construct the predictive model.
   1. NA
   2. NA
   3. Report the performance of each model on the training set, by reporting the training MSE or RMSE.

Next I fit the model for the training set using linear regression. Once the model was fit I predicted the values based off the training set predictors. See code below:

linreg = LinearRegression()

linreg.fit(X\_train,y\_train)

y\_pred\_linreg\_train = linreg.predict(X\_train)

print("MSE train:", mean\_squared\_error(y\_train,y\_pred\_linreg\_train))

Results:

MSE train: 307.79638108324053

* 1. Report the performance of each model on the testing set, by reporting the training MSE or RMSE.

Using the model fitted to the training set I predicted the values based off the predictors in the testing set. See code below:

y\_pred\_linreg\_test = linreg.predict(X\_test)

print("MSE test:", mean\_squared\_error(y\_test,y\_pred\_linreg\_test))

Results:

MSE test: 2013.0746173758412

1. Using your predictive models from previous questions above, report thebest predictive model for this dataset. The best predictive model is theone with the smallest training and test errors.

Now that I have calculated the MSE training and testing for each prediciton model. I am able to get the average between the training and testing for each prediciton model. Based off of these averages the Random Forest model produced the most accurate predictions. See code below:

RT\_mse\_avg = (RT\_trainmse + RT\_testmse)/2

LB\_mse\_avg = (LB\_trainmse + LB\_testmse)/2

RF\_mse\_avg = (RF\_trainmse + RF\_testmse)/2

Boost\_mse\_avg = (Boost\_trainmse + Boost\_testmse)/2

lr\_mse\_avg = (lr\_trainmse + lr\_testmse)/2

print("Regression Tree MSE Avg: ", RT\_mse\_avg)

print("Bagging MSE Avg", LB\_mse\_avg)

print("Random Forest MSE Avg", RF\_mse\_avg)

print("Boosting MSE Avg", Boost\_mse\_avg)

print("Linear Regression MSE Avg", lr\_mse\_avg)

Results:

Regression Tree MSE Avg: 2476.3700538932026

Bagging MSE Avg 1254.9433204104066

Random Forest MSE Avg 1144.272213633246

Boosting MSE Avg 1144.7487670157545

Linear Regression MSE Avg 1160.435499229541

**Problem 2**

1. Use classification trees to construct your classifier.

To split the dataset I imported train\_test\_split and the breast cancer data set. I used a 75% split. See code below:

# -\*- coding: utf-8 -\*-

"""

Created on Mon Jun 21 16:47:01 2021

@author: 16319

"""

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import auc, roc\_curve

from sklearn import datasets

from sklearn.metrics import confusion\_matrix, classification\_report, precision\_score

from sklearn.model\_selection import GridSearchCV

from sklearn.svm import SVC

from sklearn import metrics

from sklearn import preprocessing

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import KFold

data\_set = datasets.load\_breast\_cancer()

X= preprocessing.scale(data\_set.data)

y= data\_set.target

X\_train,X\_test,y\_train,y\_test=train\_test\_split(X,y,test\_size=0.25, random\_state=0)

First I imported GridSearchCV and DecisionTreeClassifier. Using these I was able to test the max\_depth and min\_sample\_split parameters. See code below:

max\_depth\_range = np.arange(1,11,1)

mss\_range = np.arange(2,11,1)

tuned\_parameters\_lr = [{'max\_depth': max\_depth\_range,

'min\_samples\_split': mss\_range, }]

lr = GridSearchCV(DecisionTreeClassifier(), tuned\_parameters\_lr, cv=KFold(n\_splits=3), scoring = 'accuracy')

lr.fit(X\_train.astype('int64'), y\_train)

print(lr.best\_params\_)

Results:

{'max\_depth': 9, 'min\_samples\_split': 5}

Using these parameters I was able to fit the data using the training set and then I made my prediction based off of the training set. Once I had the predictions I was able to calculate the misclassification error rate for the training predictions. See code below:

linear = DecisionTreeClassifier(max\_depth = 9, min\_samples\_split=5)

linear.fit(X\_train.astype('int64'),y\_train)

y\_pred\_lr\_train = linear.predict(X\_train)

ct\_mer\_train = 1 - metrics.accuracy\_score(y\_train,y\_pred\_lr\_train)

conf\_\_lr = pd.DataFrame(metrics.confusion\_matrix(y\_train, y\_pred\_lr\_train))

TP\_lr = conf\_\_lr.iloc[0,0]

FP\_lr = conf\_\_lr.iloc[1,0]

FN\_lr = conf\_\_lr.iloc[0,1]

TN\_lr = conf\_\_lr.iloc[1,1]

print("Misclassification Error Rate", ct\_mer\_train)

Results:

Misclassification Error Rate 0.10563380281690138

To produce the confusion matrix for training and testing I used the confusion\_matrix command from sklearn.metrics. Using the values from the confusion matrix I was able to calculate the corresponding metrics. This includes the testing misclassification error rate. See code below:

print("Training Confusion Matrix \n", metrics.confusion\_matrix(y\_train, y\_pred\_lr\_train))

print("Recall:", TP\_lr/(TP\_lr+FN\_lr))

print("Specificity:", TN\_lr/(TN\_lr+FP\_lr))

print("Fallout:", FP\_lr/(FP\_lr + TN\_lr))

print("PPV:", TP\_lr/(TP\_lr + FP\_lr))

print("Accuracy:", (TP\_lr + TN\_lr)/(TP\_lr +TN\_lr + FP\_lr + FN\_lr))

y\_pred\_lr\_test = linear.predict(X\_test)

ct\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_lr\_test)

print("Misclassification Error Rate", ct\_mer\_test)

print("Testing Confusion Matrix \n", metrics.confusion\_matrix(y\_test, y\_pred\_lr\_test))

ct\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_lr\_test)

conf\_\_lr1 = pd.DataFrame(metrics.confusion\_matrix(y\_test, y\_pred\_lr\_test))

