CSC2062 AIMA – Assignment 2

Elliot Dickie

40431475

Introduction

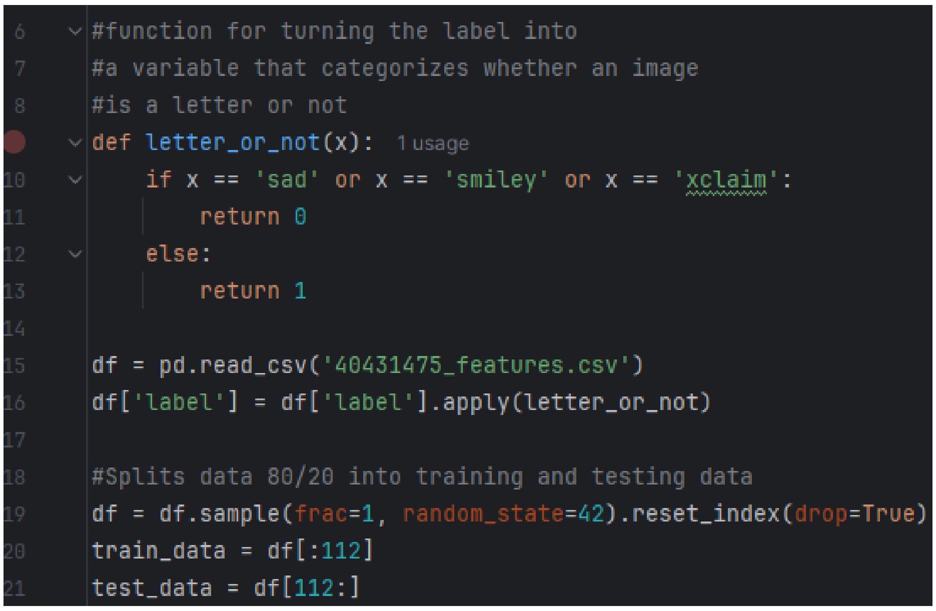
In the following report, I will program, analyze and evaluate logistic regression, k-nearest-neighbour and random forest (decision tree) machine learning algorithms. I will visualize and dissect the performance of these algorithms to extract the best machine learning model. In section 1, I will use a logistic regression model to classify letters and non-letters, calculate a series of key metrics and graph an ROC curve. In section 2, I will use knn techniques and crossvalidation to categorize the data into letters, smiley faces, sad faces, and exclamation marks, before graphing error rates to determine the best value of K. In section 3, I will use random forest to distinguish between categories, and evaluate the best parameters for the random forest algorithm. This will complete my report, and conclude an complete analysis of several different classification algorithms.

Section 1

In this section, I will use the data I created in assignment 1 to create logistic regression models, and use statistics and graphs to analyze and improve them. Logistic regression is a supervised classification algorithm that predicts the probability of an outcome for a binary classification task. It's bounded between 1 and 0 to determine probability.

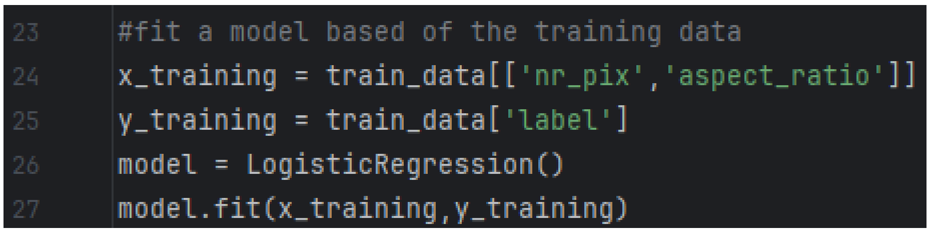
Section 1.1

In order to split the items into a training and testing set (to avoid overfitting), I first imported them into a dataframe, as it allows easier handling of the data. Then, I labelled the data with a 0 or 1 based on whether they were a letter or not (as logisitic regression is a supervised classification method, this is required). Finally, I randomized the order of the data, and split it at the 112th item.



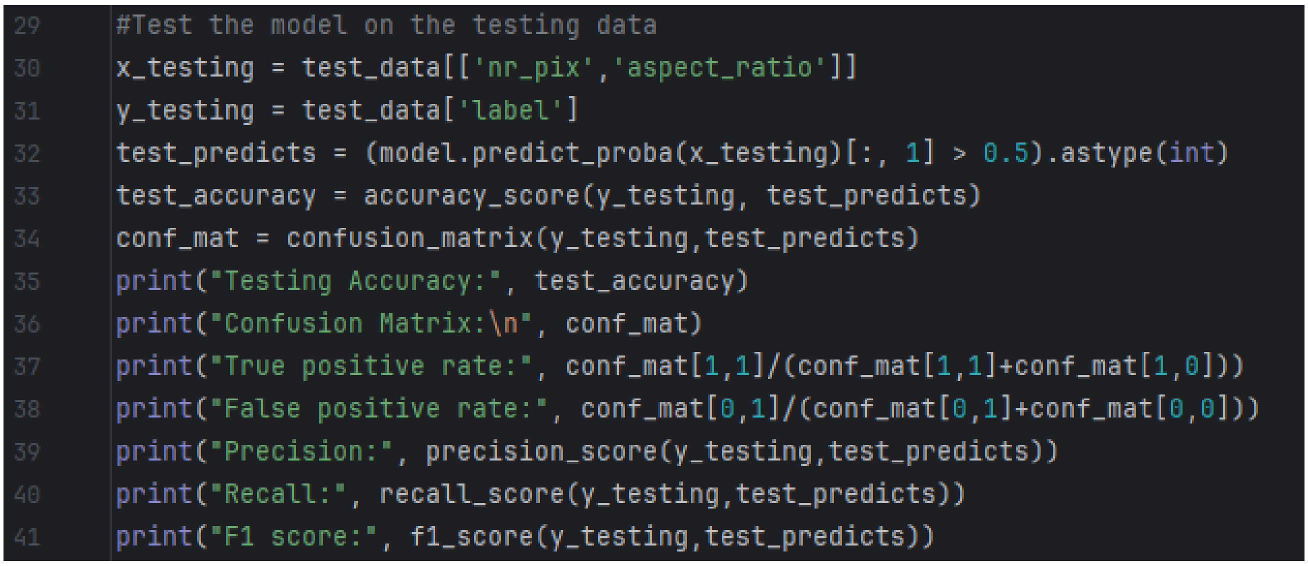
```python
#function for turning the label into
#a variable that categorizes whether an image
#is a letter or not
def letter_or_not(x):  1 usage
    if x == 'sad' or x == 'smiley' or x == 'xclaim':
        return 0
    else:
        return 1

df = pd.read_csv('40431475_features.csv')
df['label'] = df['label'].apply(letter_or_not)

#Splits data 80/20 into training and testing data
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
train_data = df[:112]
test_data = df[112:]
```

Now that my data was appropriately split and labelled, I can proceed with training the logistic regression model, using my partitioned training data.[1]



```
#fit a model based of the training data
x_training = train_data[['nr_pix','aspect_ratio']]
y_training = train_data['label']
model = LogisticRegression()
model.fit(x_training,y_training)
```

I fit the model to number of pixels and aspect ratio using scikit-learn's logistic regression library. Now that my initial model is complete, I can analyze it using key statistics.

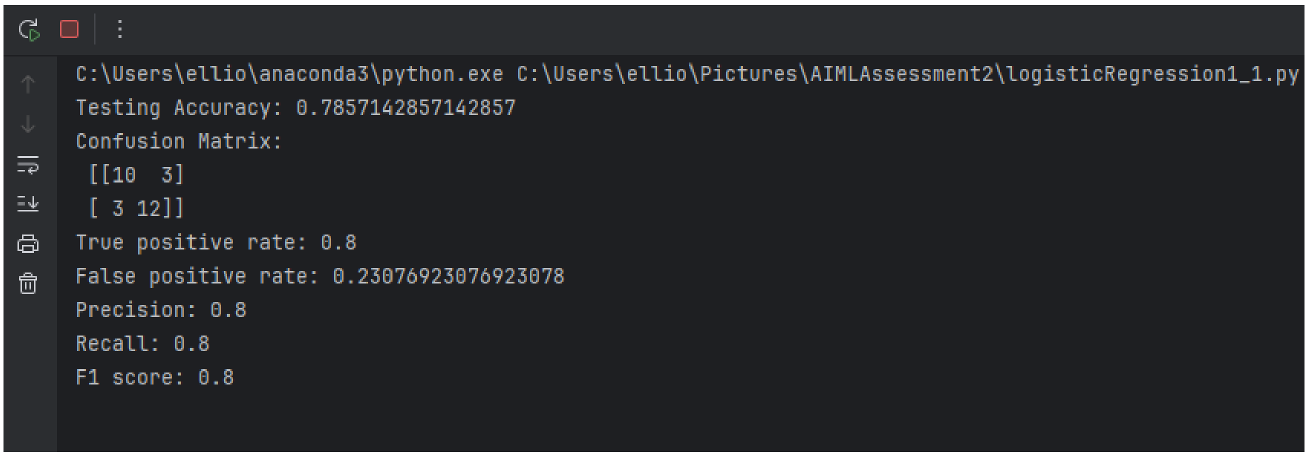


```
#Test the model on the testing data
x_testing = test_data[['nr_pix','aspect_ratio']]
y_testing = test_data['label']
test_predicts = (model.predict_proba(x_testing)[:, 1] > 0.5).astype(int)
test_accuracy = accuracy_score(y_testing, test_predicts)
conf_mat = confusion_matrix(y_testing,test_predicts)
print("Testing Accuracy:", test_accuracy)
print("Confusion Matrix:\n", conf_mat)
print("True positive rate:", conf_mat[1,1]/(conf_mat[1,1]+conf_mat[1,0]))
print("False positive rate:", conf_mat[0,1]/(conf_mat[0,1]+conf_mat[0,0]))
print("Precision:", precision_score(y_testing,test_predicts))
print("Recall:", recall_score(y_testing,test_predicts))
print("F1 score:", f1_score(y_testing,test_predicts))
```

I make a set of test predictions using a decision threshold of 0.5, and analyze the accuracy and various key metrics based on those test predictions. I use scikit-learn's metrics to pull the testing accuracy, confusion matrix, precision score, recall score and f1 score, and pull the true positive and false positive rates myself using the following formulas:

$$True\ Positive\ Rate = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$
$$False\ Positive\ Rate = \frac{False\ Positives}{False\ Positives + True\ Negatives}$$

Here we can see the results:



```
C:\Users\ellio\anaconda3\python.exe C:\Users\ellio\Pictures\AIMLAssessment2\logisticRegression1_1.py
Testing Accuracy: 0.7857142857142857
Confusion Matrix:
 [[10  3]
 [ 3 12]]
True positive rate: 0.8
False positive rate: 0.23076923076923078
Precision: 0.8
Recall: 0.8
F1 score: 0.8
```

There are a couple interesting things we can divulge from this information. First, the testing data set consists of a slight of majority positives (letters)(as shown by the confusion matrix). This is good for the random sample, as the dataset overall leans towards a slight majority positive, meaning we might see precision be slightly higher than it would be otherwise.

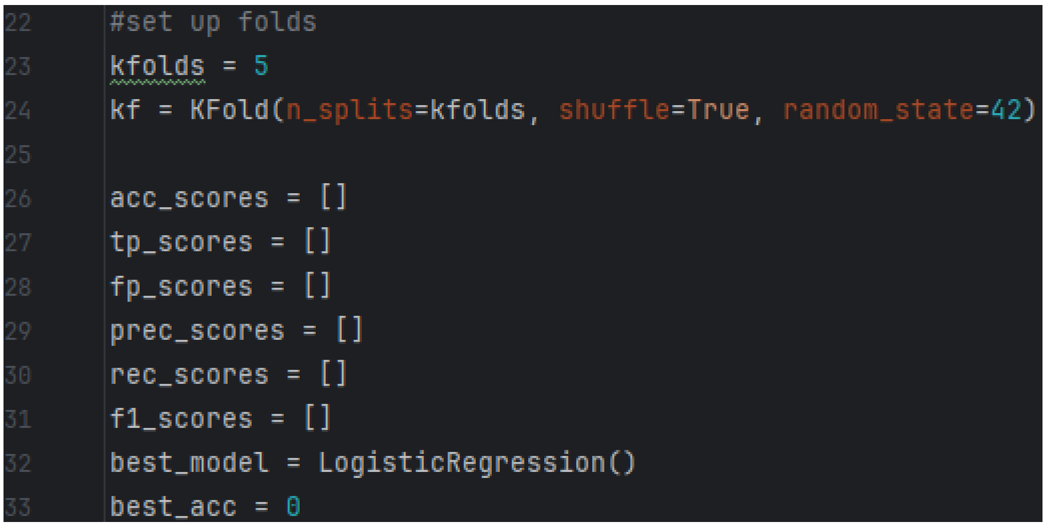
In terms of accuracy, there is room for improvement. ~78.6% is not subpar, but looking at our tpr and fpr statistics, we can see that model struggles with false positives(specificity), while the true positive rate (which reflects sensitivity) remains comparatively low.

