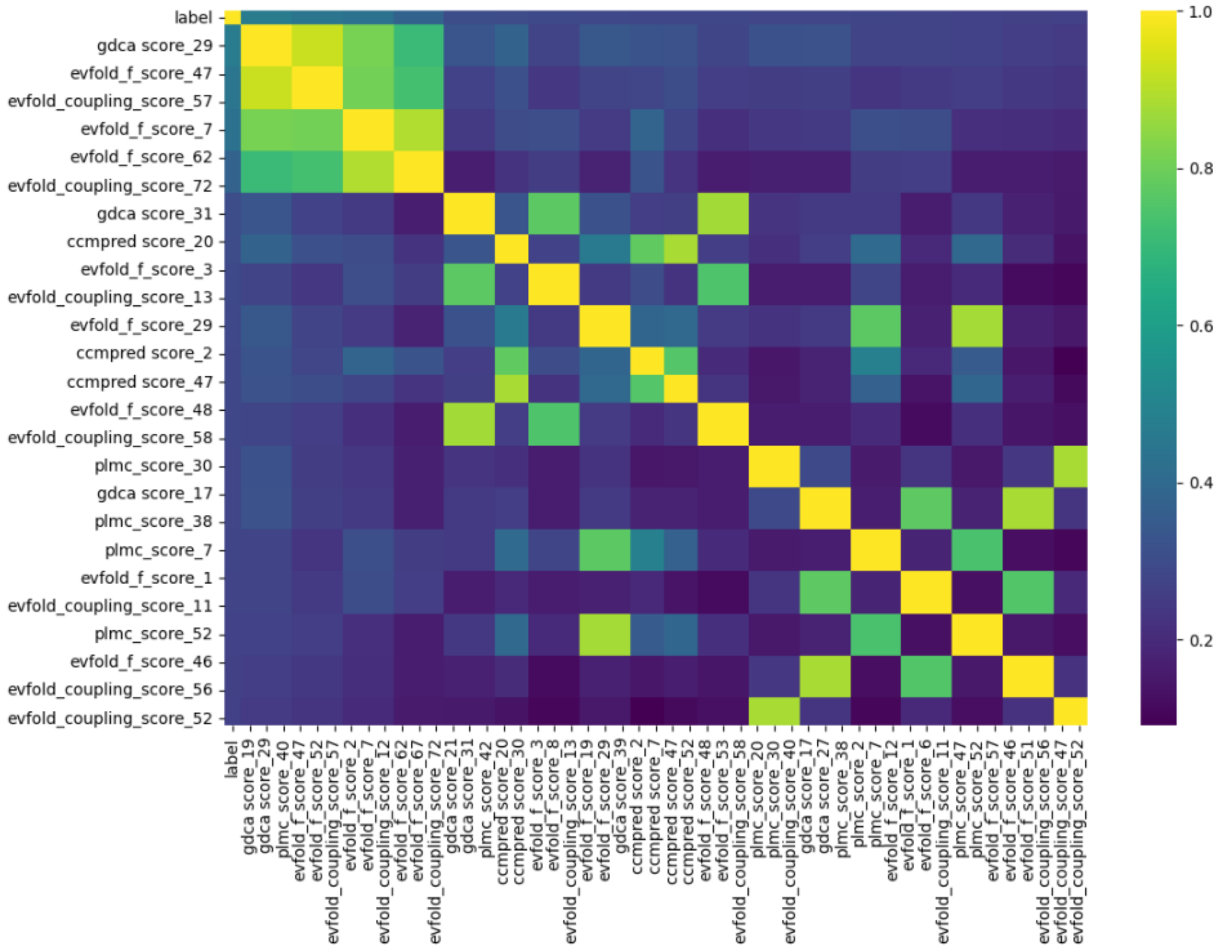
BINF610 – HW2 Report Write-up

For our second homework assignment, we were provided with a dataset consisting of 10,000 examples of residue pairs. Each pair is represented by a 375-dimensional vector of features derived from protein sequence co-evolution, and it is believed that each observation can provide relevant information on whether the pair is in contact or not. To predict the contact status of residue pairs, we were asked to build a Support Vector Machine model with three different kernels: linear, polynomial degree 2, and RBF. Additionally, we were asked to develop a greedy Decision Tree (CART) algorithm and a Random Forest model to classify residue pairs.

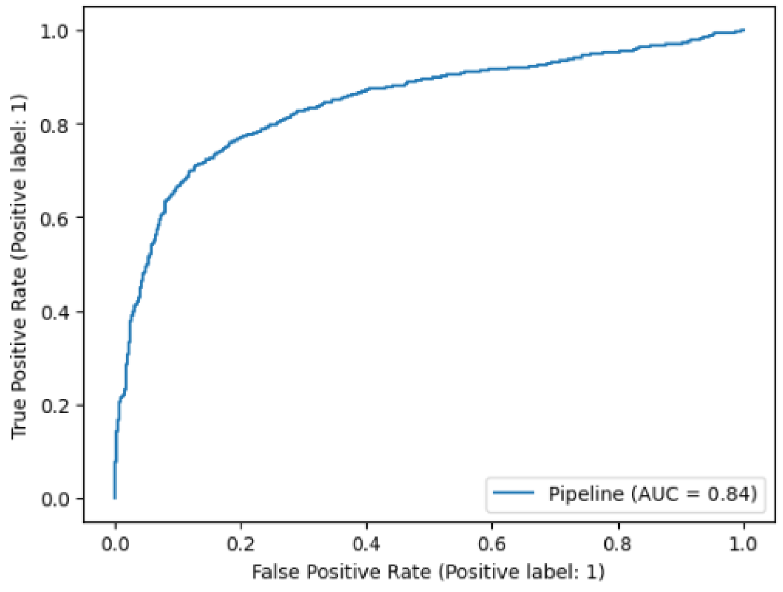
Given the size of the dataset, I was concerned about the computational power required to train and test these models with all 375 features. Therefore, I conducted a correlation analysis to identify the features that were most strongly correlated with the contact status of residue pairs. The screenshot below shows the top 50 features that were found to be highly correlated with the "label" feature, which represents the contact status of the residue pair.