TP\_lr1 = conf\_\_lr1.iloc[0,0]

FP\_lr1 = conf\_\_lr1.iloc[1,0]

FN\_lr1 = conf\_\_lr1.iloc[0,1]

TN\_lr1 = conf\_\_lr1.iloc[1,1]

print("Misclassification Error Rate", ct\_mer\_test)

print("Recall:", TP\_lr1/(TP\_lr1+FN\_lr1))

print("Specificity:", TN\_lr1/(TN\_lr1+FP\_lr1))

print("Fallout:", FP\_lr1/(FP\_lr1 + TN\_lr1))

print("PPV:", TP\_lr1/(TP\_lr1 + FP\_lr1))

print("Accuracy:", (TP\_lr1 + TN\_lr1)/(TP\_lr1 +TN\_lr1 + FP\_lr1 + FN\_lr1))

Results:

Training Confusion Matrix

[[154 5]

[ 40 227]]

Recall: 0.9685534591194969

Specificity: 0.850187265917603

Fallout: 0.149812734082397

PPV: 0.7938144329896907

Accuracy: 0.8943661971830986

Misclassification Error Rate 0.1398601398601399

Testing Confusion Matrix

[[49 4]

[16 74]]

Misclassification Error Rate 0.1398601398601399

Recall: 0.9245283018867925

Specificity: 0.8222222222222222

Fallout: 0.17777777777777778

PPV: 0.7538461538461538

Accuracy: 0.8601398601398601

1. Use bagging classification to construct your classifier. Report also the bagging important variables.
   1. NA
   2. .

Using GridSearchCV and BaggingClassifier() I was able to test the parameters n\_estimators, max\_samples, and max\_features. See code below:

n\_est\_range = np.arange(1,101,10)

max\_sample\_range = np.arange(1,10,1)

max\_features\_range = np.arange(1,10,1)

tuned\_parameters\_bag = [{ 'n\_estimators': n\_est\_range,

'max\_samples': max\_sample\_range,

'max\_features': max\_features\_range

}]

bag = GridSearchCV(BaggingClassifier(), tuned\_parameters\_bag, cv=KFold(n\_splits=3), scoring = 'accuracy')

bag.fit(X\_train.astype('int64'), y\_train)

print(bag.best\_params\_)

Results:

{'max\_features': 7, 'max\_samples': 9, 'n\_estimators': 51}

* 1. .
  2. .
  3. .

Using these parameters I was able to fit the prediction model based off of the training set data. Once this was fit properly I was able to make predictions on both the training and testing sets. Once these predictions were made I was able to produce the confusion matrix and corresponding metrics as well as the misclassification error rate. See code below:

bagging = BaggingClassifier(max\_features = 7, max\_samples = 9, n\_estimators = 51)

bagging.fit(X\_train.astype('int64'),y\_train)

y\_pred\_bag\_train = bagging.predict(X\_train)

bag\_mer\_train = 1 - metrics.accuracy\_score(y\_train,y\_pred\_bag\_train)

conf\_\_bag = pd.DataFrame(metrics.confusion\_matrix(y\_train, y\_pred\_bag\_train))

TP\_bag = conf\_\_bag.iloc[0,0]

FP\_bag = conf\_\_bag.iloc[1,0]

FN\_bag = conf\_\_bag.iloc[0,1]

TN\_bag = conf\_\_bag.iloc[1,1]

print("Misclassification Error Rate", bag\_mer\_train)

print("Training Confusion Matrix \n", metrics.confusion\_matrix(y\_train, y\_pred\_bag\_train))

print("Recall:", TP\_bag/(TP\_bag+FN\_bag))

print("Specificity:", TN\_bag/(TN\_bag+FP\_bag))

print("Fallout:", FP\_bag/(FP\_bag + TN\_bag))

print("PPV:", TP\_bag/(TP\_bag + FP\_bag))

print("Accuracy:", (TP\_bag + TN\_bag)/(TP\_bag +TN\_bag + FP\_bag + FN\_bag))

y\_pred\_bag\_test = bagging.predict(X\_test)

bag\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_bag\_test)

print("Misclassification Error Rate", bag\_mer\_test)

print("Testing Confusion Matrix \n", metrics.confusion\_matrix(y\_test, y\_pred\_bag\_test))

bag\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_bag\_test)

conf\_\_bag1 = pd.DataFrame(metrics.confusion\_matrix(y\_test, y\_pred\_bag\_test))

TP\_bag1 = conf\_\_bag1.iloc[0,0]

FP\_bag1 = conf\_\_bag1.iloc[1,0]

FN\_bag1 = conf\_\_bag1.iloc[0,1]

TN\_bag1 = conf\_\_bag1.iloc[1,1]

print("Misclassification Error Rate", bag\_mer\_test)

print("Recall:", TP\_bag1/(TP\_bag1+FN\_bag1))

print("Specificity:", TN\_bag1/(TN\_bag1+FP\_bag1))

print("Fallout:", FP\_bag1/(FP\_bag1 + TN\_bag1))

print("PPV:", TP\_bag1/(TP\_bag1 + FP\_bag1))

print("Accuracy:", (TP\_bag1 + TN\_bag1)/(TP\_bag1 +TN\_bag1 + FP\_bag1 + FN\_bag1))

Results:

Misclassification Error Rate 0.07276995305164324

Training Confusion Matrix

[[129 30]

[ 1 266]]

Recall: 0.8113207547169812

Specificity: 0.9962546816479401

Fallout: 0.003745318352059925

PPV: 0.9923076923076923

Accuracy: 0.9272300469483568

Testing Confusion Matrix

[[43 10]

[ 0 90]]

Misclassification Error Rate 0.0699300699300699

Recall: 0.8113207547169812

Specificity: 1.0

Fallout: 0.0

PPV: 1.0

Accuracy: 0.9300699300699301

f.