Our precision and recall stands at 0.8, which means that most positives identified are actually positive, and that most positives in the dataset are found. Of course, given that precision and recall have the same value, the harmonic mean of the two F1, is also 0.8.

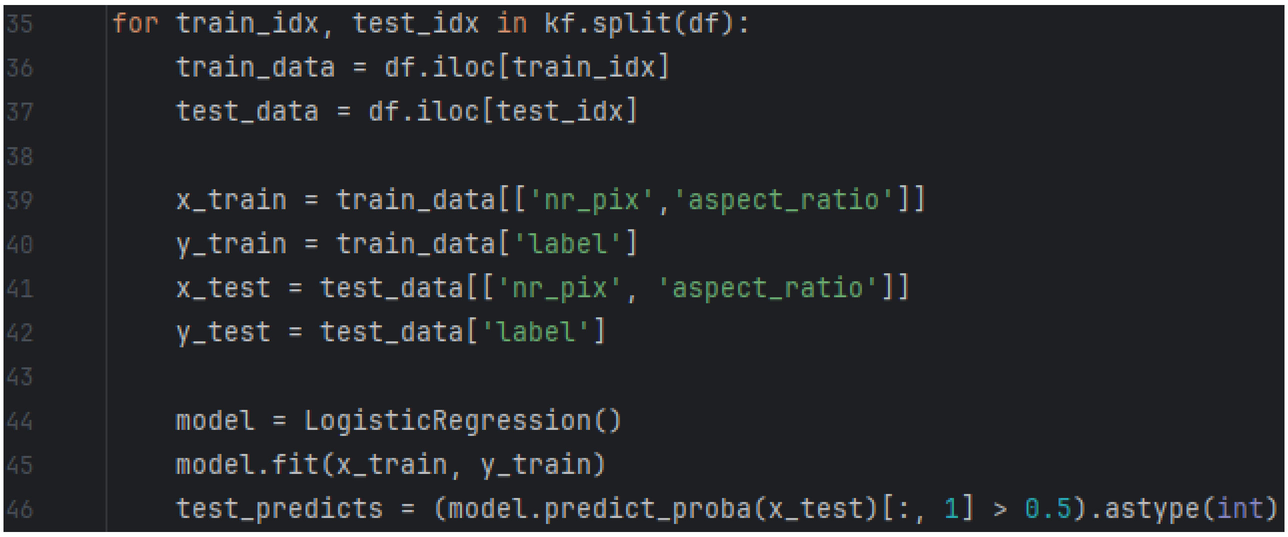
Section 1.2

Crossvalidation is a technique used for assessing how a model adapts to unseen data. It works by splitting the data into folds, and then using some of the folds for training, and some for testing. We will be conducting 5 fold crossvalidation in order to determine how well our model generalizes to unseen data.

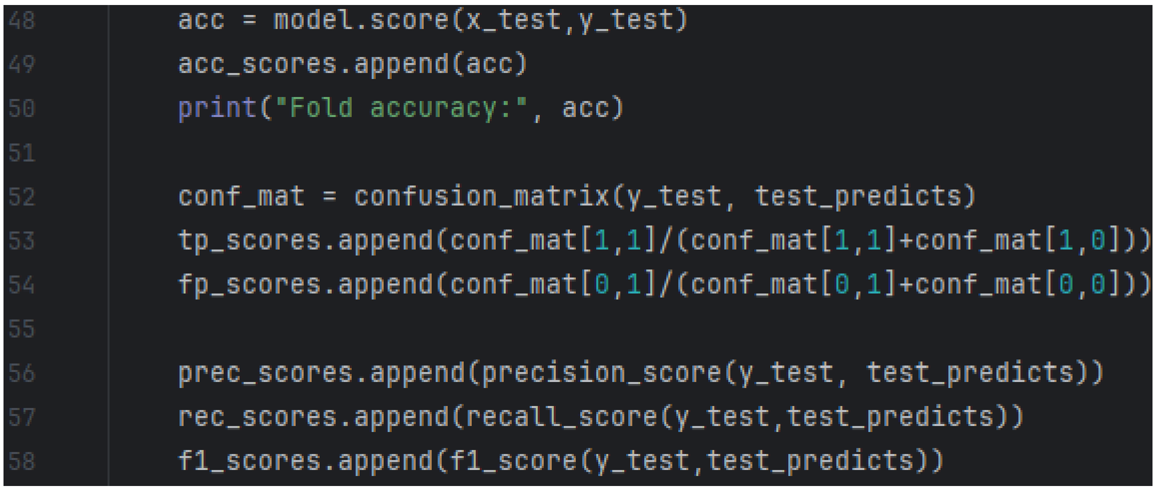
To perform crossvalidation, we first proceed as normal with labelling and randomizing the data. However, then we set our kfolds (5) and create arrays for our important metrics (for calculating the means later).


```
#set up folds
kfolds = 5
kf = KFold(n_splits=kfolds, shuffle=True, random_state=42)

acc_scores = []
tp_scores = []
fp_scores = []
prec_scores = []
rec_scores = []
f1_scores = []
best_model = LogisticRegression()
best_acc = 0
```

From there, we use our split to separate the data into 5 separate kfolds, and train our model on the training portion of the kfold, before making a series of test predictions on the testing set.[2]

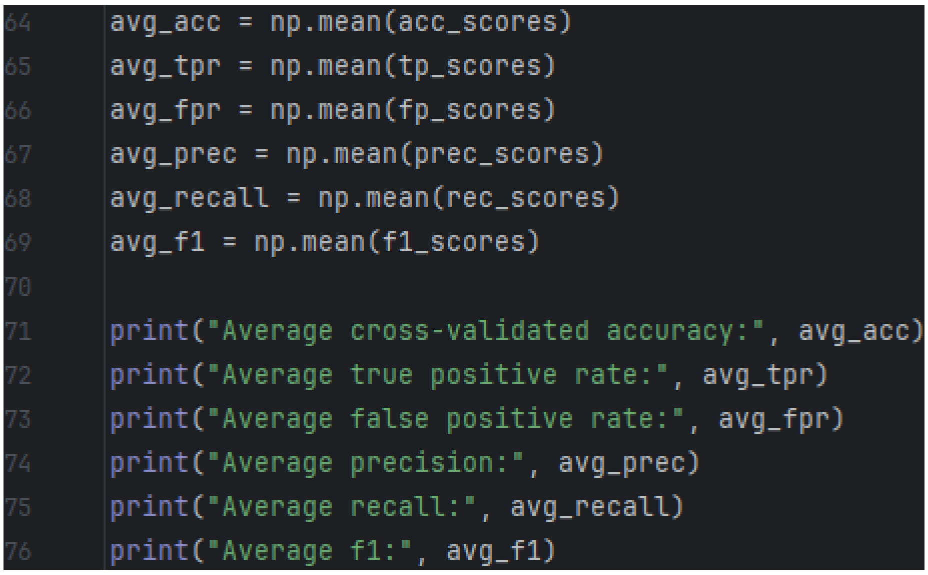

```
for train_idx, test_idx in kf.split(df):
    train_data = df.iloc[train_idx]
    test_data = df.iloc[test_idx]

    x_train = train_data[['nr_pix','aspect_ratio']]
    y_train = train_data['label']
    x_test = test_data[['nr_pix', 'aspect_ratio']]
    y_test = test_data['label']

    model = LogisticRegression()
    model.fit(x_train, y_train)
    test_predicts = (model.predict_proba(x_test)[:, 1] > 0.5).astype(int)
```

From there, we capture the same evaluation statistics, but this time we append them to an array so we can calculate the cross-validated mean of the statistics…



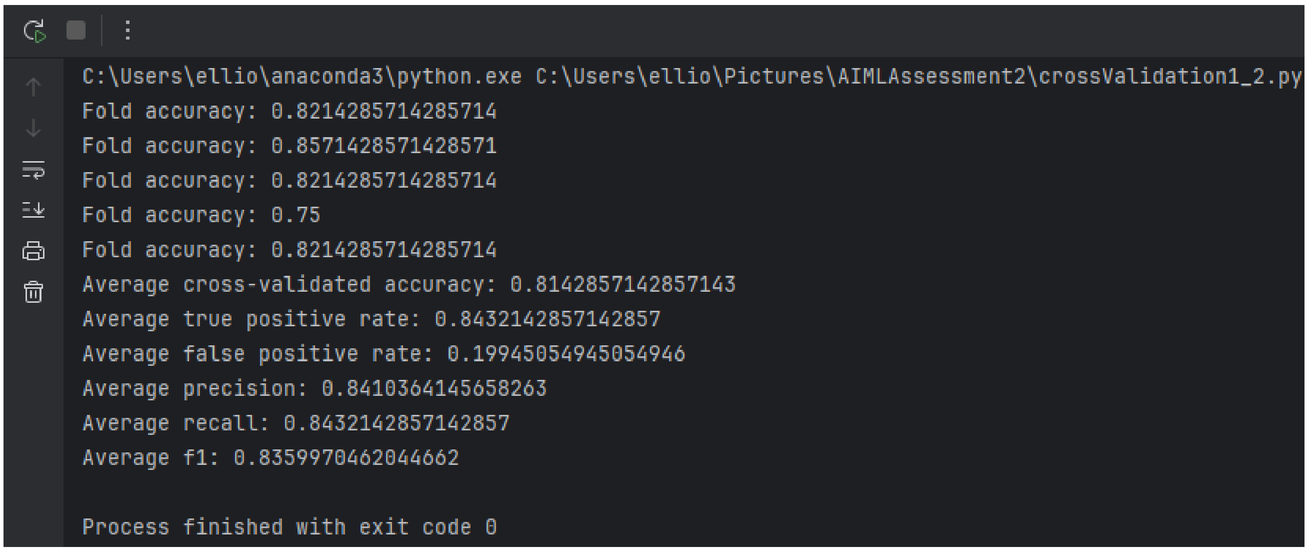
```python
        acc = model.score(x_test,y_test)
        acc_scores.append(acc)
        print("Fold accuracy:", acc)

        conf_mat = confusion_matrix(y_test, test_predicts)
        tp_scores.append(conf_mat[1,1]/(conf_mat[1,1]+conf_mat[1,0]))
        fp_scores.append(conf_mat[0,1]/(conf_mat[0,1]+conf_mat[0,0]))

        prec_scores.append(precision_score(y_test, test_predicts))
        rec_scores.append(recall_score(y_test,test_predicts))
        f1_scores.append(f1_score(y_test,test_predicts))
```

…which we calculate and output here:



```python
avg_acc = np.mean(acc_scores)
avg_tpr = np.mean(tp_scores)
avg_fpr = np.mean(fp_scores)
avg_prec = np.mean(prec_scores)
avg_recall = np.mean(rec_scores)
avg_f1 = np.mean(f1_scores)

print("Average cross-validated accuracy:", avg_acc)
print("Average true positive rate:", avg_tpr)
print("Average false positive rate:", avg_fpr)
print("Average precision:", avg_prec)
print("Average recall:", avg_recall)
print("Average f1:", avg_f1)
```

Below are the results:



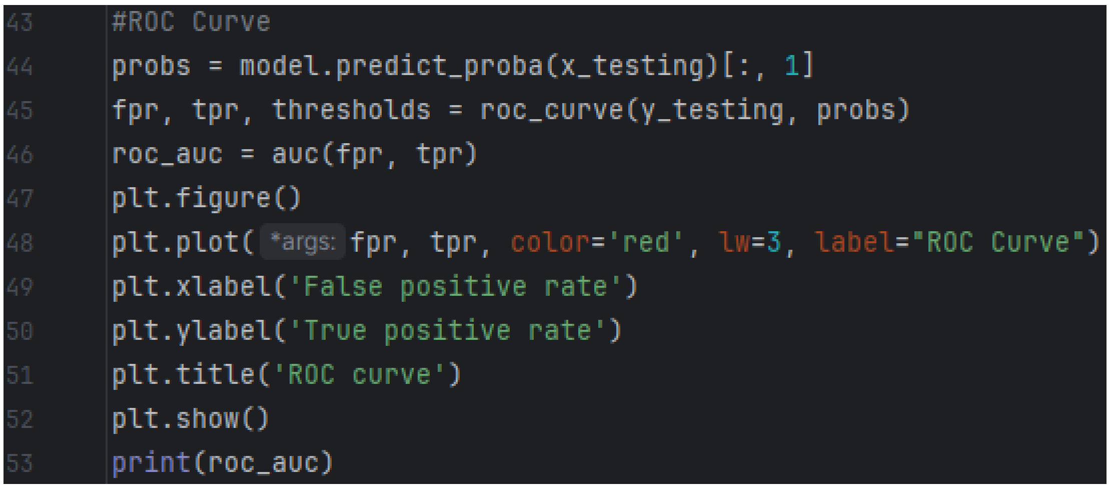
```
C:\Users\ellio\anaconda3\python.exe C:\Users\ellio\Pictures\AIMLAssessment2\crossValidation1_2.py
Fold accuracy: 0.8214285714285714
Fold accuracy: 0.8571428571428571
Fold accuracy: 0.8214285714285714
Fold accuracy: 0.75
Fold accuracy: 0.8214285714285714
Average cross-validated accuracy: 0.8142857142857143
Average true positive rate: 0.8432142857142857
Average false positive rate: 0.19945054945054946
Average precision: 0.8410364145658263
Average recall: 0.8432142857142857
Average f1: 0.8359970462044662

Process finished with exit code 0
```

Interestingly, it seems the accuracy of our initial model was fairly significantly lower than the average cross-validated accuracy. Our cross-validated models generally before significantly better, with key metrics improving across all areas. Our true positive rate/recall and false positive rate show both our sensitivity and specificity have improved, and our false positive rate has been reduced. The F1 score shows a 3.5% improvement, reflecting the overall increase.