After reducing our original 375 features to just 50, I then created my first SVM model using a Linear Kernel variation. I created three different variations of this model, each with different C values (C=0, 0.01, and 0.1). The C value represents a hyperparameter within the SVM classifier that controls the strength of the regularization, which is inversely proportional to C. This value controls the trade-off between achieving low training error and low testing error. In other words, setting a small or near-zero regularization parameter C value means that the classifier will aim for a larger margin, improving generalization, which helps to prevent overfitting. Increasing the parameter C decreases the model's chances of generalizing well to new, unseen data, as it reduces the model's margins. I was able to see this tradeoff firsthand in the various SVM models that I created. Below are screenshots that illustrate this tradeoff when setting the regularization parameter C to a lower or higher value:

-The screenshots below represent the SVM w/Linear Kernel C=0

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(svm_clf, X_test, y_test);
```



```
y_pred = svm_clf.predict(X_test)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.73      0.89      0.80      1000
           1       0.86      0.68      0.76      1000

    accuracy                           0.78      2000
   macro avg       0.80      0.78      0.78      2000
weighted avg       0.80      0.78      0.78      2000
```

```
scores = cross_val_score(svm_clf, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

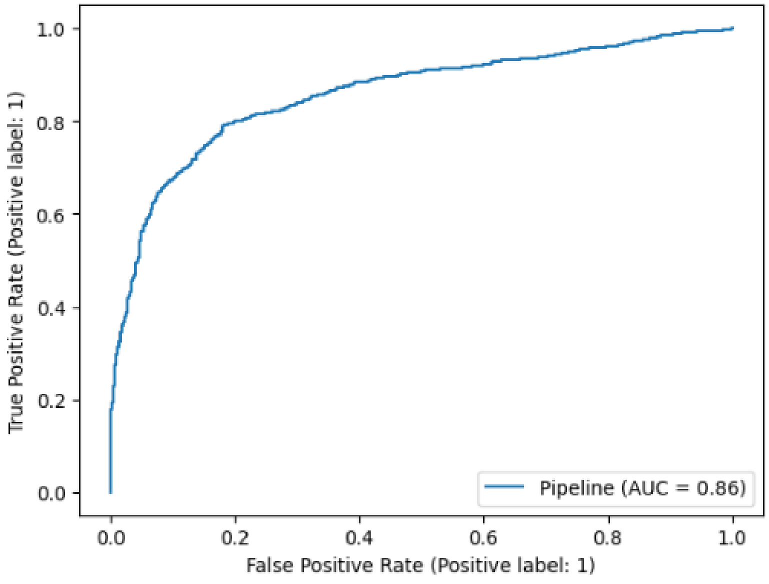
```
Array of actual scores:  [0.79  0.815 0.805 0.765 0.79  0.755 0.755 0.805 0.775 0.785]

 The mean cross validation score is 78.40%
```

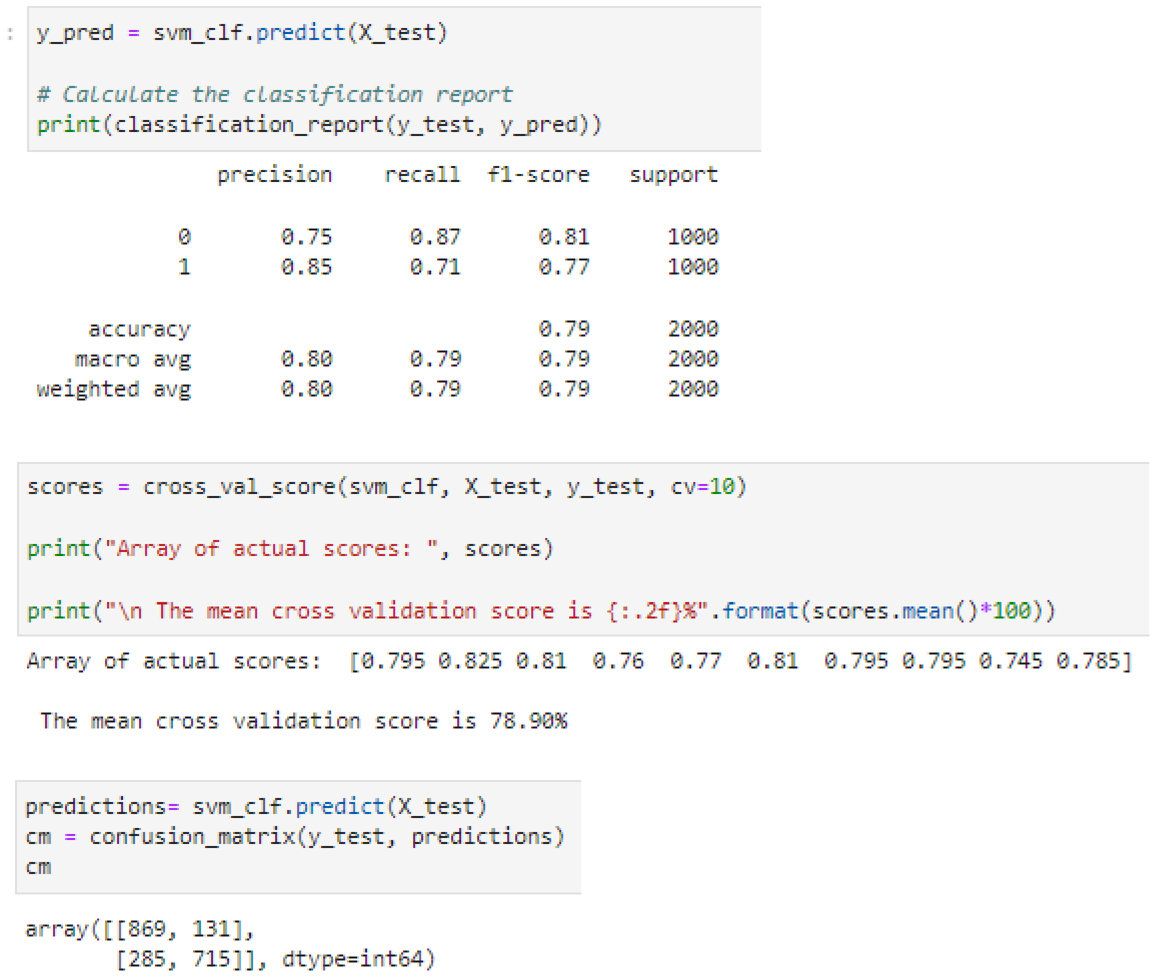
```
predictions= svm_clf.predict(X_test)
cm = confusion_matrix(y_test, predictions)
cm
```

```
array([[890, 110],
       [323, 677]], dtype=int64)
```

-The screenshots below represent the SVM w/Linear Kernel C=.1

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(svm_clf, X_test, y_test);
```





```
y_pred = svm_clf.predict(X_test)

# Calculate the classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.75      0.87      0.81      1000
           1       0.85      0.71      0.77      1000

    accuracy                           0.79      2000
   macro avg       0.80      0.79      0.79      2000
weighted avg       0.80      0.79      0.79      2000
```

```
scores = cross_val_score(svm_clf, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

```
Array of actual scores:  [0.795 0.825 0.81  0.76  0.77  0.81  0.795 0.795 0.745 0.785]

 The mean cross validation score is 78.90%
```