Using the feature\_importances attribute I was able to produce the variable importance on this model. See code below:

Importance\_bagging = pd.DataFrame({'Importance': linear.feature\_importances\_\*100},

index=X.columns)

Importance\_bagging.sort\_values(by = 'Importance',

axis = 0,

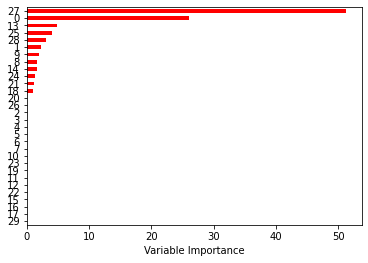
ascending = True). plot(kind = 'barh',

color = 'r', )

plt.xlabel('Variable Importance')

plt.gca().legend\_ = None

Results:



1. Use random forest classification to construct your classifier. Report also the random forest important variables.
   1. NA
   2. .

Using GridSearchCV and and RandomForestClassifier I was able to test the parameters n\_estimators, max\_depth, and min\_samples\_split. See code below:

n\_estim\_range = np.arange(80,120,10)

max\_depth\_range = np.arange(1,10,1)

min\_samples\_range = np.arange(2,10,1)

tuned\_parameters\_rf = [{ 'n\_estimators': n\_estim\_range,

'max\_depth': max\_depth\_range,

'min\_samples\_split': min\_samples\_range

}]

rf = GridSearchCV(RandomForestClassifier(), tuned\_parameters\_rf, cv=KFold(n\_splits=3), scoring = 'accuracy')

rf.fit(X\_train.astype('int64'), y\_train)

print(rf.best\_params\_)

Results:

{'max\_depth': 8, 'min\_samples\_split': 3, 'n\_estimators': 80}

* 1. .
  2. .
  3. .

Using these parameters I was able to fit the prediction model based off of the training set data. Once this was fit properly I was able to make predictions on both the training and testing sets. Once these predictions were made I was able to produce the confusion matrix and corresponding metrics as well as the misclassification error rate. See code below:

rforest = RandomForestClassifier(max\_depth = 8, min\_samples\_split = 3, n\_estimators = 80)

rforest.fit(X\_train.astype('int64'),y\_train)

y\_pred\_rf\_train = rforest.predict(X\_train)

rf\_mer\_train = 1 - metrics.accuracy\_score(y\_train,y\_pred\_rf\_train)

conf\_\_rf = pd.DataFrame(metrics.confusion\_matrix(y\_train, y\_pred\_rf\_train))

TP\_rf = conf\_\_rf.iloc[0,0]

FP\_rf = conf\_\_rf.iloc[1,0]

FN\_rf = conf\_\_rf.iloc[0,1]

TN\_rf = conf\_\_rf.iloc[1,1]

print("Misclassification Error Rate", rf\_mer\_train)

print("Training Confusion Matrix \n", metrics.confusion\_matrix(y\_train, y\_pred\_rf\_train))

print("Recall:", TP\_rf/(TP\_rf+FN\_rf))

print("Specificity:", TN\_rf/(TN\_rf+FP\_rf))

print("Fallout:", FP\_rf/(FP\_rf + TN\_rf))

print("PPV:", TP\_rf/(TP\_rf + FP\_rf))

print("Accuracy:", (TP\_rf + TN\_rf)/(TP\_rf +TN\_rf + FP\_rf + FN\_rf))

y\_pred\_rf\_test = rforest.predict(X\_test)

rf\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_rf\_test)

print("Misclassification Error Rate", rf\_mer\_test)

print("Testing Confusion Matrix \n", metrics.confusion\_matrix(y\_test, y\_pred\_rf\_test))

rf\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_rf\_test)

conf\_\_rf1 = pd.DataFrame(metrics.confusion\_matrix(y\_test, y\_pred\_rf\_test))

TP\_rf1 = conf\_\_rf1.iloc[0,0]

FP\_rf1 = conf\_\_rf1.iloc[1,0]

FN\_rf1 = conf\_\_rf1.iloc[0,1]

TN\_rf1 = conf\_\_rf1.iloc[1,1]

print("Misclassification Error Rate", rf\_mer\_test)

print("Recall:", TP\_rf1/(TP\_rf1+FN\_rf1))

print("Specificity:", TN\_rf1/(TN\_rf1+FP\_rf1))

print("Fallout:", FP\_rf1/(FP\_rf1 + TN\_rf1))

print("PPV:", TP\_rf1/(TP\_rf1 + FP\_rf1))

print("Accuracy:", (TP\_rf1 + TN\_rf1)/(TP\_rf1 +TN\_rf1 + FP\_rf1 + FN\_rf1))

Results:

Misclassification Error Rate 0.05868544600938963

Training Confusion Matrix

[[148 11]

[ 14 253]]

Recall: 0.9308176100628931

Specificity: 0.947565543071161

Fallout: 0.052434456928838954

PPV: 0.9135802469135802

Accuracy: 0.9413145539906104

Misclassification Error Rate 0.0699300699300699

Testing Confusion Matrix

[[48 5]

[ 5 85]]

Misclassification Error Rate 0.0699300699300699

Recall: 0.9056603773584906

Specificity: 0.9444444444444444

Fallout: 0.05555555555555555

PPV: 0.9056603773584906

Accuracy: 0.9300699300699301

* 1. .