Section 1.3

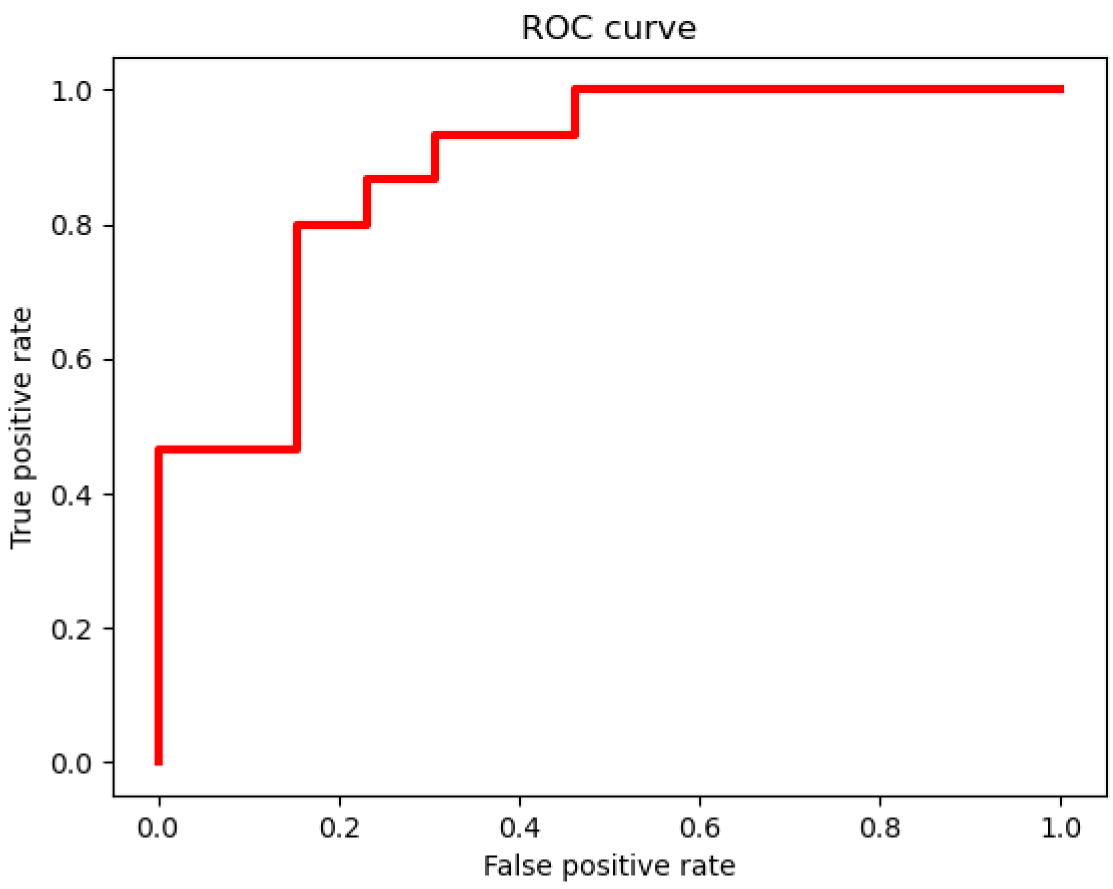
Now, we proceed to plot an ROC curve for our classifier, to evaluate it's performance across different classifcation thresholds. An ROC plots the true positive rate against the false positive rate at different thresholds. This helps to select the most optimal model.

I'm going to create one ROC curve for my initial (not cross-validated)model, and an additional one to evaluate the best model from my crossvalidation. I calculate a series of probabilites using the model on the unseen (testing) data, and then use scikit-learn's roc_curve function to calculate the false positive rate, true positive rate and thresholds (I also use auc to calculate the area under the curve).



```python
#ROC Curve
probs = model.predict_proba(x_testing)[:, 1]
fpr, tpr, thresholds = roc_curve(y_testing, probs)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot( *args: fpr, tpr, color='red', lw=3, label="ROC Curve")
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.show()
print(roc_auc)
```

Finally, I plot the graph use matplotlib.

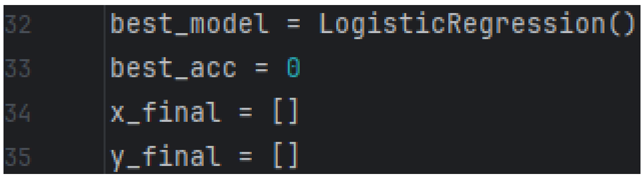


Our area under the curve was 0.882051282051282.

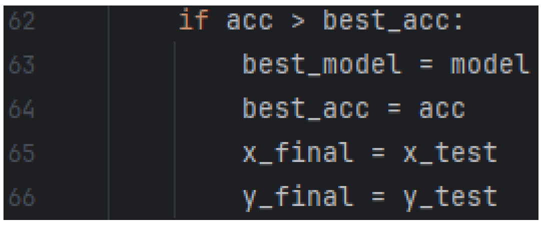
While our initial model's accuracy was fairly mediocre, it's roc curve is actually very strong. An AUC of 0.88 demonstrates that the model is useful for distinguishing between classes.

Now, let's move onto to our crossvalidated model.

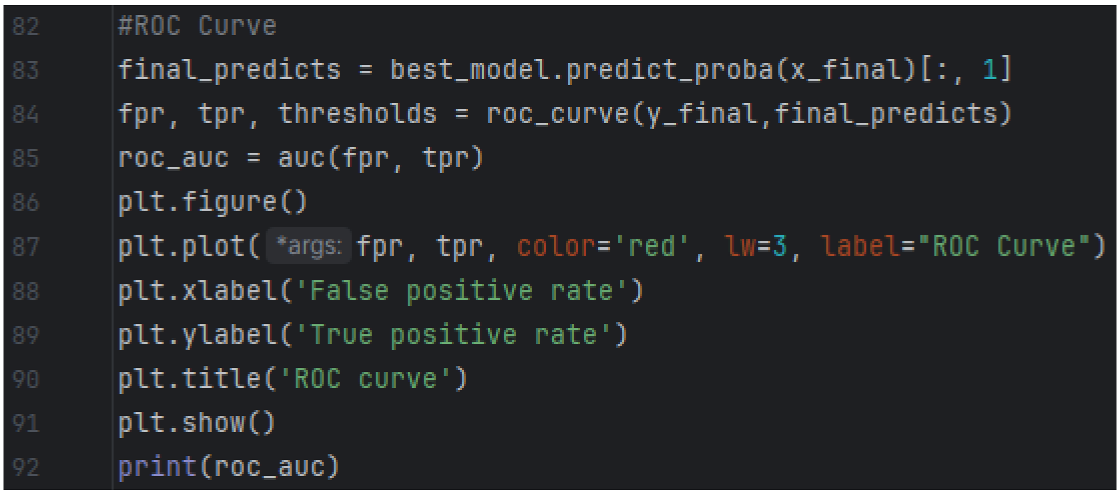
In order to capture the best model, I create variables to store the best accuracy, best model and their testing data.


```
best_model = LogisticRegression()
best_acc = 0
x_final = []
y_final = []
```

If the current accuracy of our model is greater that the best accuracy, I replace these variables with the values for our current model.

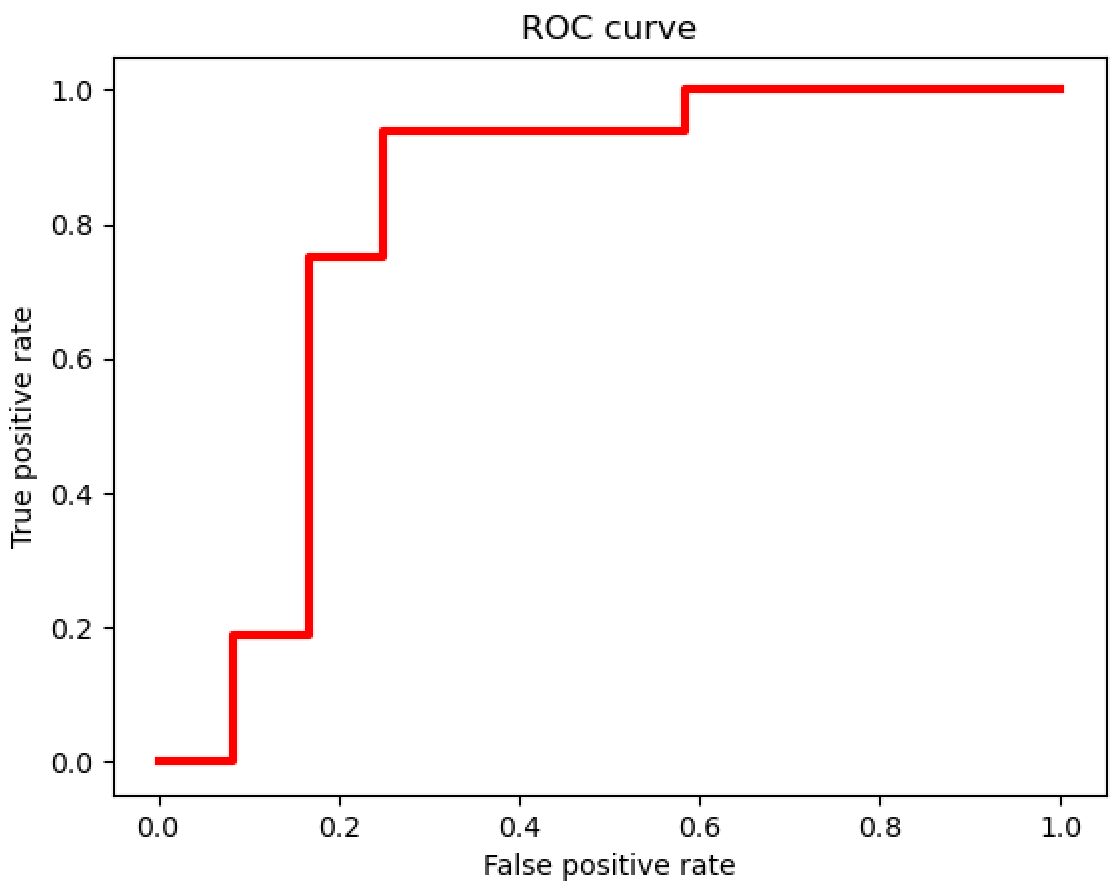

```
    if acc > best_acc:
        best_model = model
        best_acc = acc
        x_final = x_test
        y_final = y_test
```

From there, we can proceed as normal with calculating the ROC curve.



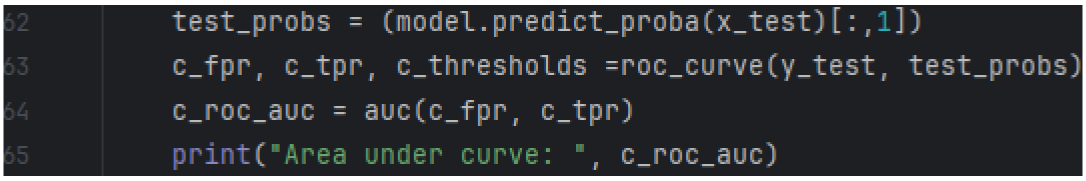
```python
#ROC Curve
final_predicts = best_model.predict_proba(x_final)[:, 1]
fpr, tpr, thresholds = roc_curve(y_final,final_predicts)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot( *args: fpr, tpr, color='red', lw=3, label="ROC Curve")
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.show()
print(roc_auc)
```

However, something interesting can be noticed when we run the code and plot the ROC curve.



