DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Homework Assignment No. 10:

HW No. 10: Nonparametric Classifiers, ROC Curves and AUC

submitted to:

Professor Joseph Picone
ECE 8527: Introduction to Pattern Recognition and Machine Learning
Temple University
College of Engineering
1947 North 12th Street
Philadelphia, Pennsylvania 19122

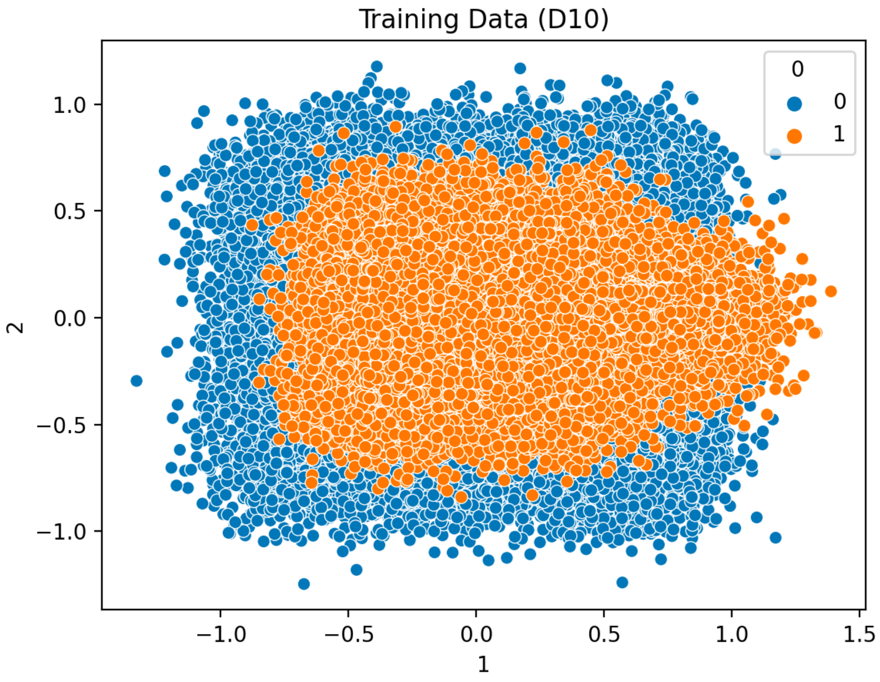
March 30th, 2023

prepared by:

Gavin Koma
Email: gavintkoma@temple.edu

A. TASK 1

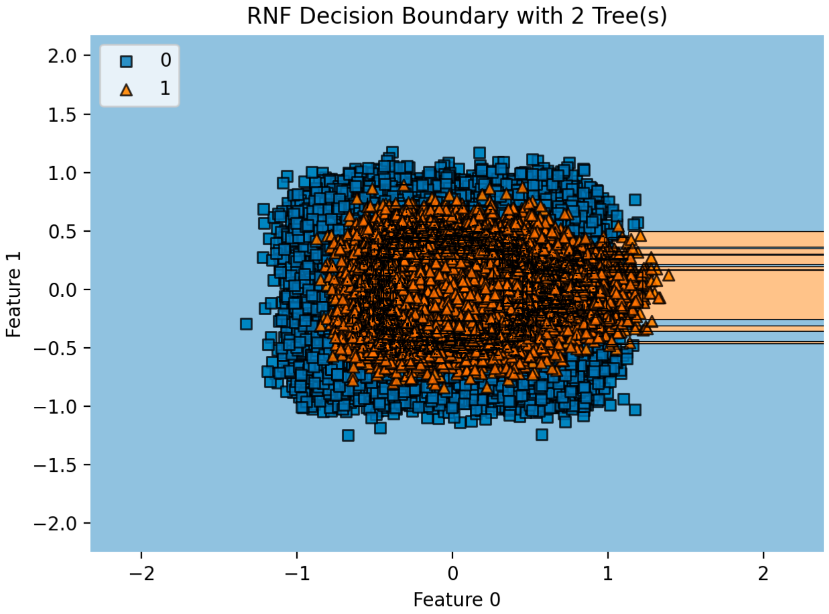
Task 1 of this assignment requires us to use Dataset 10 and implement a Random Forests (RNF) and Support Vector Machines (SVM) using standard Python Packages. We are then to plot the performance on only the eval set as a function of the number of decision tress and as a number of support vectors. To conclude this task, we are to include a table that shows the performance on /train, /dev, and /eval. To begin, this task, I first implement standard Python libraries and will create one graph that plots the original data and another that will plot the RNF decision boundary. These include os (for directory management), pandas and numpy (for basic mathematical computations), and sklearn (for a variety of algorithmic implementations). I will begin by working with the /train data to train my model for the RNF implementation and assess the corresponding ROC curve and their error rates. We obtain the following graph from plotting the training data:

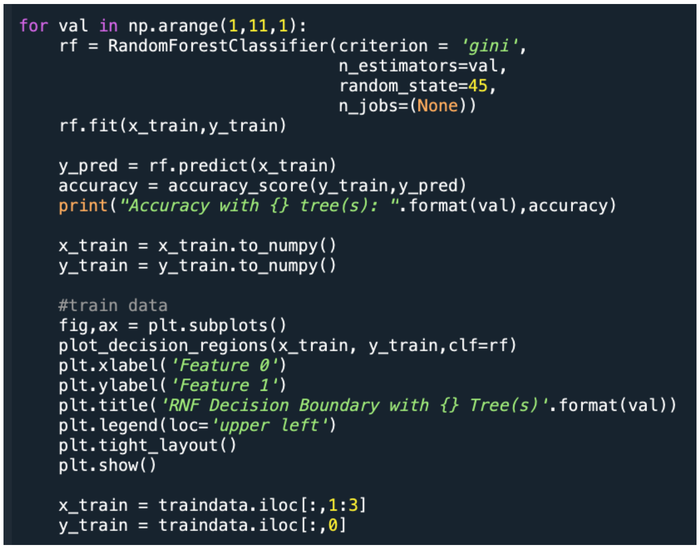


From here, it is essential to determine what hyperparameters to choose for our RNF algorithm. I have chosen to follow the same parameters that are used in imld.py which has 'gini' for criterion, random_state set to 45, and n_jobs having a Boolean value of False. Unfortunately, I am unsure how many decision trees we should include in our algorithm. RNF algorithms are incredibly prone to overfitting and the chances for this increase as we increase the number of decision tress included in our search.

I decided to create a for-loop that runs a RNF algorithm with n number of decision trees where n is a value provided from a list of 10 numbers ranging from 1 to 10. As expected, we are able to get an accuracy of 0.99243 with 10 decision trees. I am inclined to say that although this accuracy is fantastic at sorting through training data, I worry that it will be unable to generalize for our values in the /dev data and /eval data. For this reason, I am going to do a bit of pruning and instead choose to use a random forest with only 2 trees. 2 trees still give a great accuracy of 0.96187. I am further inclined to say that choosing the accuracy score that is less (but still above 90%) will not be detrimental for our purposes and will potentially allow us to get a higher accuracy when using other data values.

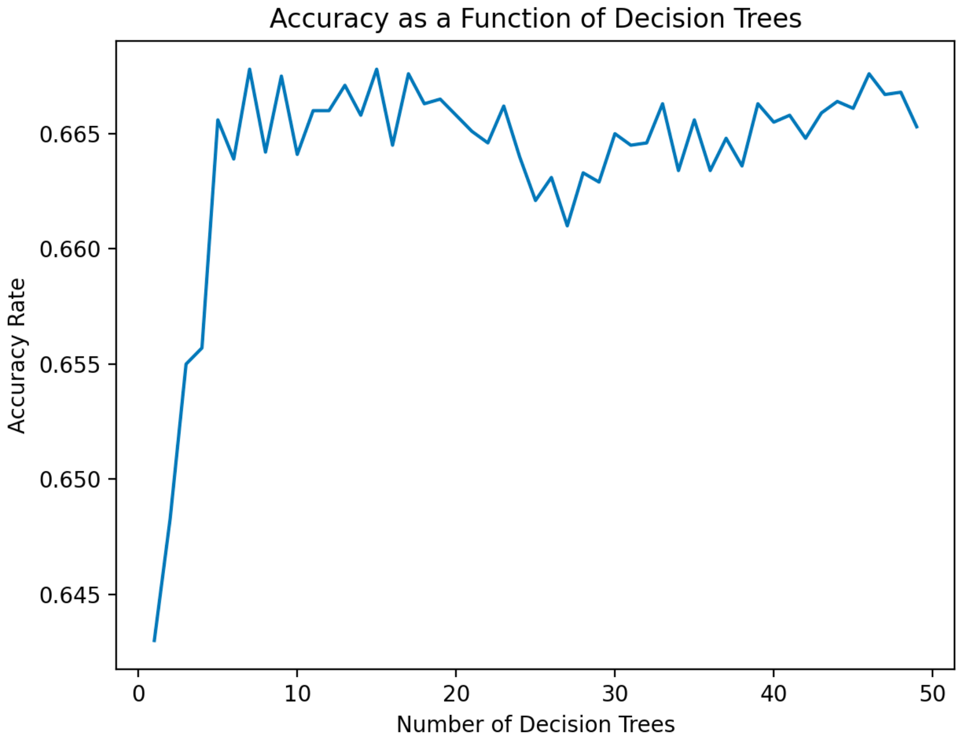
Here is the corresponding code and its output decision boundary:

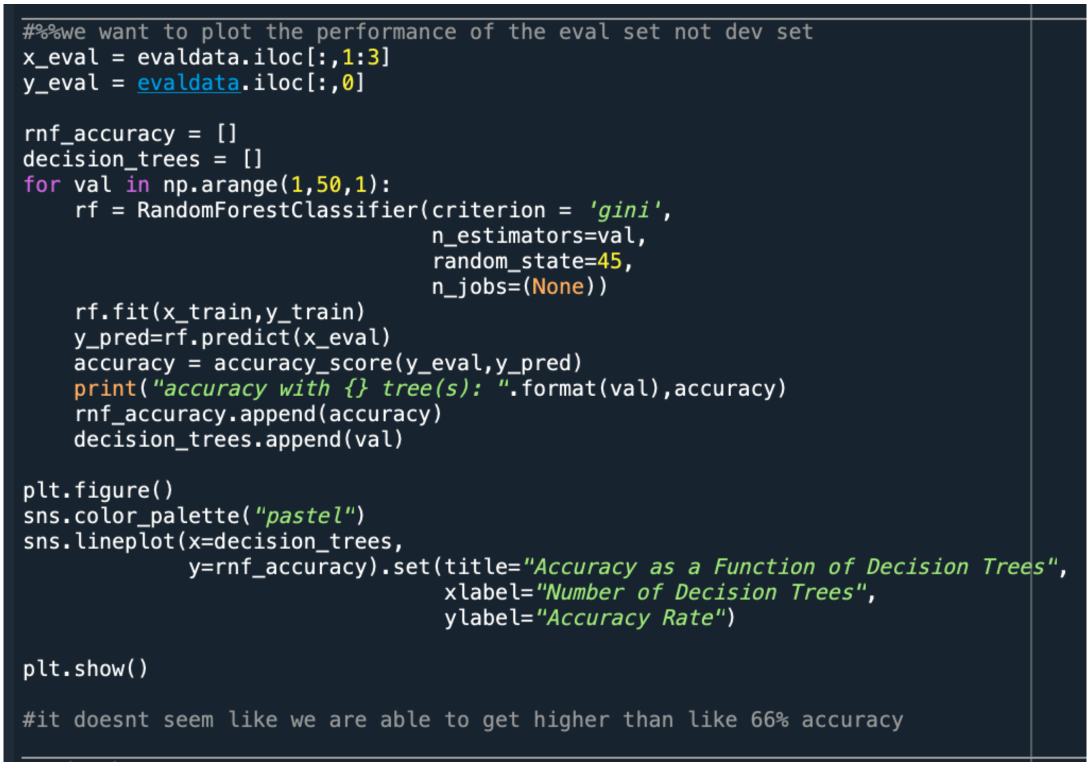




```
for val in np.arange(1,11,1):
    rf = RandomForestClassifier(criterion = 'gini',
                                n_estimators=val,
                                random_state=45,
                                n_jobs=(None))
    rf.fit(x_train,y_train)

    y_pred = rf.predict(x_train)
    accuracy = accuracy_score(y_train,y_pred)
    print("Accuracy with {} tree(s): ".format(val),accuracy)

    x_train = x_train.to_numpy()
    y_train = y_train.to_numpy()

    #train data
    fig,ax = plt.subplots()
    plot_decision_regions(x_train, y_train,clf=rf)
    plt.xlabel('Feature 0')
    plt.ylabel('Feature 1')
    plt.title('RNF Decision Boundary with {} Tree(s)'.format(val))
    plt.legend(loc='upper left')
    plt.tight_layout()
    plt.show()

    x_train = traindata.iloc[:,1:3]
    y_train = traindata.iloc[:,0]
```

After successfully implementing the RNF algorithm, our next goal is to plot performance as a function of the number of decision trees *on the eval set*. To do so, I perform a similar action as above. I call my data, create a for-loop, set my hyperparameters, and append my accuracy values to a list to be plotted after. I am able to obtain the following plot with this corresponding code:





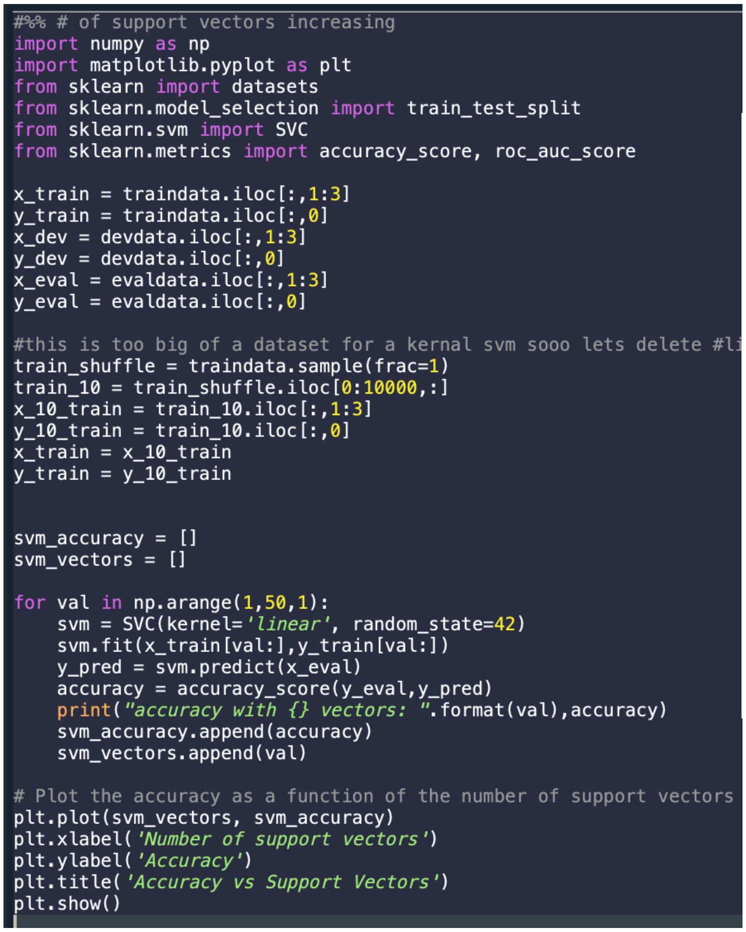
```
#%%we want to plot the performance of the eval set not dev set
x_eval = evaldata.iloc[:,1:3]
y_eval = evaldata.iloc[:,0]

rnf_accuracy = []
decision_trees = []
for val in np.arange(1,50,1):
    rf = RandomForestClassifier(criterion = 'gini',
                                n_estimators=val,
                                random_state=45,
                                n_jobs=(None))
    rf.fit(x_train,y_train)
    y_pred=rf.predict(x_eval)
    accuracy = accuracy_score(y_eval,y_pred)
    print("accuracy with {} tree(s): ".format(val),accuracy)
    rnf_accuracy.append(accuracy)
    decision_trees.append(val)

plt.figure()
sns.color_palette("pastel")
sns.lineplot(x=decision_trees,
             y=rnf_accuracy).set(title="Accuracy as a Function of Decision Trees",
                                 xlabel="Number of Decision Trees",
                                 ylabel="Accuracy Rate")

plt.show()

#it doesnt seem like we are able to get higher than like 66% accuracy
```

Here, we see a peak accuracy of 0.6678 with 7 decision trees included in our RNF algorithm. I understand that normally we are able to obtain an incredibly high accuracy rate with RNF. I am now curious if the lack of data points is resulting in a poor accuracy or if there is notably more overlap than seen in the training data. The same code was run but with /dev data in order to assess the accuracy (0.606 with 7 decision trees).

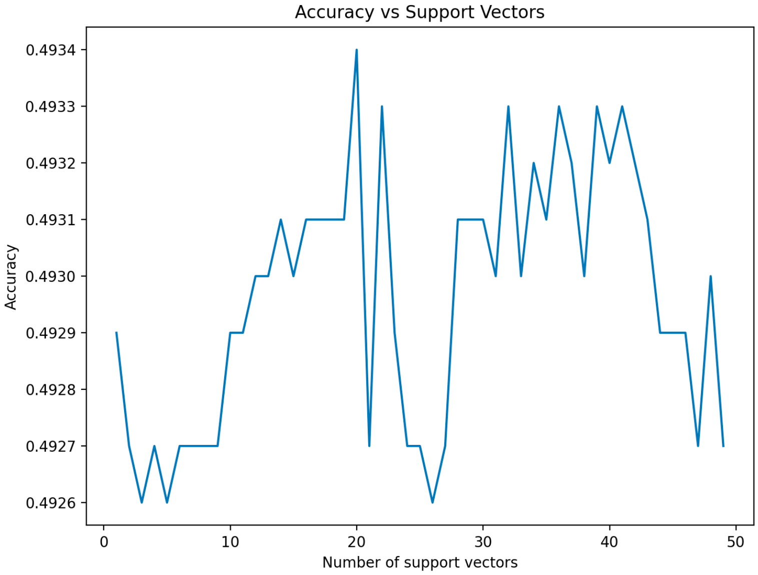
The next step of this Task is to implement a Support Vector Machine of varying types to assess the data. I began this step by just including all of my training data (100,000 datapoints) to build my algorithm. However, I realized that this soon wouldn't be feasible. Everytime I ran the first step of this code, my computer's fan would immediately start blasting and I could hear the screams rising from my computer's CPU. After doing some quick research, I realized how computationally intensive SVM's can be because of their basis around kernel functions.