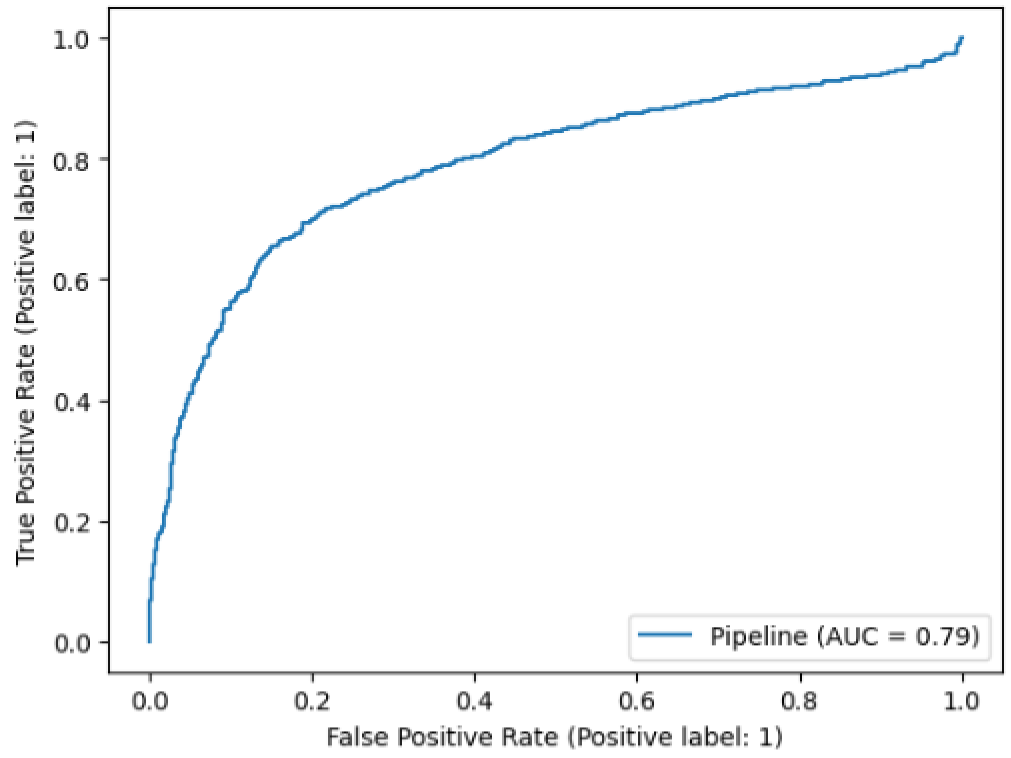
```
predictions= svm_clf.predict(X_test)
cm = confusion_matrix(y_test, predictions)
cm
```

```
array([[869, 131],
       [285, 715]], dtype=int64)
```

Although the differences between the models are subtle, it is apparent that setting a higher C value and increasing the margin to promote more margin violations allow the model to generalize better to the test data. This is particularly evident in the ROC curves and AUC values of the two models. The impact of the C value is even more pronounced in the SVM model with a Polynomial degree of 2. The screenshots below illustrate the effects of the penalty parameter of the error term, C:

- SVM w/Polynomial of Degree 2 Kernel C=.0

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(polynomial_svm_clf, X_test, y_test);
```



```
y_pred = polynomial_svm_clf.predict(X_test)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.63      0.94      0.75      1000
           1       0.88      0.45      0.60      1000

    accuracy                           0.69      2000
   macro avg       0.75      0.70      0.68      2000
weighted avg       0.75      0.69      0.68      2000
```

```
scores = cross_val_score(polynomial_svm_clf, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

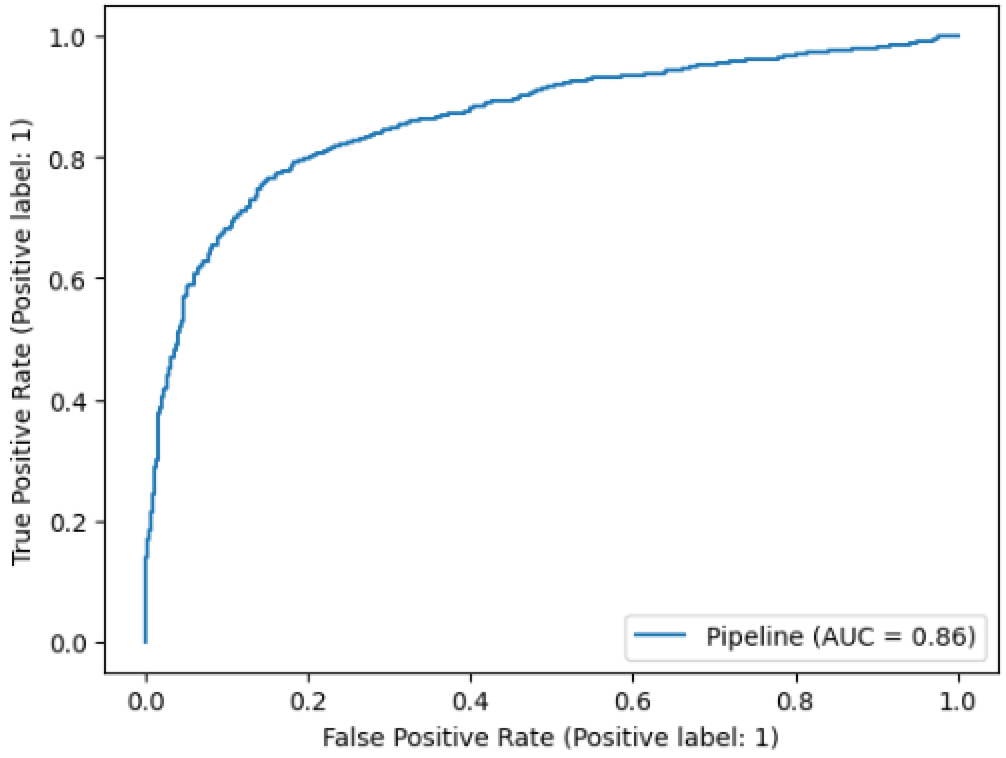
```
Array of actual scores:  [0.645 0.725 0.7   0.68  0.695 0.695 0.715 0.74  0.66  0.675]

 The mean cross validation score is 69.30%
```

```
predictions= polynomial_svm_clf.predict(X_test)
cm = confusion_matrix(y_test, predictions)
cm
```

```
array([[937,  63],
       [547, 453]], dtype=int64)
```

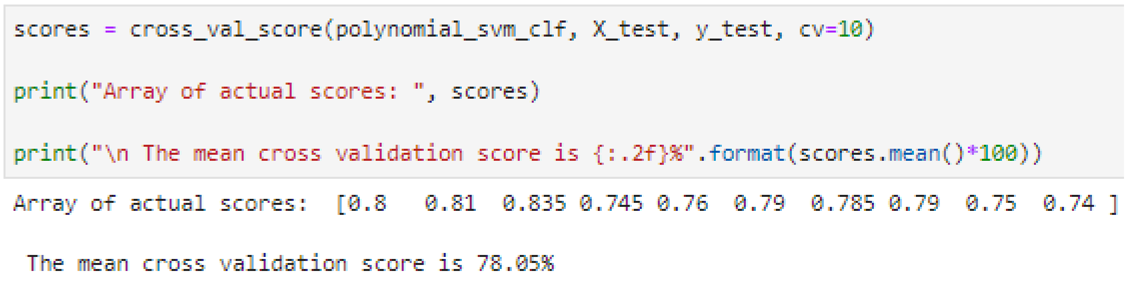
- Here are screenshots from the SVM w/Polynomial of Degree 2 Kernel C=.1