Using the feature\_importances attribute I was able to produce the variable importance on this model. See code below:

Importance\_rforest = pd.DataFrame({'Importance': rforest.feature\_importances\_\*100})

Importance\_rforest.sort\_values(by = 'Importance',

axis = 0,

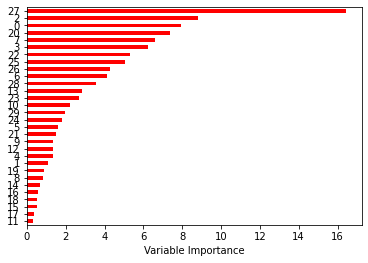
ascending = True). plot(kind = 'barh',

color = 'r', )

plt.xlabel('Variable Importance')

plt.gca().legend\_ = None

Results:



1. Use boosting classification to construct your classifier. Report also the boosting important variables.
   1. NA
   2. .

Using GridSearchCV and GradientBoostingClassifier I was able to test the parameters learn\_rate, subsample, and n\_estimators. See code below:

learn\_rate\_range = np.arange(0.1,1,.1)

subsample\_range = np.arange(0.1,1,.1)

tuned\_parameters\_boost = [{ 'learning\_rate': learn\_rate\_range,

'n\_estimators': n\_estim\_range,

'subsample': subsample\_range

}]

boost = GridSearchCV(GradientBoostingClassifier(), tuned\_parameters\_boost, cv=KFold(n\_splits=3), scoring = 'accuracy')

boost.fit(X\_train.astype('int64'), y\_train)

print(boost.best\_params\_)

Results:

{'max\_depth': 8, 'min\_samples\_split': 3, 'n\_estimators': 80}

* 1. .
  2. .
  3. .

Using these parameters I was able to fit the prediction model based off of the training set data. Once this was fit properly I was able to make predictions on both the training and testing sets. Once these predictions were made I was able to produce the confusion matrix and corresponding metrics as well as the misclassification error rate. See code below:

boosting = GradientBoostingClassifier(learning\_rate = 0.1, n\_estimators = 90, subsample = 0.30000000000000004)

boosting.fit(X\_train.astype('int64'),y\_train)

y\_pred\_boost\_train = boosting.predict(X\_train)

boost\_mer\_train = 1 - metrics.accuracy\_score(y\_train,y\_pred\_boost\_train)

conf\_\_boost = pd.DataFrame(metrics.confusion\_matrix(y\_train, y\_pred\_boost\_train))

TP\_boost = conf\_\_boost.iloc[0,0]

FP\_boost = conf\_\_boost.iloc[1,0]

FN\_boost = conf\_\_boost.iloc[0,1]

TN\_boost = conf\_\_boost.iloc[1,1]

print("Misclassification Error Rate", boost\_mer\_train)

print("Training Confusion Matrix \n", metrics.confusion\_matrix(y\_train, y\_pred\_boost\_train))

print("Recall:", TP\_boost/(TP\_boost+FN\_boost))

print("Specificity:", TN\_boost/(TN\_boost+FP\_boost))

print("Fallout:", FP\_boost/(FP\_boost + TN\_boost))

print("PPV:", TP\_boost/(TP\_boost + FP\_boost))

print("Accuracy:", (TP\_boost + TN\_boost)/(TP\_boost +TN\_boost + FP\_boost + FN\_boost))

y\_pred\_boost\_test = boosting.predict(X\_test)

boost\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_boost\_test)

print("Misclassification Error Rate", boost\_mer\_test)

print("Testing Confusion Matrix \n", metrics.confusion\_matrix(y\_test, y\_pred\_boost\_test))

boost\_mer\_test = 1 - metrics.accuracy\_score(y\_test,y\_pred\_boost\_test)

conf\_\_boost1 = pd.DataFrame(metrics.confusion\_matrix(y\_test, y\_pred\_boost\_test))

TP\_boost1 = conf\_\_boost1.iloc[0,0]

FP\_boost1 = conf\_\_boost1.iloc[1,0]

FN\_boost1 = conf\_\_boost1.iloc[0,1]

TN\_boost1 = conf\_\_boost1.iloc[1,1]

print("Misclassification Error Rate", boost\_mer\_test)

print("Recall:", TP\_boost1/(TP\_boost1+FN\_boost1))

print("Specificity:", TN\_boost1/(TN\_boost1+FP\_boost1))

print("Fallout:", FP\_boost1/(FP\_boost1 + TN\_boost1))

print("PPV:", TP\_boost1/(TP\_boost1 + FP\_boost1))

print("Accuracy:", (TP\_boost1 + TN\_boost1)/(TP\_boost1 +TN\_boost1 + FP\_boost1 + FN\_boost1))

Results:

Misclassification Error Rate 0.07511737089201875

Training Confusion Matrix

[[151 8]

[ 24 243]]

Recall: 0.949685534591195

Specificity: 0.9101123595505618

Fallout: 0.0898876404494382

PPV: 0.8628571428571429

Accuracy: 0.9248826291079812

Misclassification Error Rate 0.11888111888111885

Testing Confusion Matrix

[[47 6]

[11 79]]

Misclassification Error Rate 0.11888111888111885

Recall: 0.8867924528301887

Specificity: 0.8777777777777778

Fallout: 0.12222222222222222

PPV: 0.8103448275862069

Accuracy: 0.8811188811188811

* 1. .