Most SVM implementations explicitly store training data as an NxN matrix of distances between training points; when the training data is very large, a consequently large amount of RAM will be needed to store these distances in cache and make computations regarding them. Many forums and articles state that a randomization of the data paired with utilizing only a small percentage of training data will often result in similar accuracy values.

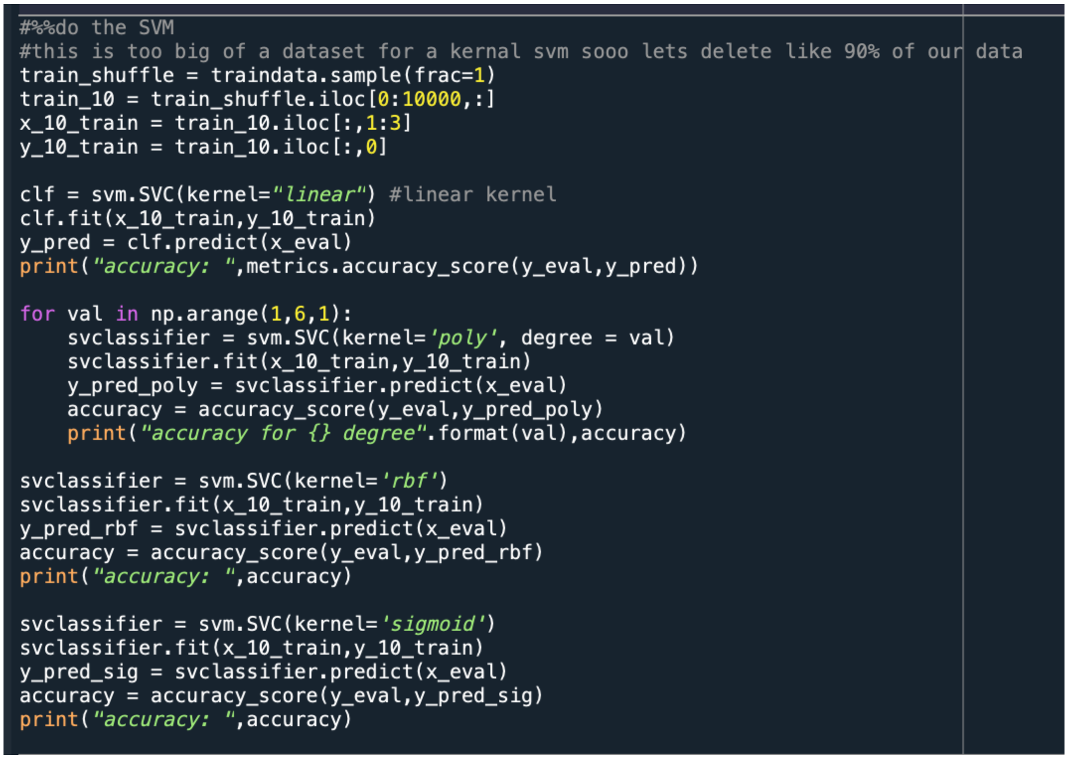
Once I had removed ~90% of the training data post-randomization, I began implementing my SVM on my /eval data. I did not see anything in regard to the number of support vector machines but rather saw a lot of documentation regarding the varying *types* of support vector machines. It wasn't until I deep-dove into a book titled "*Python for Data Analysis*" with a lovely mongoose on the cover that I came across a section regarding varying index. The code that I used to vary the support vectors is as follows and produced the following graph:



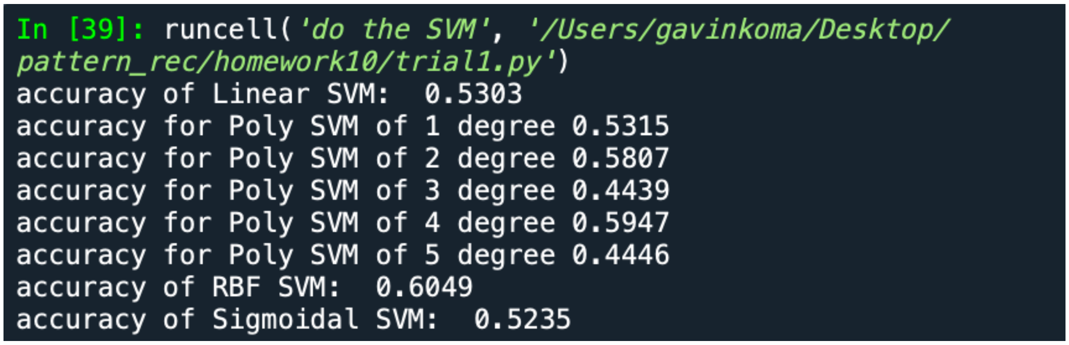
```
#%% # of support vectors increasing
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, roc_auc_score

x_train = traindata.iloc[:,1:3]
y_train = traindata.iloc[:,0]
x_dev = devdata.iloc[:,1:3]
y_dev = devdata.iloc[:,0]
x_eval = evaldata.iloc[:,1:3]
y_eval = evaldata.iloc[:,0]

#this is too big of a dataset for a kernal svm sooo lets delete #li
train_shuffle = traindata.sample(frac=1)
train_10 = train_shuffle.iloc[0:10000,:]
x_10_train = train_10.iloc[:,1:3]
y_10_train = train_10.iloc[:,0]
x_train = x_10_train
y_train = y_10_train


svm_accuracy = []
svm_vectors = []

for val in np.arange(1,50,1):
    svm = SVC(kernel='linear', random_state=42)
    svm.fit(x_train[val:],y_train[val:])
    y_pred = svm.predict(x_eval)
    accuracy = accuracy_score(y_eval,y_pred)
    print("accuracy with {} vectors: ".format(val),accuracy)
    svm_accuracy.append(accuracy)
    svm_vectors.append(val)

# Plot the accuracy as a function of the number of support vectors
plt.plot(svm_vectors, svm_accuracy)
plt.xlabel('Number of support vectors')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Support Vectors')
plt.show()
```



I found that there was minimal effect to the accuracy of my SVM implementation with varying number of support vectors. It seems that varying the number of support vectors tends to have a more notable effect on datasets of high-dimensionality, not a simple binary classification. I did read, however, that varying *types* of support vector machines can have a drastic effect on the accuracy of the model. Therefore, to test them all and determine which most adequately classifies our data, I rotated through Linear SVM, Polynomial SVM, RBF SVM, and a Sigmoidal SVM. The code that I utilized to do so are as follows:

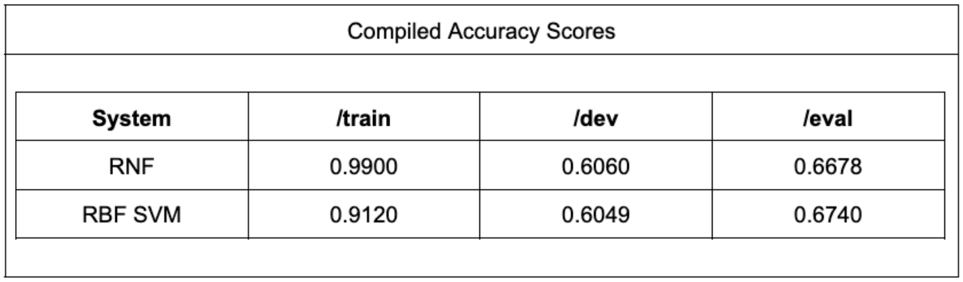


```
#%%do the SVM
#this is too big of a dataset for a kernal svm sooo lets delete like 90% of our data
train_shuffle = traindata.sample(frac=1)
train_10 = train_shuffle.iloc[0:10000,:]
x_10_train = train_10.iloc[:,1:3]
y_10_train = train_10.iloc[:,0]

clf = svm.SVC(kernel="linear") #linear kernel
clf.fit(x_10_train,y_10_train)
y_pred = clf.predict(x_eval)
print("accuracy: ",metrics.accuracy_score(y_eval,y_pred))

for val in np.arange(1,6,1):
    svclassifier = svm.SVC(kernel='poly', degree = val)
    svclassifier.fit(x_10_train,y_10_train)
    y_pred_poly = svclassifier.predict(x_eval)
    accuracy = accuracy_score(y_eval,y_pred_poly)
    print("accuracy for {} degree".format(val),accuracy)

svclassifier = svm.SVC(kernel='rbf')
svclassifier.fit(x_10_train,y_10_train)
y_pred_rbf = svclassifier.predict(x_eval)
accuracy = accuracy_score(y_eval,y_pred_rbf)
print("accuracy: ",accuracy)

svclassifier = svm.SVC(kernel='sigmoid')
svclassifier.fit(x_10_train,y_10_train)
y_pred_sig = svclassifier.predict(x_eval)
accuracy = accuracy_score(y_eval,y_pred_sig)
print("accuracy: ",accuracy)
```



```
In [39]: runcell('do the SVM', '/Users/gavinkoma/Desktop/
pattern_rec/homework10/trial1.py')
accuracy of Linear SVM:  0.5303
accuracy for Poly SVM of 1 degree 0.5315
accuracy for Poly SVM of 2 degree 0.5807
accuracy for Poly SVM of 3 degree 0.4439
accuracy for Poly SVM of 4 degree 0.5947
accuracy for Poly SVM of 5 degree 0.4446
accuracy of RBF SVM:  0.6049
accuracy of Sigmoidal SVM:  0.5235
```