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(polynomial_svm_clf, X_test, y_test);
```



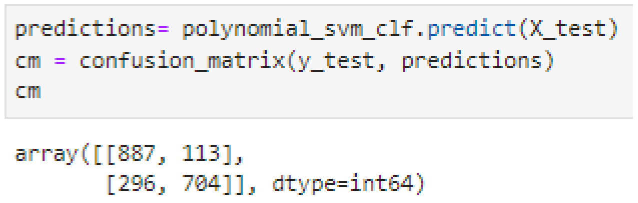
```
y_pred = polynomial_svm_clf.predict(X_test)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.75      0.89      0.81      1000
           1       0.86      0.70      0.77      1000

    accuracy                           0.80      2000
   macro avg       0.81      0.79      0.79      2000
weighted avg       0.81      0.80      0.79      2000
```



```
scores = cross_val_score(polynomial_svm_clf, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

```
Array of actual scores:  [0.8   0.81  0.835 0.745 0.76  0.79  0.785 0.79  0.75  0.74 ]

 The mean cross validation score is 78.05%
```



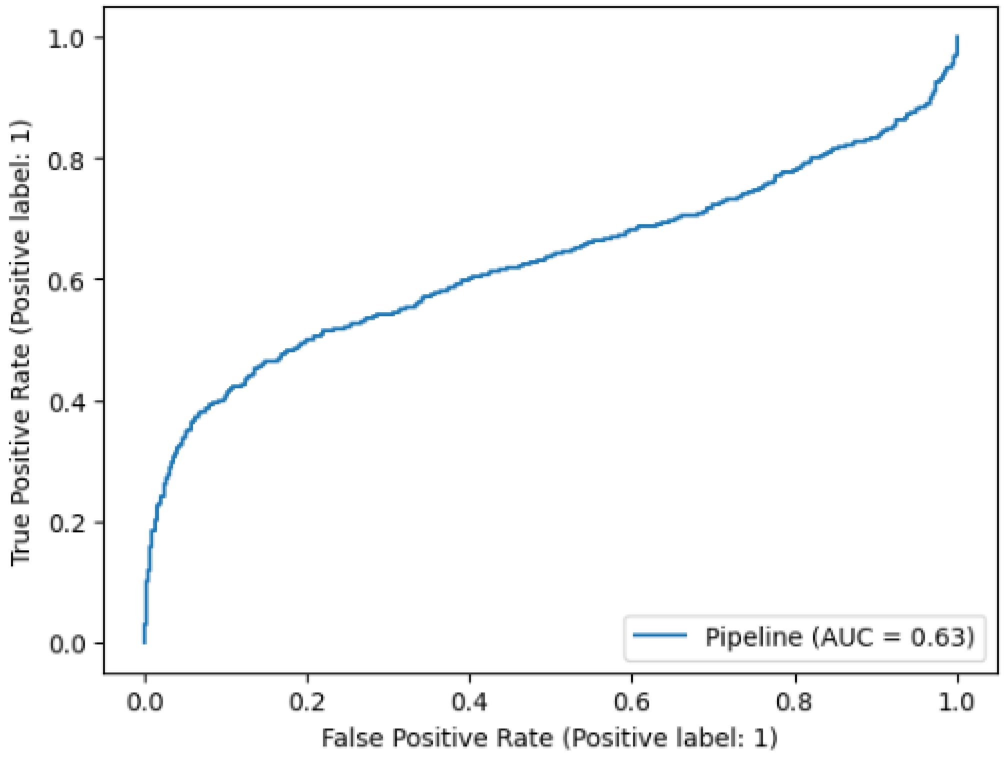
```
predictions= polynomial_svm_clf.predict(X_test)
cm = confusion_matrix(y_test, predictions)
cm
```

```
array([[887, 113],
       [296, 704]], dtype=int64)
```

After comparing the SVM with a Polynomial of Degree 2 Kernel and C values of 0.0 and 0.1, we can observe that the model with a higher C value performs better. Though this may be the case, it is important to note that increasing C too much can eventually lead to a decrease in the model's performance. By increasing C, we reduce the amount of regularization applied to the SVM model, thus making it more complex and fitting the training data much more closely. Subsequently, the model becomes more prone to overfitting. When the model learns to fit the training data too well, it can lead to poor performance on new unseen data. To illustrate this, I have attached some screenshots of a quick test that supports my argument.

- Here are screenshots from the SVM w/Polynomial of Degree 2 Kernel C=100:

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(polynomial_svm_clf, X_test, y_test);
```



```
y_pred = polynomial_svm_clf.predict(X_test)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.49      0.23      0.31      1000
           1       0.50      0.76      0.60      1000

    accuracy                           0.49      2000
   macro avg       0.49      0.49      0.45      2000
weighted avg       0.49      0.49      0.45      2000
```

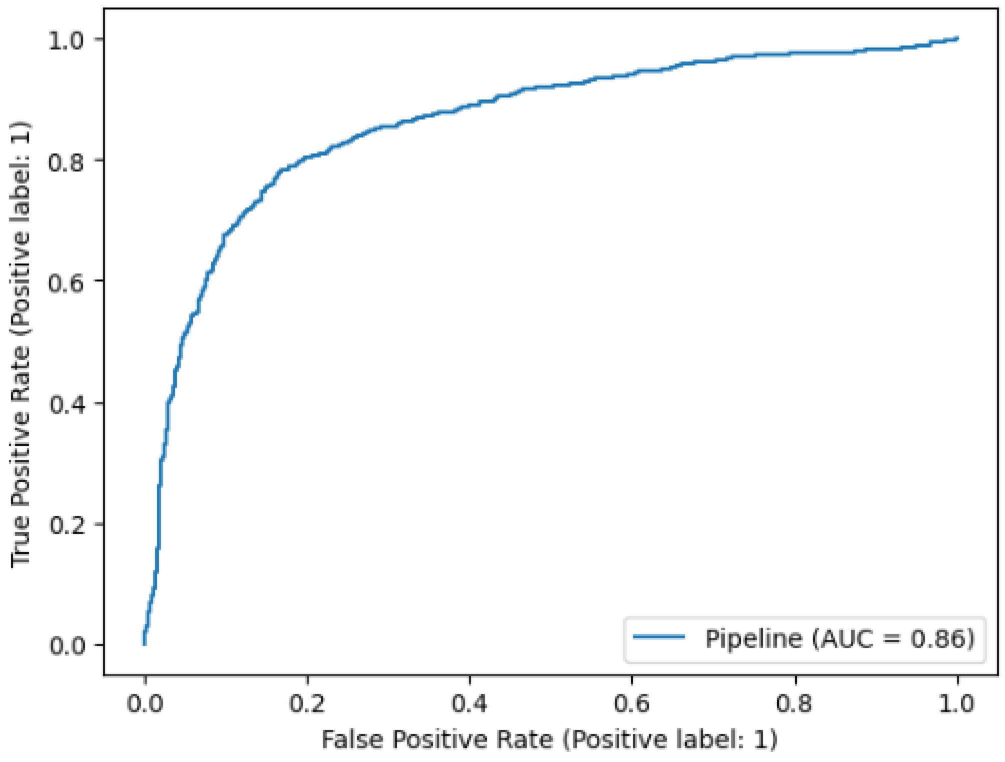
```
scores = cross_val_score(polynomial_svm_clf, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

```
Array of actual scores:  [0.735 0.47  0.71  0.71  0.775 0.765 0.715 0.73  0.495 0.725]

 The mean cross validation score is 68.30%
```