Using the feature\_importances attribute I was able to produce the variable importance on this model. See code below:

Importance\_boosting = pd.DataFrame({'Importance': boosting.feature\_importances\_\*100})

Importance\_boosting.sort\_values(by = 'Importance',

axis = 0,

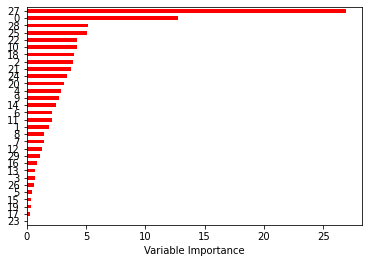
ascending = True). plot(kind = 'barh',

color = 'r', )

plt.xlabel('Variable Importance')

plt.gca().legend\_ = None

Results:



1. Use SVM with linear kernel to construct your classifier.
   1. NA
   2. .

Using GridSearchCV and SVC commands I was able to test the parameter C for the linear kernel. See code below:

c\_range = np.arange(1,100,1)

tuned\_parameters = [{'C': c\_range}]

clf = GridSearchCV(SVC(kernel='linear'), tuned\_parameters, cv=10, scoring='accuracy')

clf.fit(X\_train, y\_train)

print(clf.best\_params\_)

Results:

{'C': 1}

* 1. .
  2. .
  3. .

Using these parameters I was able to fit the prediction model based off of the training set data. Once this was fit properly I was able to make predictions on both the training and testing sets. Once these predictions were made I was able to produce the confusion matrix and corresponding metrics as well as the misclassification error rate. See code below:

svm = SVC(C=1, kernel='linear', probability=True)

svm.fit(X\_train, y\_train)

y\_pred\_linear\_train= svm.predict(X\_train)

#print(classification\_report(y\_train, y\_pred\_linear\_train, digits=3))

print(confusion\_matrix(y\_train, clf.best\_estimator\_.predict(X\_train)))

conf\_l = pd.DataFrame(confusion\_matrix(y\_train, clf.best\_estimator\_.predict(X\_train)))

TPl = conf\_l.iloc[0,0]

FPl = conf\_l.iloc[1,0]

FNl = conf\_l.iloc[0,1]

TNl = conf\_l.iloc[1,1]

print("Recall:", TPl/(TPl+FNl))

print("Specificity:", TNl/(TNl+FPl))

print("Fallout:", FPl/(FPl + TNl))

print("PPV:", TPl/(TPl + FPl))

print("Accuracy:", (TPl + TNl)/(TPl +TNl + FPl + FNl))

svm\_lr\_mer\_train = 1 - ((TPl + TNl)/(TPl +TNl + FPl + FNl))

print("Mislcassification Rate:", svm\_lr\_mer\_train)

y\_pred\_linear\_test = svm.predict(X\_test)

#print(classification\_report(y\_test, y\_pred\_linear\_test, digits=3))

print(confusion\_matrix(y\_test, clf.best\_estimator\_.predict(X\_test)))

conf\_l1 = pd.DataFrame(confusion\_matrix(y\_test, clf.best\_estimator\_.predict(X\_test)))

TPl1 = conf\_l1.iloc[0,0]

FPl1 = conf\_l1.iloc[1,0]

FNl1 = conf\_l1.iloc[0,1]

TNl1 = conf\_l1.iloc[1,1]

print("Recall:", TPl1/(TPl1+FNl1))

print("Specificity:", TNl1/(TNl1+FPl1))

print("Fallout:", FPl1/(FPl1 + TNl1))

print("PPV:", TPl1/(TPl1 + FPl1))

print("Accuracy:", (TPl1 + TNl1)/(TPl1 +TNl1 + FPl1 + FNl1))

svm\_lr\_mer\_test = 1 - ((TPl1 + TNl1)/(TPl1 +TNl1 + FPl1 + FNl1))

print("Mislcassification Rate:", svm\_lr\_mer\_test)

Results:

[[155 4]

[ 2 265]]

Recall: 0.9748427672955975

Specificity: 0.9925093632958801

Fallout: 0.00749063670411985

PPV: 0.9872611464968153

Accuracy: 0.9859154929577465

Mislcassification Rate: 0.014084507042253502

[[51 2]

[ 2 88]]

Recall: 0.9622641509433962

Specificity: 0.9777777777777777

Fallout: 0.022222222222222223

PPV: 0.9622641509433962

Accuracy: 0.972027972027972

Mislcassification Rate: 0.027972027972028024

1. Use SVM with polynomial kernel to construct your classifier.
   1. .
   2. .

Using GridSearchCV and SVC(kernel=’poly’) I was able to test the parameters C, gamma, and degree. See code below:

c\_range\_poly = np.arange(1,100,1)

gamma\_range\_poly = np.arange(0.01,0.05,0.01)

degree\_range = np.arange(1,10,1)

tuned\_parameters\_poly = [{'C': c\_range\_poly,

'degree': degree\_range,

'gamma': gamma\_range\_poly}]

clf\_poly = GridSearchCV(SVC(kernel='poly'), tuned\_parameters\_poly, cv=3, scoring='accuracy')

clf\_poly.fit(X\_train, y\_train)

print(clf\_poly.best\_params\_)

Results:

{'C': 3, 'degree': 1, 'gamma': 0.03}

* 1. .
  2. .
  3. .