We see that an RBF SVM gives us the highest accuracy of all of the SVMs. An RBF SVM or Radial Basis Function SVM is works by using a radial basis function as the kernel function. It works by finding a hyperplane that best separates the data and computes the similarity between each data point and the center of the hyperplane. The plane of any SVM is always chosen as to maximize the margin between the classes while minimizing the classification error.

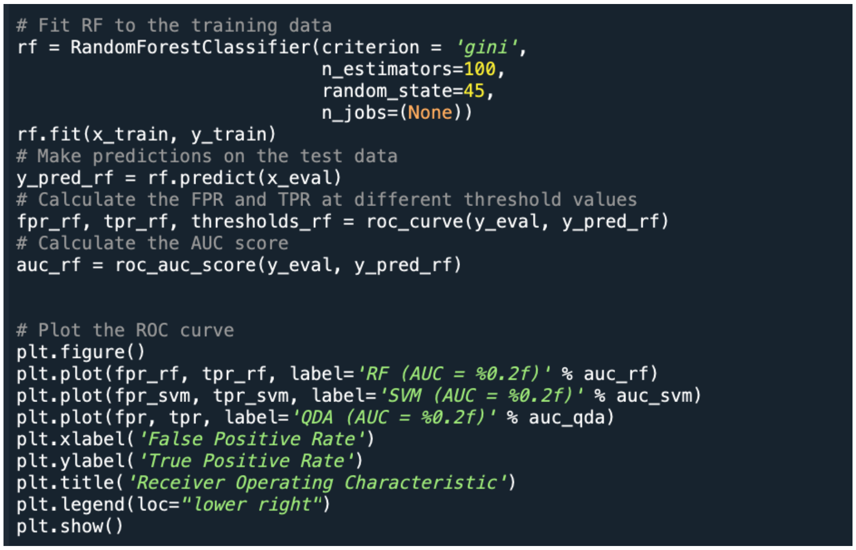


Compiled Accuracy Scores

System	/train	/dev	/eval
RNF	0.9900	0.6060	0.6678
RBF SVM	0.9120	0.6049	0.6740

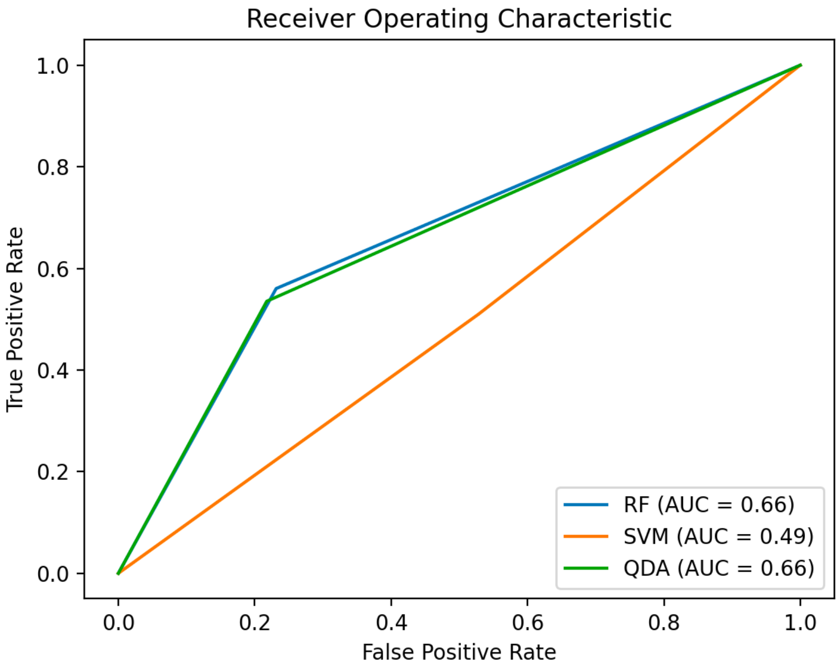
B. TASK 2

For Task 2, our goal is to create and ROC curve for QDA, SVM, and RNF. This portion of the homework was fairly straight forward, and each plot was to be performed on the /eval set. To do so, I first performed a QDA and calculated the fpr, tpr, and threshold values using the roc_curve function from sklearn metrics. I calculated the ROC values in the same way for both SVM and for my RNF classifier. To avoid repetition, I have included the RNF code to calculate the ROC values and my plot function. The only difference between the QDA and SVM are the implementations that were performed. These can be seen, however, in the previous Task where I have already included snippets of code.



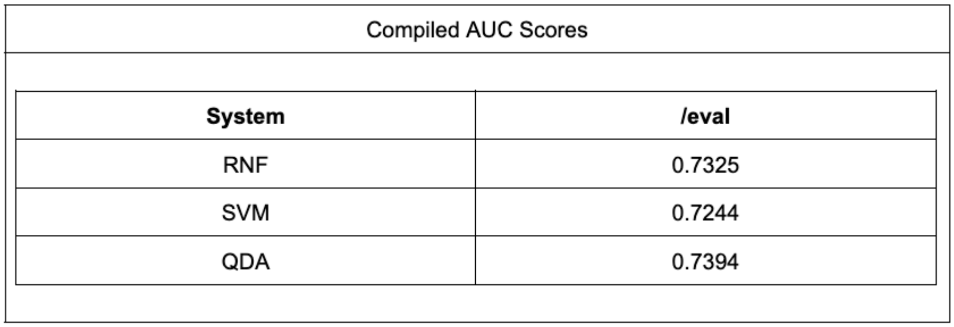
```
# Fit RF to the training data
rf = RandomForestClassifier(criterion = 'gini',
                            n_estimators=100,
                            random_state=45,
                            n_jobs=(None))
rf.fit(x_train, y_train)
# Make predictions on the test data
y_pred_rf = rf.predict(x_eval)
# Calculate the FPR and TPR at different threshold values
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_eval, y_pred_rf)
# Calculate the AUC score
auc_rf = roc_auc_score(y_eval, y_pred_rf)


# Plot the ROC curve
plt.figure()
plt.plot(fpr_rf, tpr_rf, label='RF (AUC = %0.2f)' % auc_rf)
plt.plot(fpr_svm, tpr_svm, label='SVM (AUC = %0.2f)' % auc_svm)