After creating different SVM models with a kernel type of linear and polynomial, I moved onto using RBF. Below are screenshots of the different variations (C = 0, .01, and .1):

- SVM w/RBF Kernel C=0.0

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(rbf_kernel_svm_clf, X_test, y_test);
```



```
y_pred = rbf_kernel_svm_clf.predict(X_test)

# Calculate the classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.74      0.89      0.81      1000
           1       0.86      0.69      0.77      1000

    accuracy                           0.79      2000
   macro avg       0.80      0.79      0.79      2000
weighted avg       0.80      0.79      0.79      2000
```

```
scores = cross_val_score(rbf_kernel_svm_clf, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

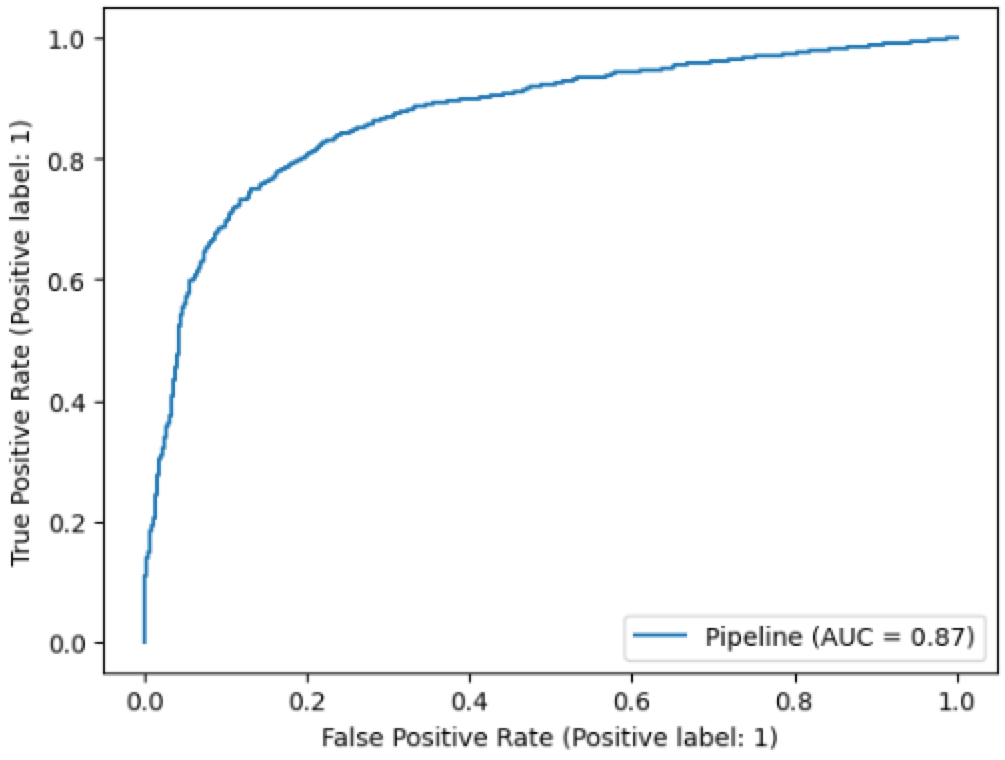
```
Array of actual scores:  [0.775 0.815 0.805 0.765 0.785 0.805 0.785 0.83  0.775 0.78 ]

 The mean cross validation score is 79.20%
```

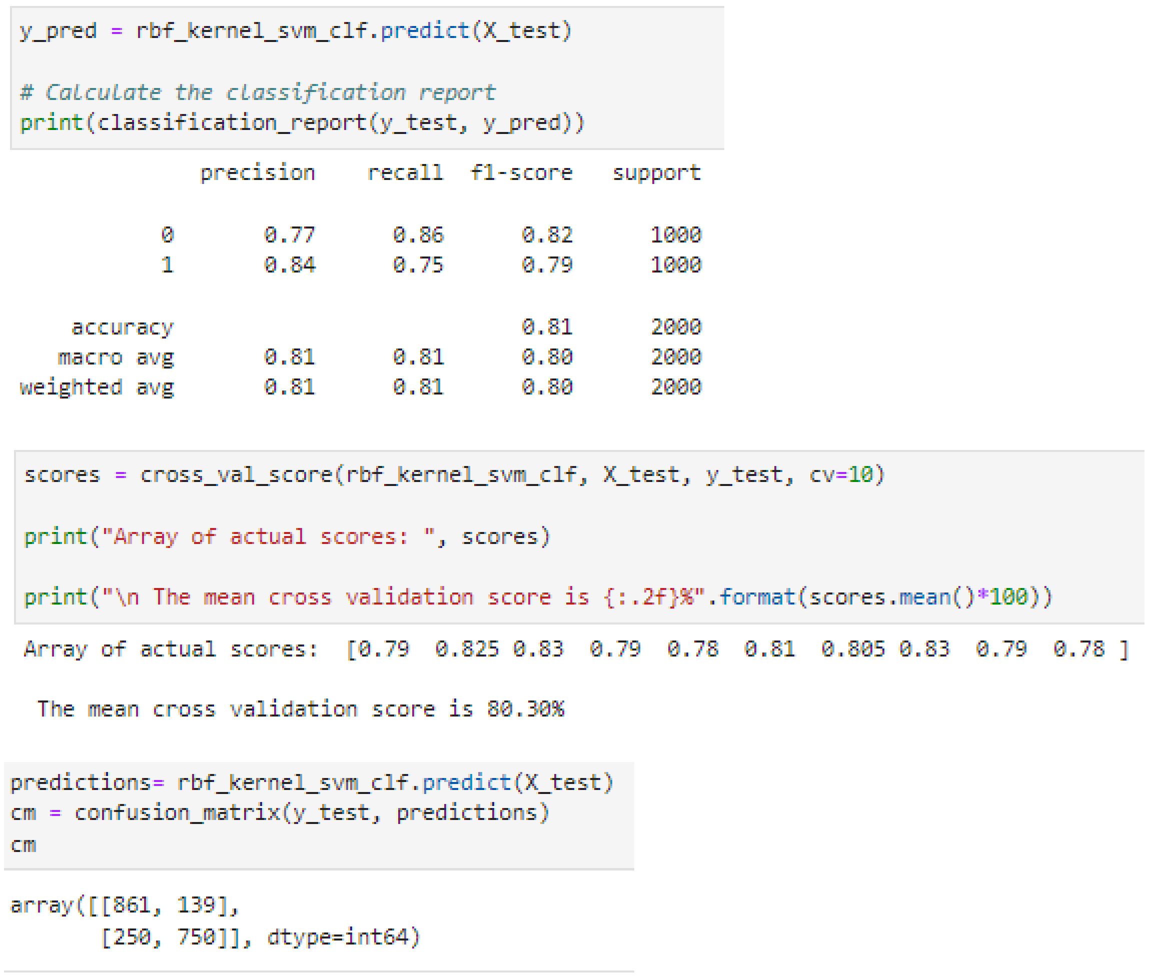
```
predictions= rbf_kernel_svm_clf.predict(X_test)
cm = confusion_matrix(y_test, predictions)
cm
```

```
array([[887, 113],
       [308, 692]], dtype=int64)
```