Using these parameters I was able to fit the prediction model based off of the training set data. Once this was fit properly I was able to make predictions on both the training and testing sets. Once these predictions were made I was able to produce the confusion matrix and corresponding metrics as well as the misclassification error rate. See code below:

svm\_poly = SVC(C=3, kernel='poly', gamma=0.03, degree = 1, probability=True)

svm\_poly.fit(X\_train, y\_train)

y\_pred\_poly\_train = svm\_poly.predict(X\_train)

#print(classification\_report(y\_train, y\_pred\_poly\_train, digits=3))

print(confusion\_matrix(y\_train, clf\_poly.best\_estimator\_.predict(X\_train)))

conf\_poly = pd.DataFrame(confusion\_matrix(y\_train, clf\_poly.best\_estimator\_.predict(X\_train)))

TP\_poly = conf\_poly.iloc[0,0]

FP\_poly = conf\_poly.iloc[1,0]

FN\_poly = conf\_poly.iloc[0,1]

TN\_poly = conf\_poly.iloc[1,1]

print("Recall:", TP\_poly/(TP\_poly+FN\_poly))

print("Specificity:", TN\_poly/(TN\_poly+FP\_poly))

print("Fallout:", FP\_poly/(FP\_poly + TN\_poly))

print("PPV:", TP\_poly/(TP\_poly + FP\_poly))

print("Accuracy:", (TP\_poly + TN\_poly)/(TP\_poly +TN\_poly + FP\_poly + FN\_poly))

svm\_poly\_mer\_train = 1 - ((TP\_poly + TN\_poly)/(TP\_poly +TN\_poly + FP\_poly + FN\_poly))

print("Mislcassification Rate:", svm\_poly\_mer\_train)

y\_pred\_poly\_test = svm\_poly.predict(X\_test)

#print(classification\_report(y\_test, y\_pred\_poly\_test, digits=3))

print(confusion\_matrix(y\_test, clf\_poly.best\_estimator\_.predict(X\_test)))

conf\_\_poly1 = pd.DataFrame(confusion\_matrix(y\_test, clf\_poly.best\_estimator\_.predict(X\_test)))

TP\_poly1 = conf\_\_poly1.iloc[0,0]

FP\_poly1 = conf\_\_poly1.iloc[1,0]

FN\_poly1 = conf\_\_poly1.iloc[0,1]

TN\_poly1 = conf\_\_poly1.iloc[1,1]

print("Recall:", TP\_poly1/(TP\_poly1+FN\_poly1))

print("Specificity:", TN\_poly1/(TN\_poly1+FP\_poly1))

print("Fallout:", FP\_poly1/(FP\_poly1 + TN\_poly1))

print("PPV:", TP\_poly1/(TP\_poly1 + FP\_poly1))

print("Accuracy:", (TP\_poly1 + TN\_poly1)/(TP\_poly1 +TN\_poly1 + FP\_poly1 + FN\_poly1))

svm\_poly\_mer\_test = 1 - ((TP\_poly1 + TN\_poly1)/(TP\_poly1 +TN\_poly1 + FP\_poly1 + FN\_poly1))

print("Mislcassification Rate:", svm\_poly\_mer\_test)

Results:

[[155 4]

[ 1 266]]

Recall: 0.9748427672955975

Specificity: 0.9962546816479401

Fallout: 0.003745318352059925

PPV: 0.9935897435897436

Accuracy: 0.9882629107981221

Mislcassification Rate: 0.011737089201877882

[[50 3]

[ 2 88]]

Recall: 0.9433962264150944

Specificity: 0.9777777777777777

Fallout: 0.022222222222222223

PPV: 0.9615384615384616

Accuracy: 0.965034965034965

Mislcassification Rate: 0.034965034965035

1. Use SVM with Gaussian (RBF) kernel to construct your classifier.
   1. NA
   2. .

Using GridSearchCV and SVC(kernel=’rbf’) I was able to test the parameters C and gamma. See code below:

c\_range\_rbf = np.arange(1,100,1)

gamma\_range\_rbf = np.arange(0.01,0.05,0.01)

tuned\_parameters\_rbf = [{'C': c\_range\_rbf,

'gamma': gamma\_range\_rbf}]

clf\_rbf = GridSearchCV(SVC(kernel='rbf'), tuned\_parameters\_rbf, cv=3, scoring='accuracy')

clf\_rbf.fit(X\_train, y\_train)

print(clf\_rbf.best\_params\_)

Results:

{'C': 5, 'gamma': 0.01}

* 1. .
  2. .
  3. .