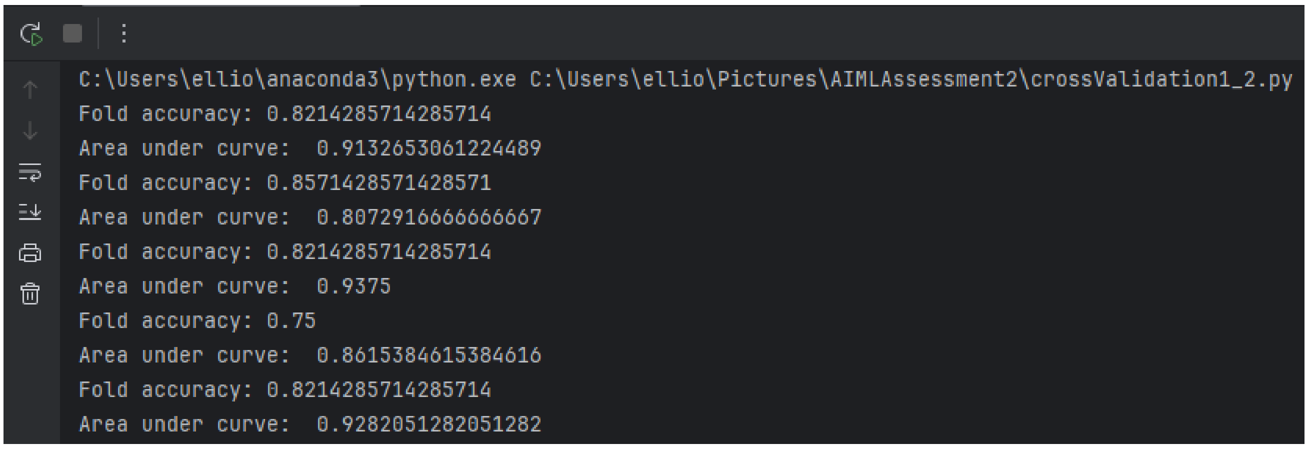
Area under the curve: 0.8072916666666667

Our ROC curve is actually significantly weaker than our initial model. After observing the data, I realized that this was because the dataset overall leaned positive. This meant that when our crossvalidated model over-predicted positives, it's accuracy increased, but it's ROC curve weakened. I decided to evaluate the ROC curves on the rest of the models, to see where they landed.



```python
    test_probs = (model.predict_proba(x_test)[:,1])
    c_fpr, c_tpr, c_thresholds =roc_curve(y_test, test_probs)
    c_roc_auc = auc(c_fpr, c_tpr)
    print("Area under curve: ", c_roc_auc)
```

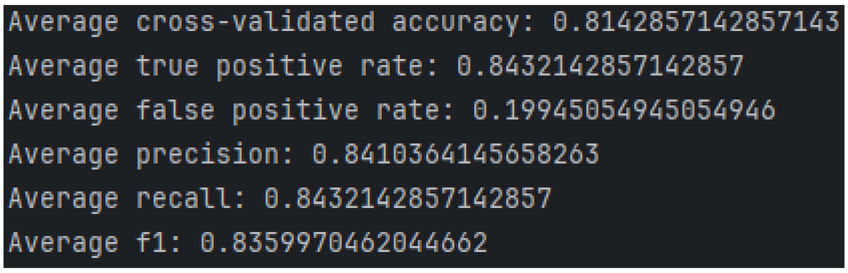
Adding this code, I could see the area under the curve next to the model accuracy.

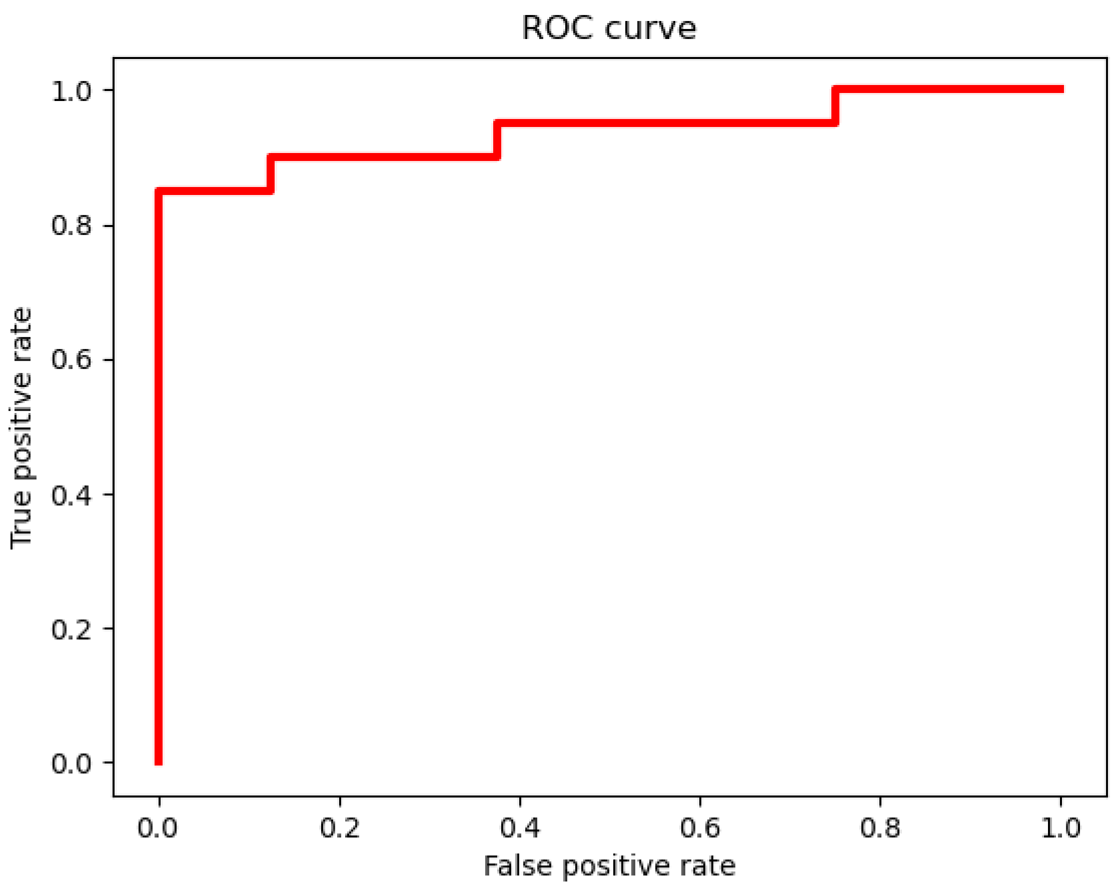

```
C:\Users\ellio\anaconda3\python.exe C:\Users\ellio\Pictures\AIMLAssessment2\crossValidation1_2.py
Fold accuracy: 0.8214285714285714
Area under curve:  0.9132653061224489
Fold accuracy: 0.8571428571428571
Area under curve:  0.8072916666666667
Fold accuracy: 0.8214285714285714
Area under curve:  0.9375
Fold accuracy: 0.75
Area under curve:  0.8615384615384616
Fold accuracy: 0.8214285714285714
Area under curve:  0.9282051282051282
```

This shows that while many of our models have inferior fold accuracy to the model we selected, they have much superior area under the curve. The trade-off was also fairly imbalanced, as losing single digits percentages in accuracy meant double digit gains in the area under the curve.

Based off this insight, I decided to select my best model using auc under the curve, instead of fold accuracy, to avoid rewarding the models for overpredicting positives.

Here is our final, best model:


```
Average cross-validated accuracy: 0.8142857142857143
Average true positive rate: 0.8432142857142857
Average false positive rate: 0.19945054945054946
Average precision: 0.8410364145658263
Average recall: 0.8432142857142857
Average f1: 0.8359970462044662
```



Area under the curve: 0.9375

Section 2

In this section, I will use the knn algorithm with 4 of my chosen features to classify letters (consisting of a and j),happy and faces, and exclamation marks. K-nearest-neighbours is an algorithm that works by classifying an algorithm based on a plurality vote of a number of it's 'neighbours' (training data points), which are determined by their euclidean distance from the testing data point.

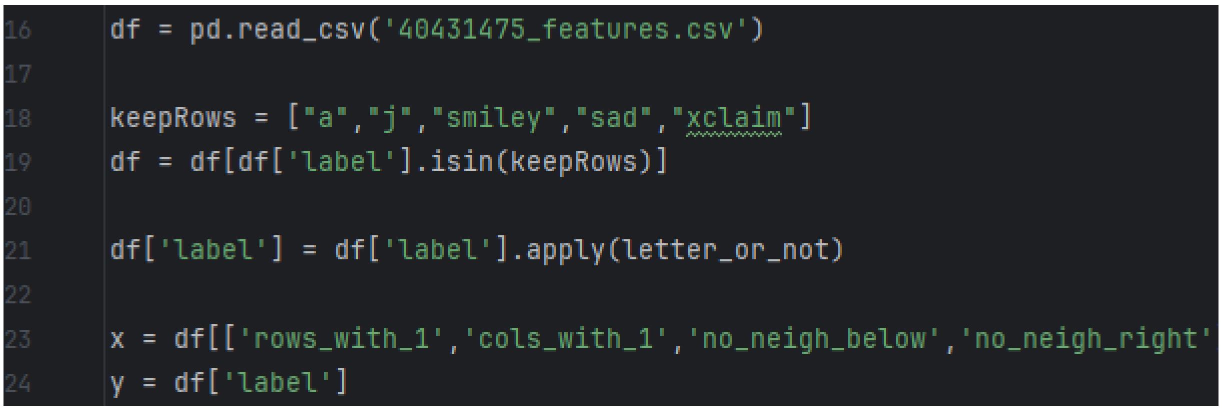
Section 2.1

To select 4 features from the dataset, I decided to draw on the data from my hypothesis tests (part of our classification is letters vs non-letters) in assignment 1, with 3 criteria in mind:

1. The T-statistic must be quite high, so that the points are far apart to make the euclidean distance significant
2. The p-value must be very low, so that there is a clear difference between letters and non-letters for classification
3. There must a large variety of different data-points, so that a point may be compared with several other datapoints if k is high (This rules out connected areas and eyes).

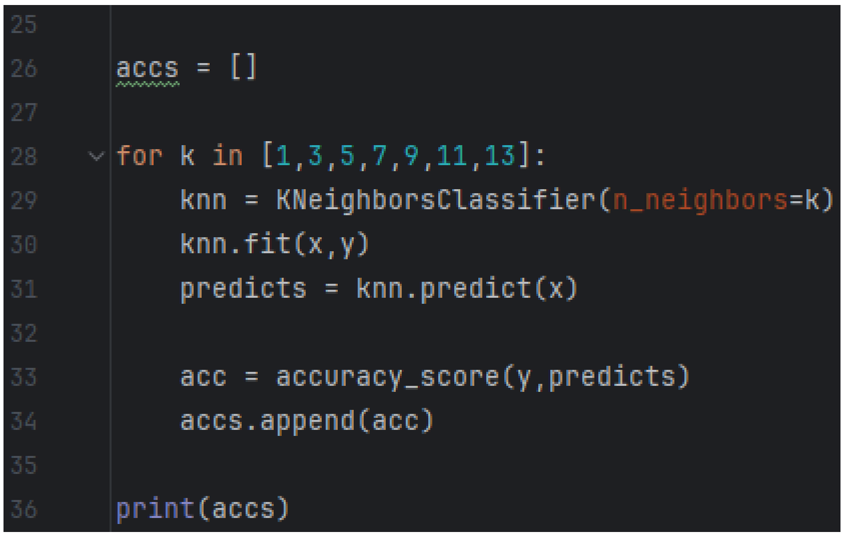
The four features which met my criteria were 'rows_with_1','cols_with_1', 'no_neigh_below' and 'no_neigh_right'.

Now I can proceed with dimensionality reduction, removing the letters we're not classifying, and the features we're not using.