- SVM w/RBF Kernel C=0.1

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(rbf_kernel_svm_clf, X_test, y_test);
```





```
y_pred = rbf_kernel_svm_clf.predict(X_test)

# Calculate the classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.77      0.86      0.82      1000
           1       0.84      0.75      0.79      1000

    accuracy                           0.81      2000
   macro avg       0.81      0.81      0.80      2000
weighted avg       0.81      0.81      0.80      2000
```

```
scores = cross_val_score(rbf_kernel_svm_clf, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

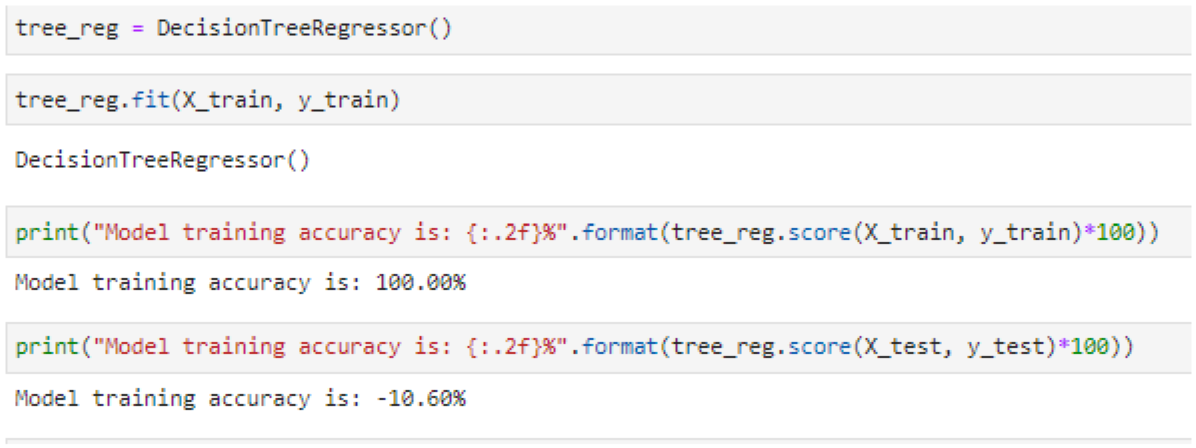
```
Array of actual scores:  [0.79  0.825 0.83  0.79  0.78  0.81  0.805 0.83  0.79  0.78 ]

 The mean cross validation score is 80.30%
```

```
predictions= rbf_kernel_svm_clf.predict(X_test)
cm = confusion_matrix(y_test, predictions)
cm
```

```
array([[861, 139],
       [250, 750]], dtype=int64)
```

After building the SVM models, our next task was to build a Decision Tree for classification and regression (CART) to further our attempt to classify whether a residue pair is in contact or not. Initially, I built a full decision tree without any regularization. The screenshots below represent that model.

- Full Tree without Regularization:



```
tree_reg = DecisionTreeRegressor()
```

```
tree_reg.fit(X_train, y_train)
```

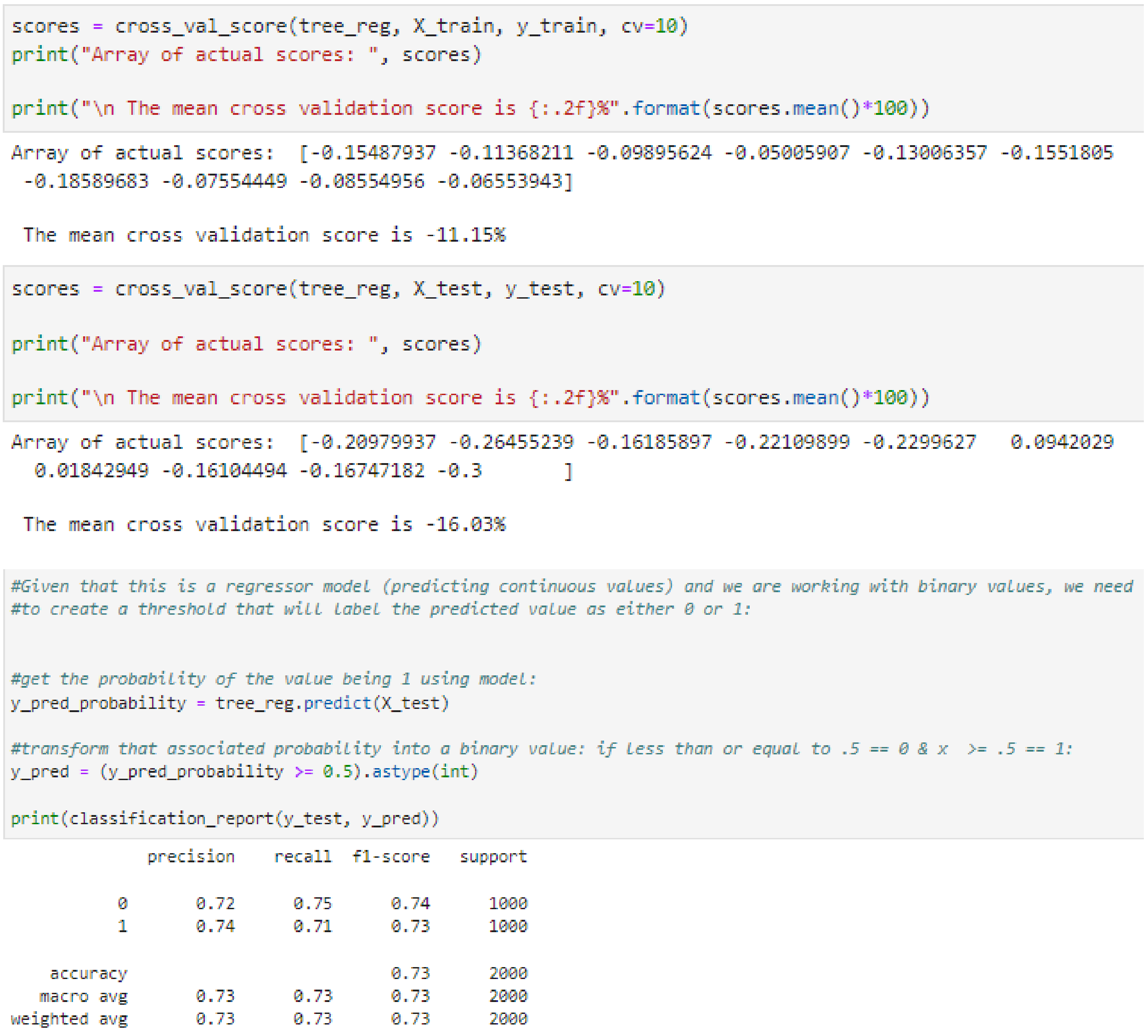
```
DecisionTreeRegressor()
```

```
print("Model training accuracy is: {:.2f}%".format(tree_reg.score(X_train, y_train)*100))
```

```
Model training accuracy is: 100.00%
```

```
print("Model training accuracy is: {:.2f}%".format(tree_reg.score(X_test, y_test)*100))
```

```
Model training accuracy is: -10.60%
```



```
scores = cross_val_score(tree_reg, X_train, y_train, cv=10)
print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