Using these parameters I was able to fit the prediction model based off of the training set data. Once this was fit properly I was able to make predictions on both the training and testing sets. Once these predictions were made I was able to produce the confusion matrix and corresponding metrics as well as the misclassification error rate. See code below:

svm\_rbf = SVC(C=5, kernel='rbf', gamma=0.01, probability = True)

svm\_rbf.fit(X\_train, y\_train)

y\_pred\_rbf\_train = svm\_rbf.predict(X\_train)

#print(classification\_report(y\_train, y\_pred\_rbf\_train, digits=3))

print(confusion\_matrix(y\_train, clf\_rbf.best\_estimator\_.predict(X\_train)))

conf\_rbf = pd.DataFrame(confusion\_matrix(y\_train, clf\_rbf.best\_estimator\_.predict(X\_train)))

TP\_rbf = conf\_rbf.iloc[0,0]

FP\_rbf = conf\_rbf.iloc[1,0]

FN\_rbf = conf\_rbf.iloc[0,1]

TN\_rbf = conf\_rbf.iloc[1,1]

print("Recall:", TP\_rbf/(TP\_rbf+FN\_rbf))

print("Specificity:", TN\_rbf/(TN\_rbf+FP\_rbf))

print("Fallout:", FP\_rbf/(FP\_rbf + TN\_rbf))

print("PPV:", TP\_rbf/(TP\_rbf + FP\_rbf))

print("Accuracy:", (TP\_rbf + TN\_rbf)/(TP\_rbf +TN\_rbf + FP\_rbf + FN\_rbf))

svm\_rbf\_mer\_train = 1 - ((TP\_rbf + TN\_rbf)/(TP\_rbf +TN\_rbf + FP\_rbf + FN\_rbf))

print("Mislcassification Rate:", svm\_rbf\_mer\_train)

y\_pred\_rbf\_test = svm\_rbf.predict(X\_test)

#print(classification\_report(y\_test, y\_pred\_rbf\_test, digits=3))

print(confusion\_matrix(y\_test, clf\_rbf.best\_estimator\_.predict(X\_test)))

conf\_\_rbf1 = pd.DataFrame(confusion\_matrix(y\_test, clf\_rbf.best\_estimator\_.predict(X\_test)))

TP\_rbf1 = conf\_\_rbf1.iloc[0,0]

FP\_rbf1 = conf\_\_rbf1.iloc[1,0]

FN\_rbf1 = conf\_\_rbf1.iloc[0,1]

TN\_rbf1 = conf\_\_rbf1.iloc[1,1]

print("Recall:", TP\_rbf1/(TP\_rbf1+FN\_rbf1))

print("Specificity:", TN\_rbf1/(TN\_rbf1+FP\_rbf1))

print("Fallout:", FP\_rbf1/(FP\_rbf1 + TN\_rbf1))

print("PPV:", TP\_rbf1/(TP\_rbf1 + FP\_rbf1))

print("Accuracy:", (TP\_rbf1 + TN\_rbf1)/(TP\_rbf1 +TN\_rbf1 + FP\_rbf1 + FN\_rbf1))

svm\_rbf\_mer\_test = 1 - ((TP\_rbf1 + TN\_rbf1)/(TP\_rbf1 +TN\_rbf1 + FP\_rbf1 + FN\_rbf1))

print("Mislcassification Rate:", svm\_rbf\_mer\_test)

Results:

[[154 5]

[ 1 266]]

Recall: 0.9685534591194969

Specificity: 0.9962546816479401

Fallout: 0.003745318352059925

PPV: 0.9935483870967742

Accuracy: 0.9859154929577465

Mislcassification Rate: 0.014084507042253502

[[51 2]

[ 2 88]]

Recall: 0.9622641509433962

Specificity: 0.9777777777777777

Fallout: 0.022222222222222223

PPV: 0.9622641509433962

Accuracy: 0.972027972027972

Mislcassification Rate: 0.027972027972028024

1. Compare the different methods by using ROC curves and AUC. Plot the ROC curves on the same graph.

To produce this graph I first calculated the true and false positive rates on each model. Once I obtained this I was able to plot these values. See code below:

#linear kernel

y\_pred\_linear\_prob = svm.predict\_proba(X\_test)

fpr\_lr, tpr\_lr, thresholds = roc\_curve(y\_test, y\_pred\_linear\_prob[: , 1], pos\_label= 1)

print("SVMClassifier (linear kernel): {0}".format(auc(fpr\_lr,tpr\_lr)))

#polynomial kernel

y\_pred\_poly\_prob = svm\_poly.predict\_proba(X\_test)

fpr\_poly, tpr\_poly, thresholds\_poly = roc\_curve(y\_test, y\_pred\_poly\_prob[:,1], pos\_label= 1)

print("SVMClassifier (polynomial kernel): {0}".format(auc(fpr\_poly,tpr\_poly)))

#rbf kernel

y\_pred\_rbf\_prob = svm\_rbf.predict\_proba(X\_test)

fpr\_rbf, tpr\_rbf, thresholds\_rbf = roc\_curve(y\_test, y\_pred\_rbf\_prob[:,1], pos\_label= 1)

print("SVMClassifier (rbf kernel): {0}".format(auc(fpr\_rbf,tpr\_rbf)))

#classification tree

y\_pred\_clf\_prob = linear.predict\_proba(X\_test)

fpr\_clf, tpr\_clf, thresholds\_clf = roc\_curve(y\_test, y\_pred\_clf\_prob[:,1], pos\_label= 1)

print("Classification Tree: {0}".format(auc(fpr\_clf,tpr\_clf)))

#bagging

y\_pred\_bag\_prob = bagging.predict\_proba(X\_test)

fpr\_bag, tpr\_bag, thresholds\_bag = roc\_curve(y\_test, y\_pred\_bag\_prob[:,1], pos\_label= 1)

print("Bagging: {0}".format(auc(fpr\_bag,tpr\_bag)))

#random forest

y\_pred\_rf\_prob = rforest.predict\_proba(X\_test)

fpr\_rf, tpr\_rf, thresholds\_rf = roc\_curve(y\_test, y\_pred\_rf\_prob[:,1], pos\_label= 1)

print("Random Forest: {0}".format(auc(fpr\_rf,tpr\_rf)))