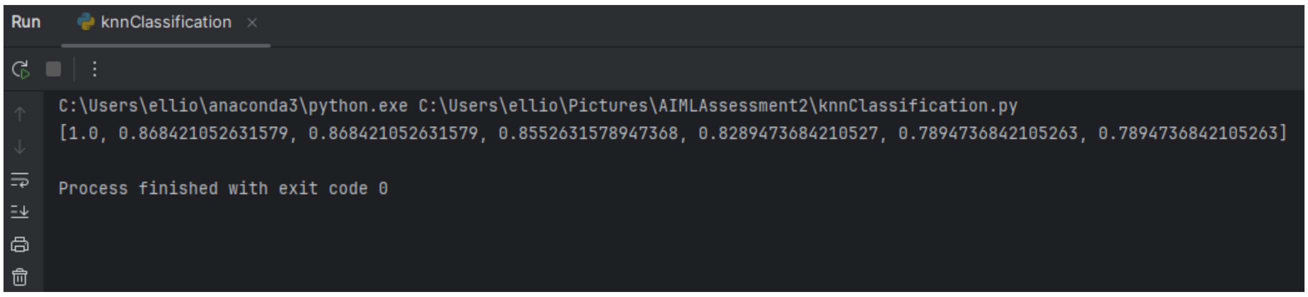


```
df = pd.read_csv('40431475_features.csv')

keepRows = ["a","j","smiley","sad","xclaim"]
df = df[df['label'].isin(keepRows)]

df['label'] = df['label'].apply(letter_or_not)

x = df[['rows_with_1','cols_with_1','no_neigh_below','no_neigh_right'
y = df['label']
```

From there, I loop through each k value, fitting the model and recording the accuracy. [3]



```

accs = []

for k in [1,3,5,7,9,11,13]:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(x,y)
    predicts = knn.predict(x)

    acc = accuracy_score(y,predicts)
    accs.append(acc)

print(accs)
```


Run knnClassification

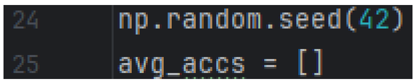
```
C:\Users\ellio\anaconda3\python.exe C:\Users\ellio\Pictures\AIMLAssessment2\knnClassification.py
[1.0, 0.868421052631579, 0.868421052631579, 0.8552631578947368, 0.8289473684210527, 0.7894736842105263, 0.7894736842105263]

Process finished with exit code 0
```

As we can see the accuracy decreases as the value of k increases. However, this is likely because we haven't created a split between the training data and testing data. With k =1, each point just finds the nearest point in euclidean space, which is guaranteed to be itself, and predicts that. Therefore, these accuracies can't really tell us anything useful about k.

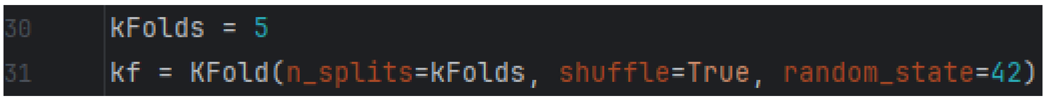
Section 2.2

To get a more accurate picture of the true best value of k, we're going to use cross-validation. [2]

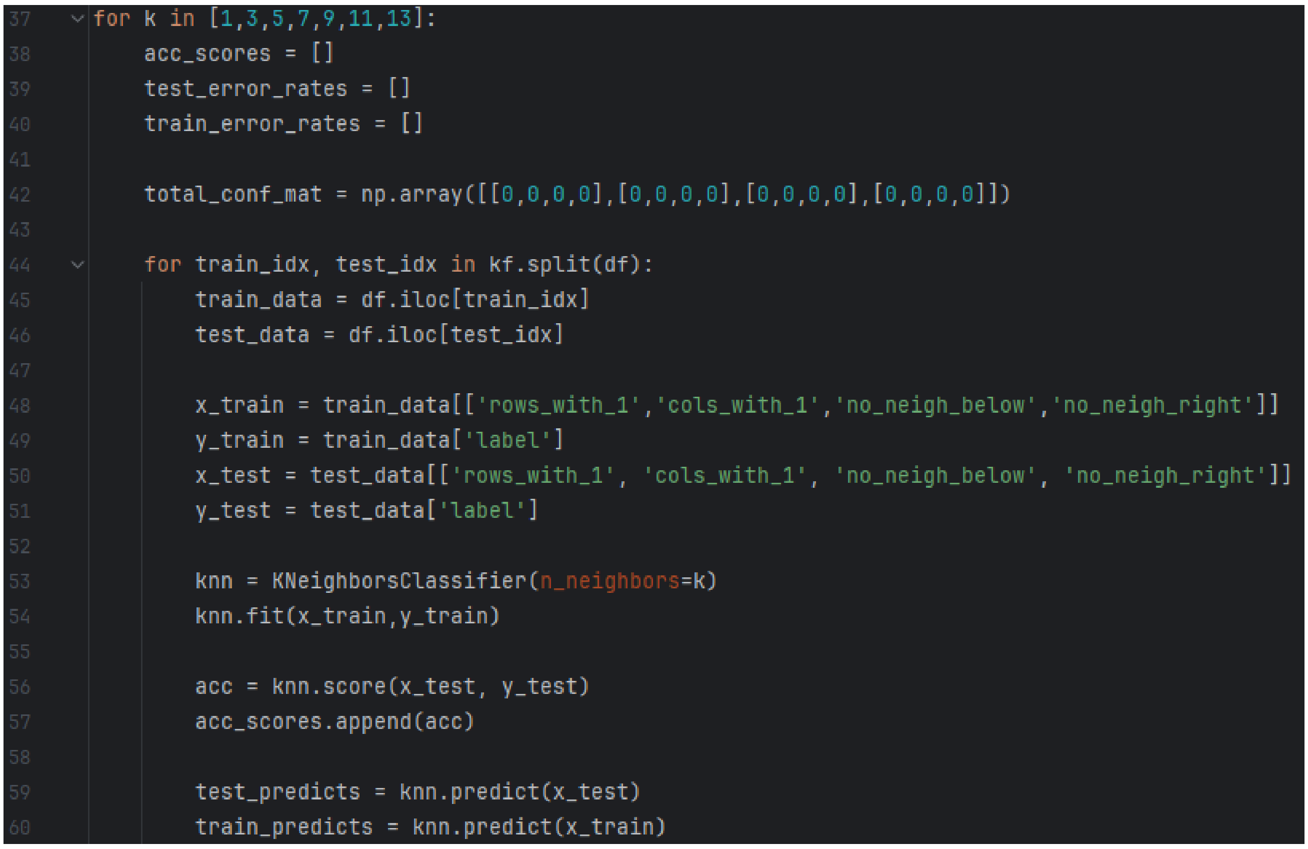
After reducing the dimensionality of the data, I set up an array to stores the average accuracies of the crossvalidations of each value of k, and then set up my kfolds.



```python
np.random.seed(42)
avg_accs = []
```

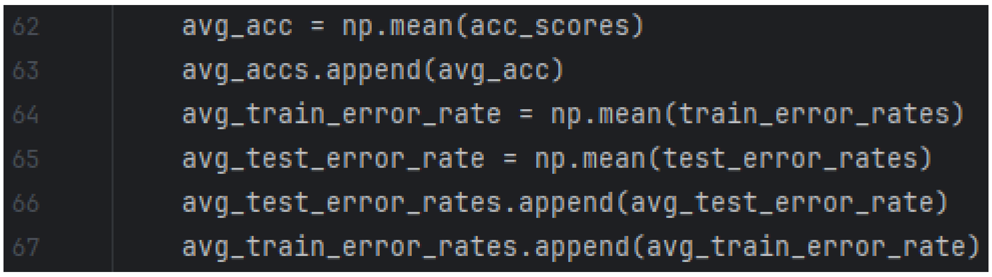


```python
kFolds = 5
kf = KFold(n_splits=kFolds, shuffle=True, random_state=42)
```



```python
for k in [1,3,5,7,9,11,13]:
    acc_scores = []
    test_error_rates = []
    train_error_rates = []

    total_conf_mat = np.array([[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])

    for train_idx, test_idx in kf.split(df):
        train_data = df.iloc[train_idx]
        test_data = df.iloc[test_idx]

        x_train = train_data[['rows_with_1','cols_with_1','no_neigh_below','no_neigh_right']]
        y_train = train_data['label']
        x_test = test_data[['rows_with_1', 'cols_with_1', 'no_neigh_below', 'no_neigh_right']]
        y_test = test_data['label']

        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(x_train,y_train)

        acc = knn.score(x_test, y_test)
        acc_scores.append(acc)

        test_predicts = knn.predict(x_test)
        train_predicts = knn.predict(x_train)
```

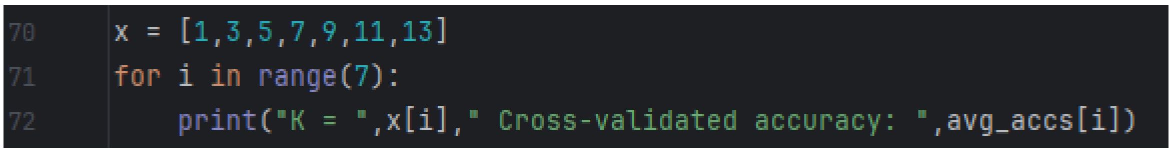
From there, I create a double loop. The inner loop represents the crossvalidation, fitting the model to kfold and then appending it to the list of accuracies.



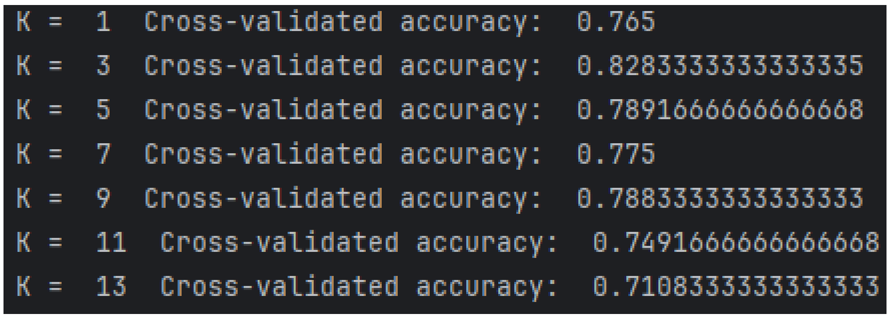
```
    avg_acc = np.mean(acc_scores)
    avg_accs.append(avg_acc)
    avg_train_error_rate = np.mean(train_error_rates)
    avg_test_error_rate = np.mean(test_error_rates)
    avg_test_error_rates.append(avg_test_error_rate)
    avg_train_error_rates.append(avg_train_error_rate)
```

The outer loop then takes that list of accuraties, calculates the average and the appends it to the list of averages, ensuring we get each crosss-validated accuracy.

Finally, we print our k values and the corresponding cross-validated accuracy.



```
x = [1,3,5,7,9,11,13]
for i in range(7):
    print("K = ",x[i]," Cross-validated accuracy: ",avg_accs[i])
```

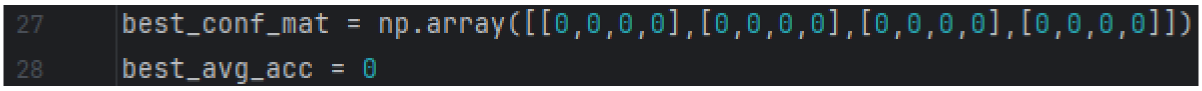


```
K =  1  Cross-validated accuracy:  0.765
K =  3  Cross-validated accuracy:  0.8283333333333335
K =  5  Cross-validated accuracy:  0.7891666666666668
K =  7  Cross-validated accuracy:  0.775
K =  9  Cross-validated accuracy:  0.7883333333333333
K =  11  Cross-validated accuracy:  0.7491666666666668
K =  13  Cross-validated accuracy:  0.7108333333333333
```

Now that we're testing on unseen data, we can observe the k = 3 seems to have the most consistent accuracy.

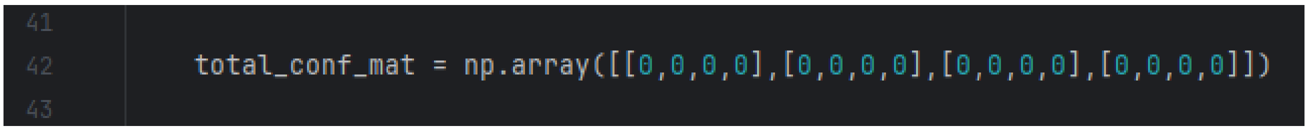
Section 2.3

To calculate the confusion matrix for the best value of k, I decided to aggregate all the confusion matrices into one confusion matrix for the best value of k. To do so, I first created variables to store the best confusion matrix and average accuracy across the cross-validation.