```
Array of actual scores:  [-0.15487937 -0.11368211 -0.09895624 -0.05005907 -0.13006357 -0.1551805
 -0.18589683 -0.07554449 -0.08554956 -0.06553943]

 The mean cross validation score is -11.15%
```

```
scores = cross_val_score(tree_reg, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

```
Array of actual scores:  [-0.20979937 -0.26455239 -0.16185897 -0.22109899 -0.2299627   0.0942029
  0.01842949 -0.16104494 -0.16747182 -0.3       ]

 The mean cross validation score is -16.03%
```

```
#Given that this is a regressor model (predicting continuous values) and we are working with binary values, we need
#to create a threshold that will label the predicted value as either 0 or 1:


#get the probability of the value being 1 using model:
y_pred_probability = tree_reg.predict(X_test)

#transform that associated probability into a binary value: if less than or equal to .5 == 0 & x  >= .5 == 1:
y_pred = (y_pred_probability >= 0.5).astype(int)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.72      0.75      0.74      1000
           1       0.74      0.71      0.73      1000

    accuracy                           0.73      2000
   macro avg       0.73      0.73      0.73      2000
weighted avg       0.73      0.73      0.73      2000
```

We know from the lecture that Decision Tree Regressors are greedy algorithms, meaning that they make locally optimal decisions at each node in the tree but do not consider the global optimal solution. Without any regularization parameters in place, we can see the effects on the model listed above. The model has trained to fit the training data perfectly and as a result has become overfit due to the lack of proper stopping criteria.

In our next variation of a Decision Tree Regressor, I applied a maximum height of 5 as a regularization parameter. With this parameter in place, we can see from the screenshots below that the model has not become as overfit as the previous model.

- Decision Tree Max-height = 5

```
tree_reg = DecisionTreeRegressor(max_depth=5)
```

```
tree_reg.fit(X_train, y_train)
```

```
DecisionTreeRegressor(max_depth=5)
```

```
print("Model training accuracy is: {:.2f}%".format(tree_reg.score(X_train, y_train)*100))
```

```
Model training accuracy is: 42.98%
```

```
print("Model training accuracy is: {:.2f}%".format(tree_reg.score(X_test, y_test)*100))
```

```
Model training accuracy is: 34.96%
```

```
scores = cross_val_score(tree_reg, X_train, y_train, cv=10)
print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

```
Array of actual scores:  [0.35798765 0.39377633 0.3427675  0.39453411 0.3897333  0.37444742
 0.34720657 0.42440016 0.37828335 0.32598184]

 The mean cross validation score is 37.29%
```

```
scores = cross_val_score(tree_reg, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

```
Array of actual scores:  [0.28806498 0.30304946 0.3247694  0.29738309 0.22719189 0.36158261
 0.33906675 0.29856834 0.20715379 0.28819876]

 The mean cross validation score is 29.35%
```

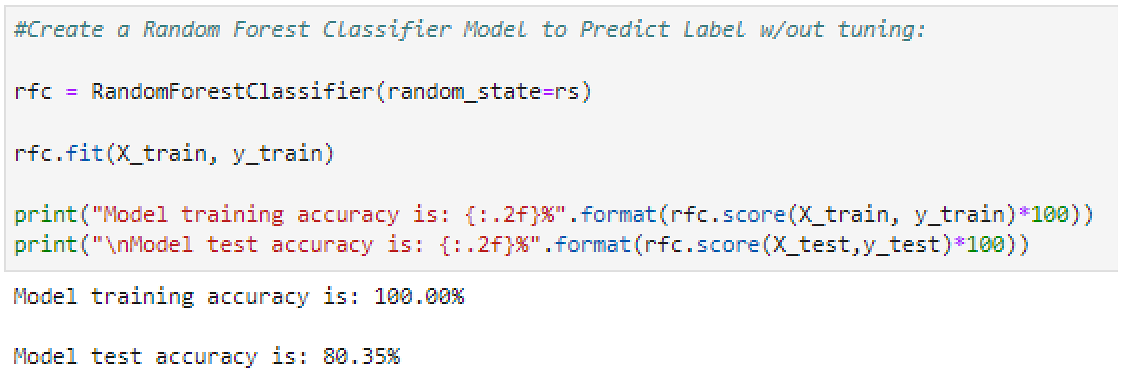
```
#Given that this is a regressor model (predicting continuous values) and we are working with binary values, we need
#to create a threshold that will label the predicted value as either 0 or 1:


#get the probability of the value being 1 using model:
y_pred_probability = tree_reg.predict(X_test)

#transform that associated probability into a binary value: if less than or equal to .5 == 0 & x  >= .5 == 1:
y_pred = (y_pred_probability >= 0.5).astype(int)

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.74      0.84      0.79      1000
           1       0.81      0.71      0.76      1000

    accuracy                           0.77      2000
   macro avg       0.78      0.77      0.77      2000
weighted avg       0.78      0.77      0.77      2000
```

As noted above, with the Decision Tree model having a regularization function in place, with a maximum height of 5, we can see that the model's performance is much better. Not only is the accuracy higher, but the model's precision and recall scores on the test data are also higher. By applying the maximum height parameter during the model's training, we were able to mitigate overfitting and improve the performance of the greedy CART algorithm.