#boosting

y\_pred\_boost\_prob = boosting.predict\_proba(X\_test)

fpr\_boost, tpr\_boost, thresholds\_boost = roc\_curve(y\_test, y\_pred\_boost\_prob[:,1], pos\_label= 1)

print("Random Forest: {0}".format(auc(fpr\_boost,tpr\_boost)))

# Plot ROC curve now

fig = plt.figure(figsize=(12,10))

ax = fig.add\_subplot(111)

# Connect diagonals

ax.plot([0, 1], [0, 1], ls="--")

# Labels etc

ax.set\_xlabel('False Positive Rate')

ax.set\_ylabel('True Positive Rate')

ax.set\_title('ROC curve')

# Set graph limits

ax.set\_xlim([0.0, 1.0])

ax.set\_ylim([0.0, 1.0])

# Plot each graph now

#ax.plot([fpr\_lr, fpr\_poly, fpr\_rbf],[tpr\_lr, tpr\_poly, tpr\_rbf], label = ["lr","poly","rbf"])

#ax.plot(fpr\_lr, tpr\_lr, 'linear', fpr\_poly, tpr\_poly, 'poly', fpr\_rbf, tpr\_rbf, 'rbf')

ax.plot(fpr\_lr,tpr\_lr,label = 'linear')

ax.plot(fpr\_poly,tpr\_poly,label='poly')

ax.plot(fpr\_rbf,tpr\_rbf,label='rbf')

ax.plot(fpr\_clf,tpr\_clf,label='clf\_tree')

ax.plot(fpr\_bag,tpr\_bag,label='bagging')

ax.plot(fpr\_rf,tpr\_rf,label='random\_forest')

ax.plot(fpr\_boost,tpr\_boost,label='boosting')

ax.plot()

# Set legend and show plot

ax.legend(loc="lower right")

plt.show()

Results:

SVMClassifier (linear kernel): 0.9941299790356394

SVMClassifier (polynomial kernel): 0.9966457023060797

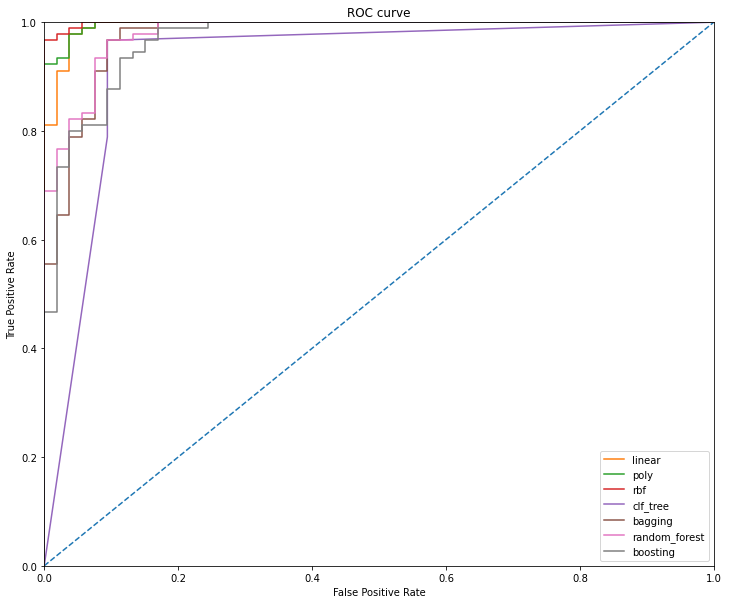
SVMClassifier (rbf kernel): 0.9987421383647799

Classification Tree: 0.9277777777777778

Bagging: 0.9746331236897275

Random Forest: 0.979874213836478

Random Forest: 0.9679245283018869



1. Using the classifiers from previous questions above, report the best classifier for this dataset. The best classifier is the one with the smallest training and test misclassification errors.

First I calculated the average misclassification error rate between the training and testing for each model. Then I was able to see that support vector model using rbf/linear kernels was the best model for this data set. See code below:

#Best Classifier

ct\_mer\_avg = (ct\_mer\_train + ct\_mer\_test)/2

bag\_mer\_avg = (bag\_mer\_train + bag\_mer\_test)/2

rf\_mer\_avg = (rf\_mer\_train + rf\_mer\_test)/2

boost\_mer\_avg = (boost\_mer\_train + boost\_mer\_test)/2

svm\_lr\_mer\_avg = (svm\_lr\_mer\_train + svm\_lr\_mer\_test)/2

svm\_poly\_mer\_avg = (svm\_poly\_mer\_train + svm\_poly\_mer\_test)/2

svm\_rbf\_mer\_avg = (svm\_rbf\_mer\_train + svm\_rbf\_mer\_test)/2

print("Classification Tree MER:", ct\_mer\_avg)

print("Bagging MER", bag\_mer\_avg)

print("Random Forest MER", rf\_mer\_avg)

print("Boosting MER", boost\_mer\_avg)

print("SVM linear MER", svm\_lr\_mer\_avg)

print("SVM poly MER", svm\_poly\_mer\_avg)

print("SVM\_rbf\_MER", svm\_rbf\_mer\_avg)

Results:

Classification Tree MER: 0.11223283758495023

Bagging MER 0.07135001149085657

Random Forest MER 0.06430775796972976

Boosting MER 0.0969992448865688

SVM linear MER 0.021028267507140763

SVM poly MER 0.023351062083456442

SVM\_rbf\_MER 0.021028267507140763