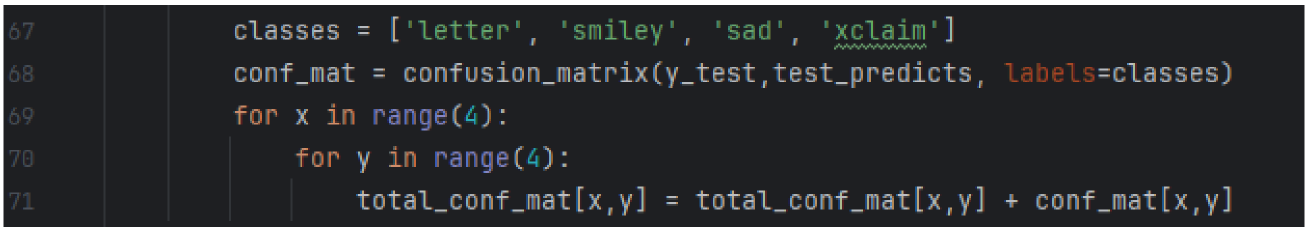
```
best_conf_mat = np.array([[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])
best_avg_acc = 0
```

Then I created a total confusion matrix inside the outer loop, to hold the aggregate confusion matrices.



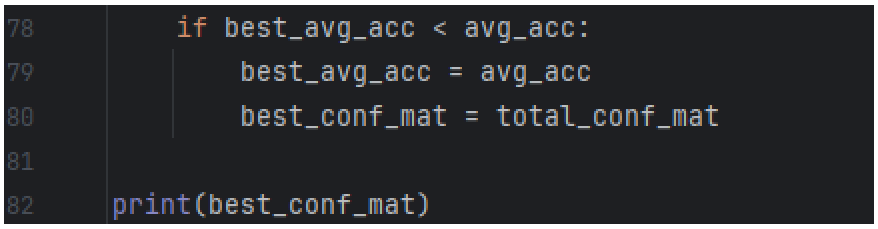
```

    total_conf_mat = np.array([[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]])

```

Inside the inner loop, I created a confusion matrix for the fold, and then added the values to the aggregate confusion matrix.



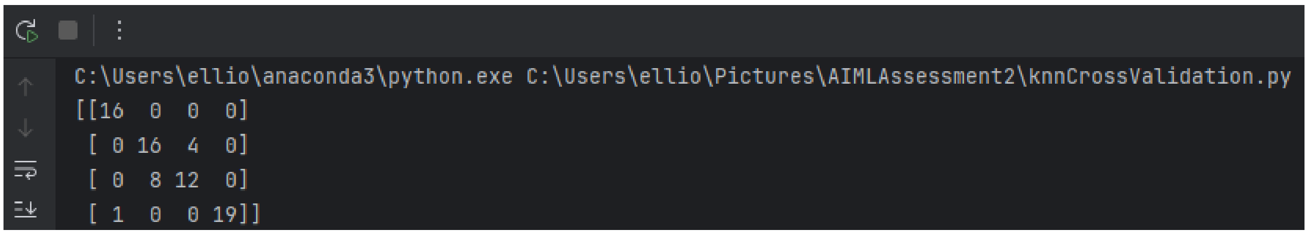
```
        classes = ['letter', 'smiley', 'sad', 'xclaim']
        conf_mat = confusion_matrix(y_test,test_predicts, labels=classes)
        for x in range(4):
            for y in range(4):
                total_conf_mat[x,y] = total_conf_mat[x,y] + conf_mat[x,y]
```

Finally, (in the outer loop) I stored the current aggregate confusion matrix if the average accuracy was higher, and at the end, I print the best confusion matrix.



```python
    if best_avg_acc < avg_acc:
        best_avg_acc = avg_acc
        best_conf_mat = total_conf_mat

print(best_conf_mat)
```

The results can be observed below.

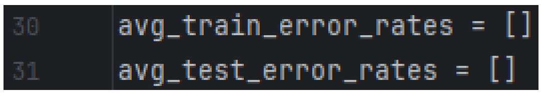


```
C:\Users\ellio\anaconda3\python.exe C:\Users\ellio\Pictures\AIMLAssessment2\knnCrossValidation.py
[[16  0  0  0]
 [ 0 16  4  0]
 [ 0  8 12  0]
 [ 1  0  0 19]]
```

The knn model very successfully classifies leeters and exclamation marks, with only one error between the two of them. However, the algorithm struggles with smiley faces and sad faces, as there are 16 smiley faces labeled sad and 8 sad faces labeled smiley. This makes sense, as they both share features like possessing two eyes and a nose.

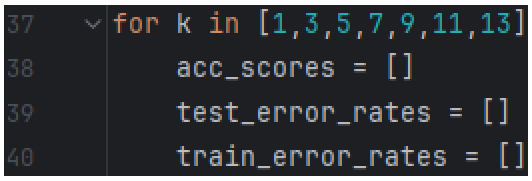
Section 2.4

In order to create the graph, I first created arrays to store the average training and testing error rates across the values of k.



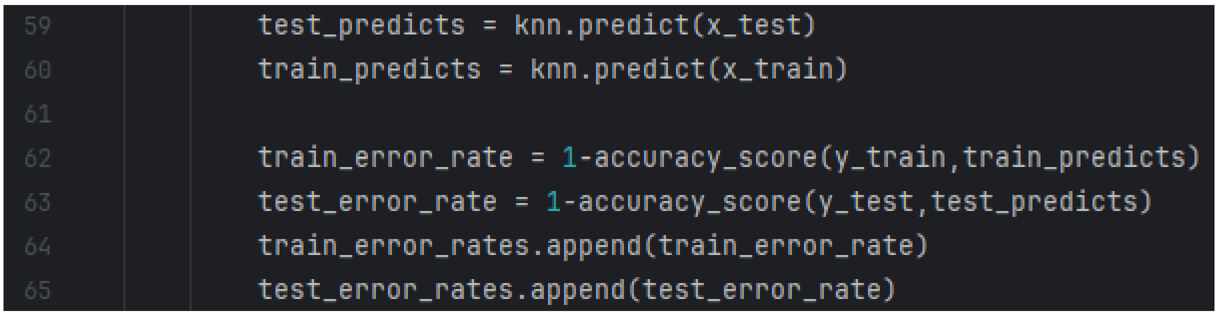
```python
avg_train_error_rates = []
avg_test_error_rates = []
```

Then, inside the outer loop, I created arrays to store the testing and training error rates across the kfolds.



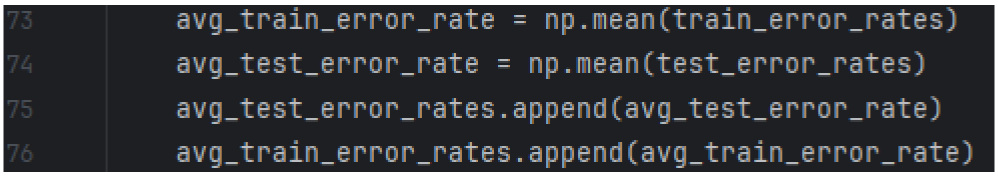
```python
for k in [1,3,5,7,9,11,13]:
    acc_scores = []
    test_error_rates = []
    train_error_rates = []
```

Next, I calculated the training and testing error rates for each kfold, and appended them to the arrays.



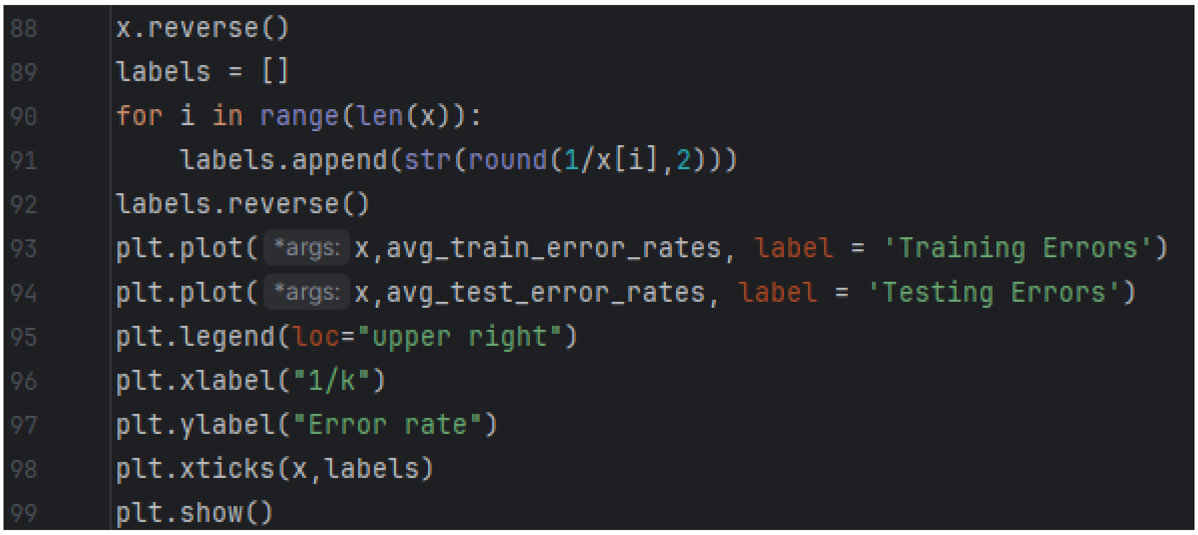
```python
        test_predicts = knn.predict(x_test)
        train_predicts = knn.predict(x_train)

        train_error_rate = 1-accuracy_score(y_train,train_predicts)
        test_error_rate = 1-accuracy_score(y_test,test_predicts)
        train_error_rates.append(train_error_rate)
        test_error_rates.append(test_error_rate)
```

I pulled the average from the arrays, and added to to the array of averages.



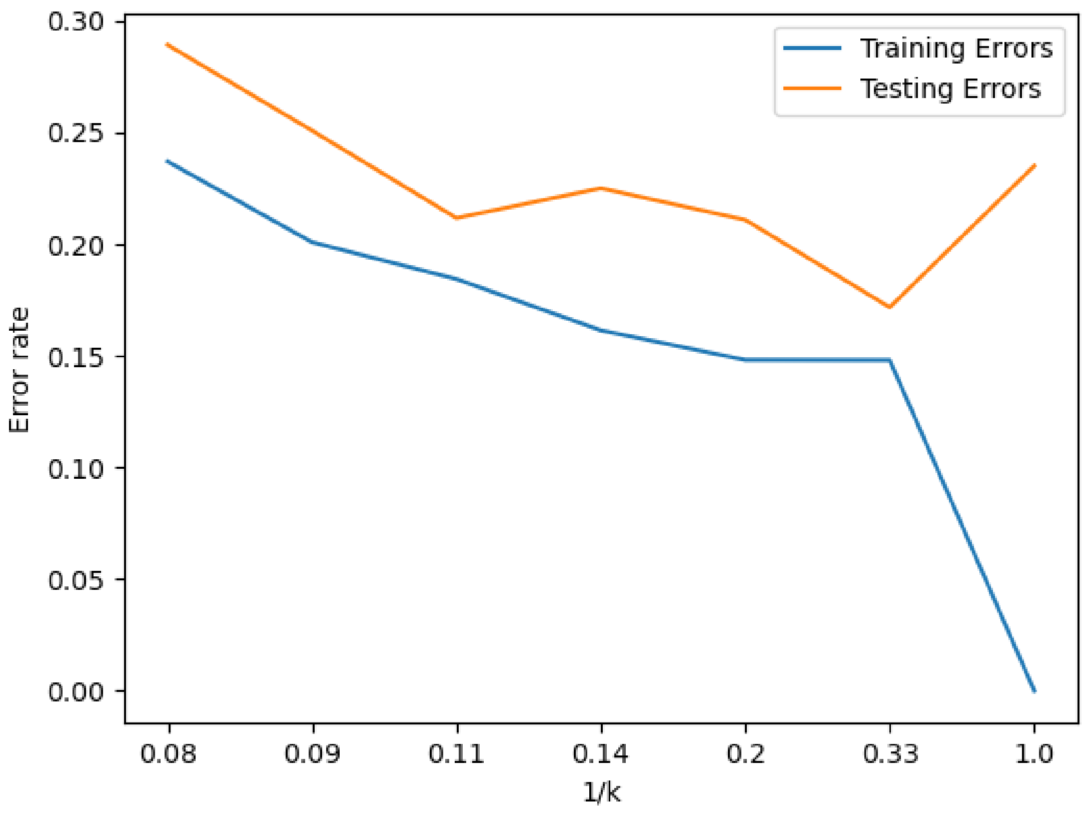
```
    avg_train_error_rate = np.mean(train_error_rates)
    avg_test_error_rate = np.mean(test_error_rates)
    avg_test_error_rates.append(avg_test_error_rate)
    avg_train_error_rates.append(avg_train_error_rate)
```

Now, I have the crossvalidated training and testing error rates for each value of k, which I can use to plot the graph using matplotlib.



```
x.reverse()
labels = []
for i in range(len(x)):
    labels.append(str(round(1/x[i],2)))
labels.reverse()
plt.plot( *args: x,avg_train_error_rates, label = 'Training Errors')
plt.plot( *args: x,avg_test_error_rates, label = 'Testing Errors')
plt.legend(loc="upper right")
plt.xlabel("1/k")
plt.ylabel("Error rate")
plt.xticks(x,labels)
plt.show()
```

Note that I reverse x, reverse the labels and set xticks. This is so the graph is scaled correctly (with 1 to 13) instead of 1/k, and two thirds of the graphs doesn't just have one data point.