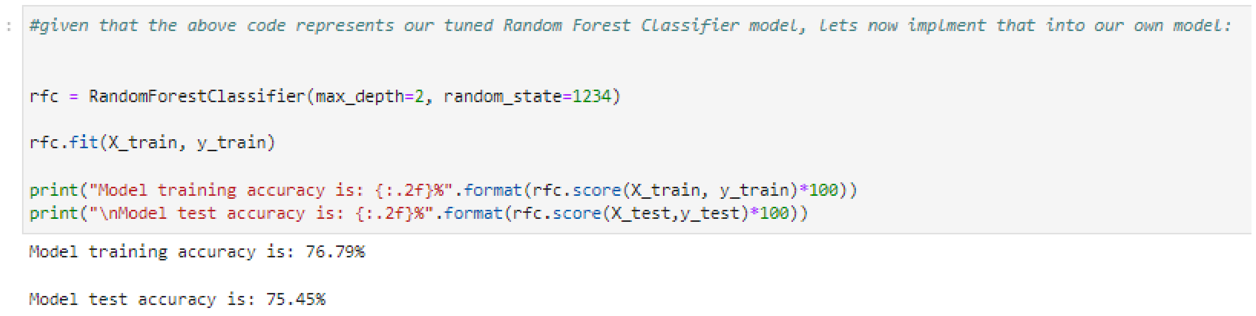
Lastly, for this assignment, I created a Random Forest Classifier model to classify whether a residue pair is in contact or not. Initially, when I created the model and set the Random Forest Classifier parameters to their default settings, the model appeared to be overfit. This was evident from the training accuracy of 100% and a model test accuracy of 80%. After performing a parameter grid search for regularization methods such as maximum depth and number of estimators, I achieved a better-fitting model with a training score of 76% and a test score of 75%. The screenshots below represent this model:

- Random Forest Classifier Model



```
#Create a Random Forest Classifier Model to Predict Label w/out tuning:

rfc = RandomForestClassifier(random_state=rs)

rfc.fit(X_train, y_train)

print("Model training accuracy is: {:.2f}%".format(rfc.score(X_train, y_train)*100))
print("\nModel test accuracy is: {:.2f}%".format(rfc.score(X_test,y_test)*100))
```

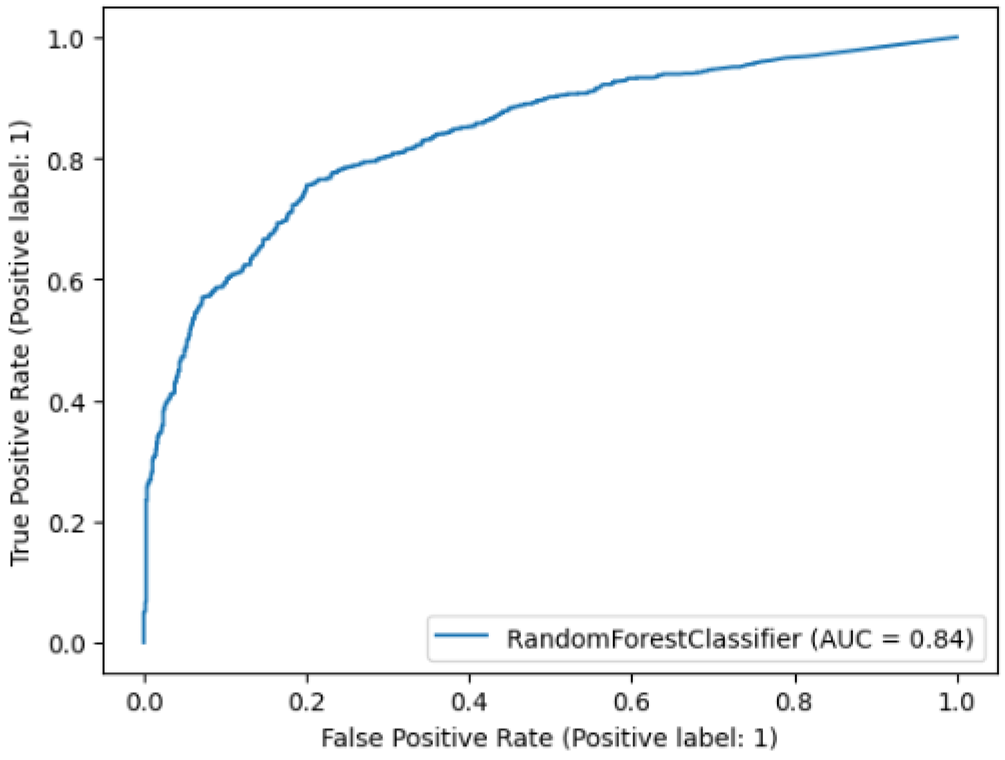
```
Model training accuracy is: 100.00%

Model test accuracy is: 80.35%
```



```
#given that the above code represents our tuned Random Forest Classifier model, lets now implment that into our own model:


rfc = RandomForestClassifier(max_depth=2, random_state=1234)

rfc.fit(X_train, y_train)

print("Model training accuracy is: {:.2f}%".format(rfc.score(X_train, y_train)*100))
print("\nModel test accuracy is: {:.2f}%".format(rfc.score(X_test,y_test)*100))
```

```
Model training accuracy is: 76.79%

Model test accuracy is: 75.45%
```

```
#Display ROC/AUC Curve for TESTING Data:
RocCurveDisplay.from_estimator(rfc, X_test, y_test);
```

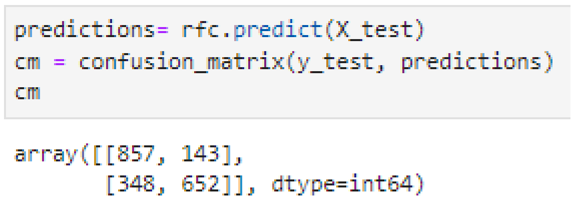


```
y_pred = rfc.predict(X_test)

# Calculate the classification report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.71      0.86      0.78      1000
           1       0.82      0.65      0.73      1000

    accuracy                           0.75      2000
   macro avg       0.77      0.75      0.75      2000
weighted avg       0.77      0.75      0.75      2000
```

```
scores = cross_val_score(rfc, X_test, y_test, cv=10)

print("Array of actual scores: ", scores)

print("\n The mean cross validation score is {:.2f}%".format(scores.mean()*100))
```

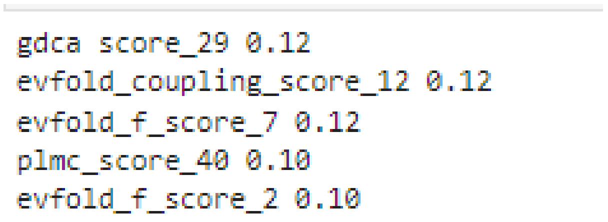
```
Array of actual scores:  [0.755 0.79  0.825 0.73  0.74  0.765 0.755 0.78  0.75  0.775]

 The mean cross validation score is 76.65%
```



```
predictions= rfc.predict(X_test)
cm = confusion_matrix(y_test, predictions)
cm
```

```
array([[857, 143],
       [348, 652]], dtype=int64)
```

As mentioned earlier, we can observe the impact of adding regularization parameters to our models to enhance their performance on unseen data. The section below illustrates the top 5 most significant features of the Random Forest Classifier model:



```
gdca score_29 0.12
evfold_coupling_score_12 0.12
evfold_f_score_7 0.12
plmc_score_40 0.10
evfold_f_score_2 0.10
```

The list above shows that for the Random Forest Classifier, the feature "gdca score_29" had a feature importance of .12, the feature "evfold_coupling_score_12" had a feature importance of .12, and so on. These feature values indicate that these they had some degree of influence on the model's performance, with some varying degree. With this list of values, we can determine which features contributed the most to our model’s output.