plt.plot(fpr, tpr, label='QDA (AUC = %0.2f)' % auc_qda)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")
plt.show()
```

This code produces the following graph:



The ROC curve is a plot that shows the performance of a binary classifier and is generally used for evaluating the performance of machine learning algorithms. The curve is created by plotting the true positive rate against the false positive rate at varying thresholds. It works to provide a visual representation of the trade-off between the TPR and the FPR. Ideally, our graph would show a curve that is close to the top left corner of the plot which would indicate a high TPR and a low FPR. With this information and knowing that we performed our algorithms on the /eval data, we would likely say that the SVM linear model has a low performance rate while RNF and QDA provide better classification.

To conclude this Task, we are to provide a table that shows the AUC for these three algorithms in /eval.



Compiled AUC Scores

System	/eval
RNF	0.7325
SVM	0.7244
QDA	0.7394

The AOC score represents the area under the Receiver Operating Characteristic (ROC) curve. In order to interpret these scores, we need to have a vague understanding of their meaning at the minimum. A value of ~0.5 shows that the model is not able to distinguish between our binary classes better than random guessing. A value of 1.0 indicates that our model is able to achieve perfect classification performance. Thus, any model should aim to be close to 1.0 or at the minimum, higher than 0.5. All of our scores for each model when classifying the /eval data show some ability to distinguish between our binary classes and indicated a fair performance.

C. CONCLUSION

Overall, this assignment was pretty straightforward. I stumbled a bit when altering the total number of support vectors and it took me a few tries to get it right. I had previously implemented the RNF algorithm for my exam extra credit so it was still fresh in my mind and easy to reintroduce for this homework assignment.