The first note of import is that my suspicions about 2.1 are vindicated. While k=1 has a very high training accuracy (as each data point can find it's perfect match) it is overfitted and suffers in terms of it's testing accuracy.

As we discovered in 2.2, k = 3 has the superior testing accuracy, meaning it is the most useful model with regards to unseen data.

While increasing the number of k initially leads to an performance increase, it gradually leads to increase in both training and testing errors, before an extremely sharp spike after k = 9.

As such, a fairly low value of k seems to be the most useful for our dataset.

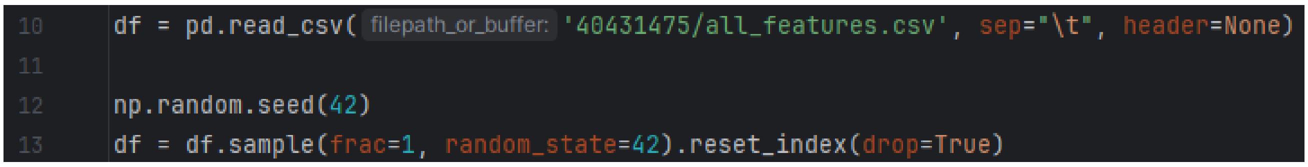
Section 3

In this section, I will perform random forest decision tree classification on a new set of data. I'll perform a grid search with the random forest parameters, and the evaluate the best of those models.

Random forest works by training a number of trees (with a maximum number of features) on boostrap samples randomly taken from the dataset. Predictions are then made via a plurality vote of all the trees in the random forest.

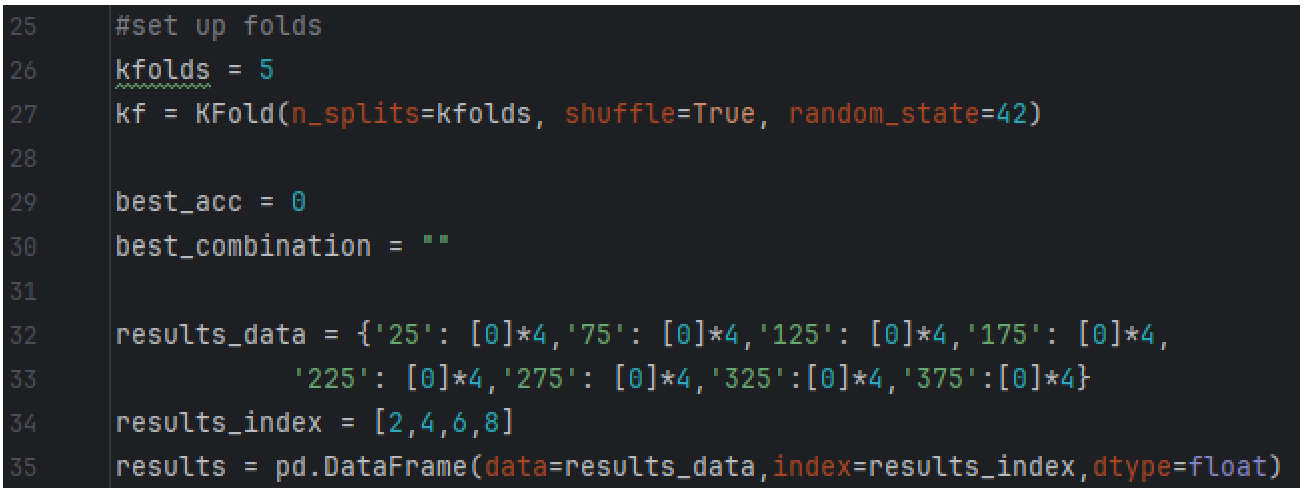
Section 3.1

To perform the gridsearch, first I import the dataset as normal (with some slight changes to accommodate the tab seperation and lack of headers in the csv). I don't actually need a label function, as this is 13 way classification so I can just use the existing labels. I also set the random seed for the bagging.



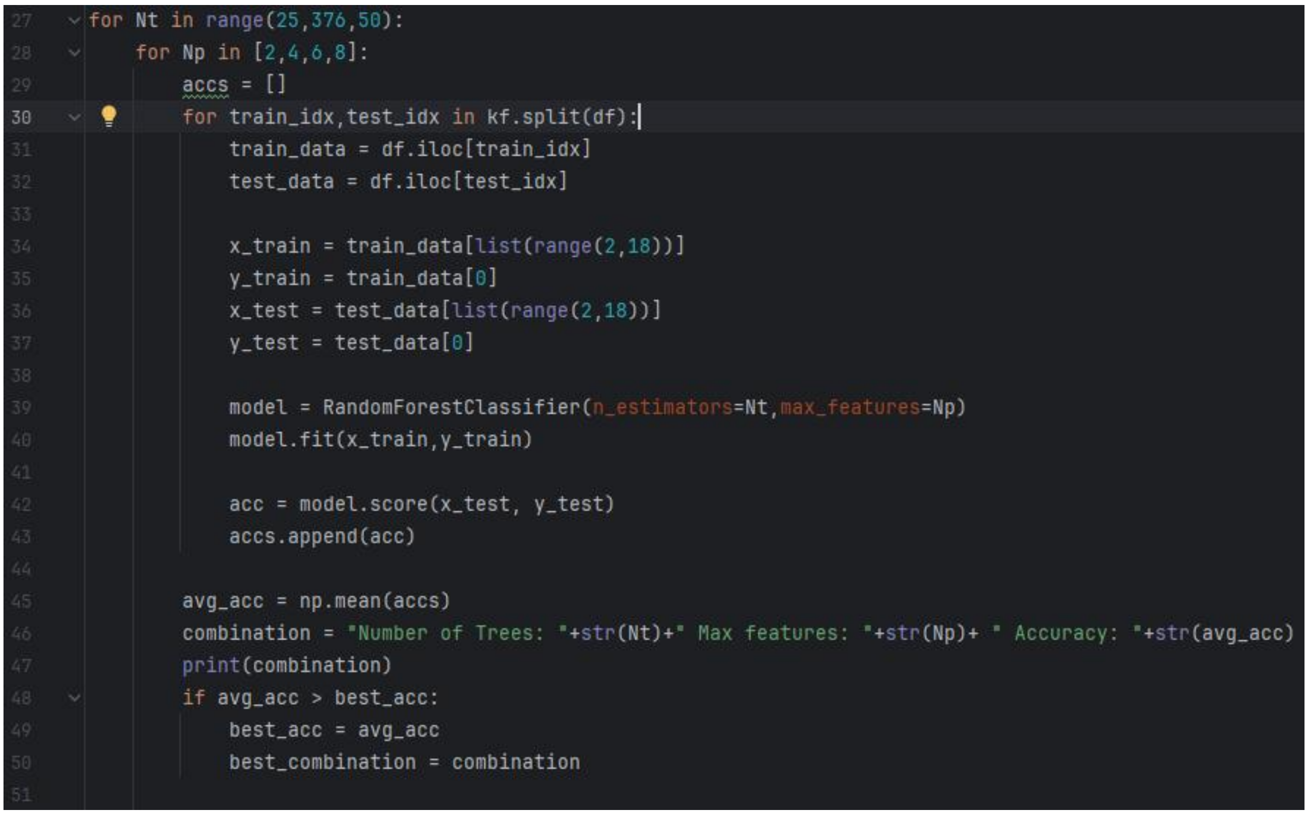
```python
df = pd.read_csv( filepath_or_buffer: '40431475/all_features.csv', sep="\t", header=None)

np.random.seed(42)
df = df.sample(frac=1, random_state=42).reset_index(drop=True)
```

Next, I set up my k-folds, two variables to store the best average accuracy and best combination, and a dataframe to store my results.



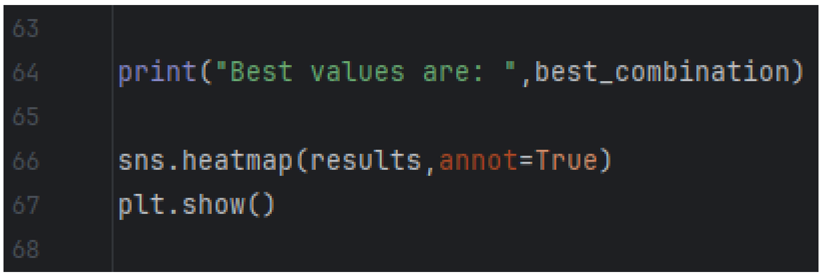
```python
#set up folds
kfolds = 5
kf = KFold(n_splits=kfolds, shuffle=True, random_state=42)

best_acc = 0
best_combination = ""

results_data = {'25': [0]*4,'75': [0]*4,'125': [0]*4,'175': [0]*4,
            '225': [0]*4,'275': [0]*4,'325':[0]*4,'375':[0]*4}
results_index = [2,4,6,8]
results = pd.DataFrame(data=results_data,index=results_index,dtype=float)
```

From there, I use three loops to complete the gridsearch. The outer loop keeps track of the number of trees, and the middle loop keep track of the predictor number. The final loop conducts the cross-validation, training the model with parameters specified by the outer loops on the k-fold. The accuracies are then recorded, the average of those accuracies are calculated and appended to the results dataframe. If the accuracy is better than the current best accuracy, it becomes the current best accuracy. [4]

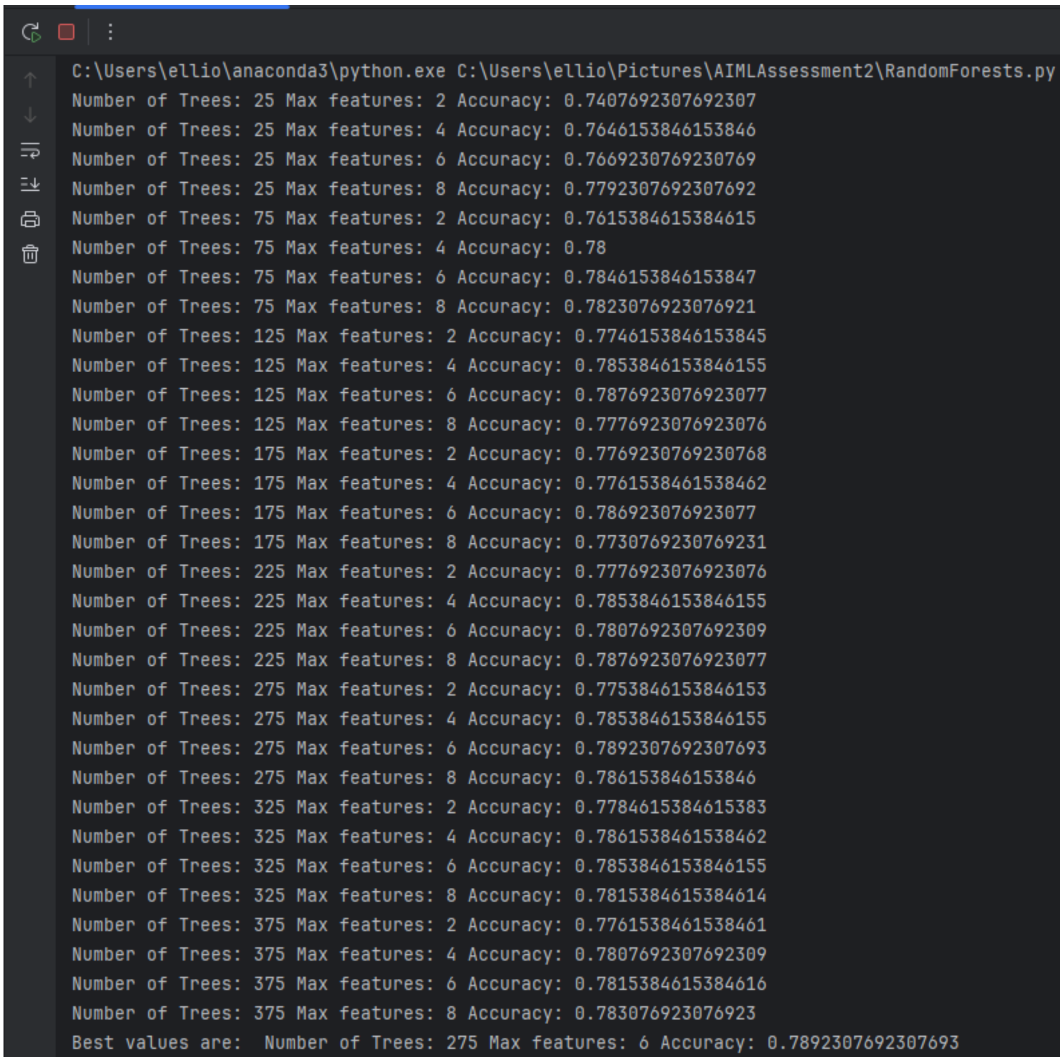


```
for Nt in range(25,376,50):
    for Np in [2,4,6,8]:
        accs = []
        for train_idx,test_idx in kf.split(df):
            train_data = df.iloc[train_idx]
            test_data = df.iloc[test_idx]

            x_train = train_data[list(range(2,18))]
            y_train = train_data[0]
            x_test = test_data[list(range(2,18))]
            y_test = test_data[0]

            model = RandomForestClassifier(n_estimators=Nt,max_features=Np)
            model.fit(x_train,y_train)

            acc = model.score(x_test, y_test)
            accs.append(acc)

        avg_acc = np.mean(accs)
        combination = "Number of Trees: "+str(Nt)+" Max features: "+str(Np)+ " Accuracy: "+str(avg_acc)
        print(combination)
        if avg_acc > best_acc:
            best_acc = avg_acc
            best_combination = combination

```

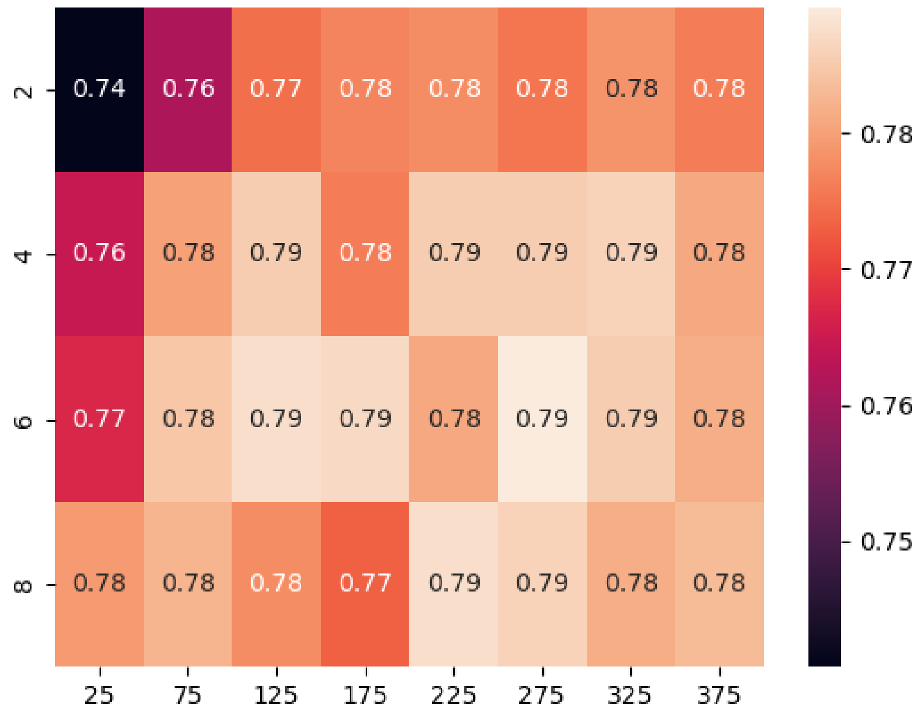
Finally, the best combination is printed, and heatmap of the results is created[5].



```

print("Best values are: ",best_combination)

sns.heatmap(results,annot=True)
plt.show()

```



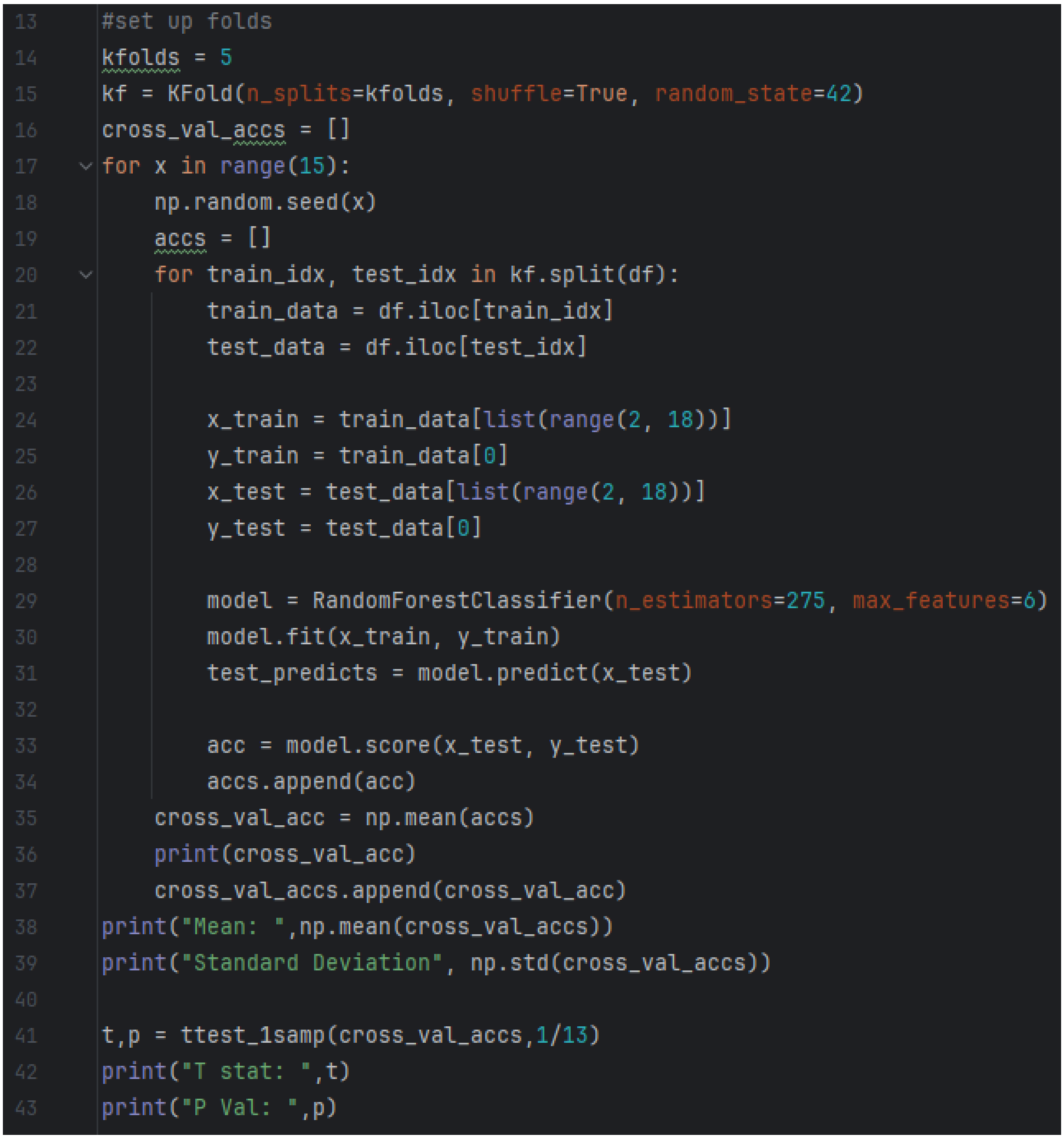
```
C:\Users\ellio\anaconda3\python.exe C:\Users\ellio\Pictures\AIMLAssessment2\RandomForests.py
Number of Trees: 25 Max features: 2 Accuracy: 0.7407692307692307
Number of Trees: 25 Max features: 4 Accuracy: 0.7646153846153846
Number of Trees: 25 Max features: 6 Accuracy: 0.7669230769230769
Number of Trees: 25 Max features: 8 Accuracy: 0.7792307692307692
Number of Trees: 75 Max features: 2 Accuracy: 0.7615384615384615
Number of Trees: 75 Max features: 4 Accuracy: 0.78
Number of Trees: 75 Max features: 6 Accuracy: 0.7846153846153847
Number of Trees: 75 Max features: 8 Accuracy: 0.7823076923076921
Number of Trees: 125 Max features: 2 Accuracy: 0.7746153846153845
Number of Trees: 125 Max features: 4 Accuracy: 0.7853846153846155
Number of Trees: 125 Max features: 6 Accuracy: 0.7876923076923077
Number of Trees: 125 Max features: 8 Accuracy: 0.7776923076923076
Number of Trees: 175 Max features: 2 Accuracy: 0.7769230769230768
Number of Trees: 175 Max features: 4 Accuracy: 0.7761538461538462
Number of Trees: 175 Max features: 6 Accuracy: 0.786923076923077
Number of Trees: 175 Max features: 8 Accuracy: 0.7730769230769231
Number of Trees: 225 Max features: 2 Accuracy: 0.7776923076923076
Number of Trees: 225 Max features: 4 Accuracy: 0.7853846153846155
Number of Trees: 225 Max features: 6 Accuracy: 0.7807692307692309
Number of Trees: 225 Max features: 8 Accuracy: 0.7876923076923077
Number of Trees: 275 Max features: 2 Accuracy: 0.7753846153846153
Number of Trees: 275 Max features: 4 Accuracy: 0.7853846153846155
Number of Trees: 275 Max features: 6 Accuracy: 0.7892307692307693
Number of Trees: 275 Max features: 8 Accuracy: 0.786153846153846
Number of Trees: 325 Max features: 2 Accuracy: 0.7784615384615383
Number of Trees: 325 Max features: 4 Accuracy: 0.7861538461538462
Number of Trees: 325 Max features: 6 Accuracy: 0.7853846153846155
Number of Trees: 325 Max features: 8 Accuracy: 0.7815384615384614
Number of Trees: 375 Max features: 2 Accuracy: 0.7761538461538461
Number of Trees: 375 Max features: 4 Accuracy: 0.7807692307692309
Number of Trees: 375 Max features: 6 Accuracy: 0.7815384615384616
Number of Trees: 375 Max features: 8 Accuracy: 0.783076923076923
Best values are:  Number of Trees: 275 Max features: 6 Accuracy: 0.7892307692307693
```



As stated by the code and shown by the heatmap, 275 trees with 6 maximum features appears to have the highest accuracy. As the number of trees increases, the crossvalidated accuracy improves. However, this has diminishing returns. 25 to 50 trees is a massive improvement, but from 275 – 375, the accuracy actually decreases. This is atypical for random forest, so is likely random noise instead of an actual decrease in performance, but it still indicates that we're not getting "bang for our buck" (pardon the informality) in terms of the additional trees and resources. For the maximum features, the accuracy sees a peak at 6 for the predictor number. This indicates that trees with lower than 6 features are underfit, being too simple to accurately capture the trend, and trees with more than six features are overfit, complex enough that they capture the noise of the data and don't generalize well.

Section 3.2

Finally, we're going to refit the model fifteen times, report the mean and standard distribution, and conduct a hypothesis test to see if it performs better than chance.



```python
#set up folds
kfolds = 5
kf = KFold(n_splits=kfolds, shuffle=True, random_state=42)
cross_val_accs = []
for x in range(15):
    np.random.seed(x)
    accs = []
    for train_idx, test_idx in kf.split(df):
        train_data = df.iloc[train_idx]
        test_data = df.iloc[test_idx]

        x_train = train_data[list(range(2, 18))]
        y_train = train_data[0]
        x_test = test_data[list(range(2, 18))]
        y_test = test_data[0]

        model = RandomForestClassifier(n_estimators=275, max_features=6)
        model.fit(x_train, y_train)
        test_predicts = model.predict(x_test)

        acc = model.score(x_test, y_test)
        accs.append(acc)
    cross_val_acc = np.mean(accs)
    print(cross_val_acc)
    cross_val_accs.append(cross_val_acc)
print("Mean: ",np.mean(cross_val_accs))
print("Standard Deviation", np.std(cross_val_accs))

t,p = ttest_1samp(cross_val_accs,1/13)
print("T stat: ",t)
print("P Val: ",p)
```

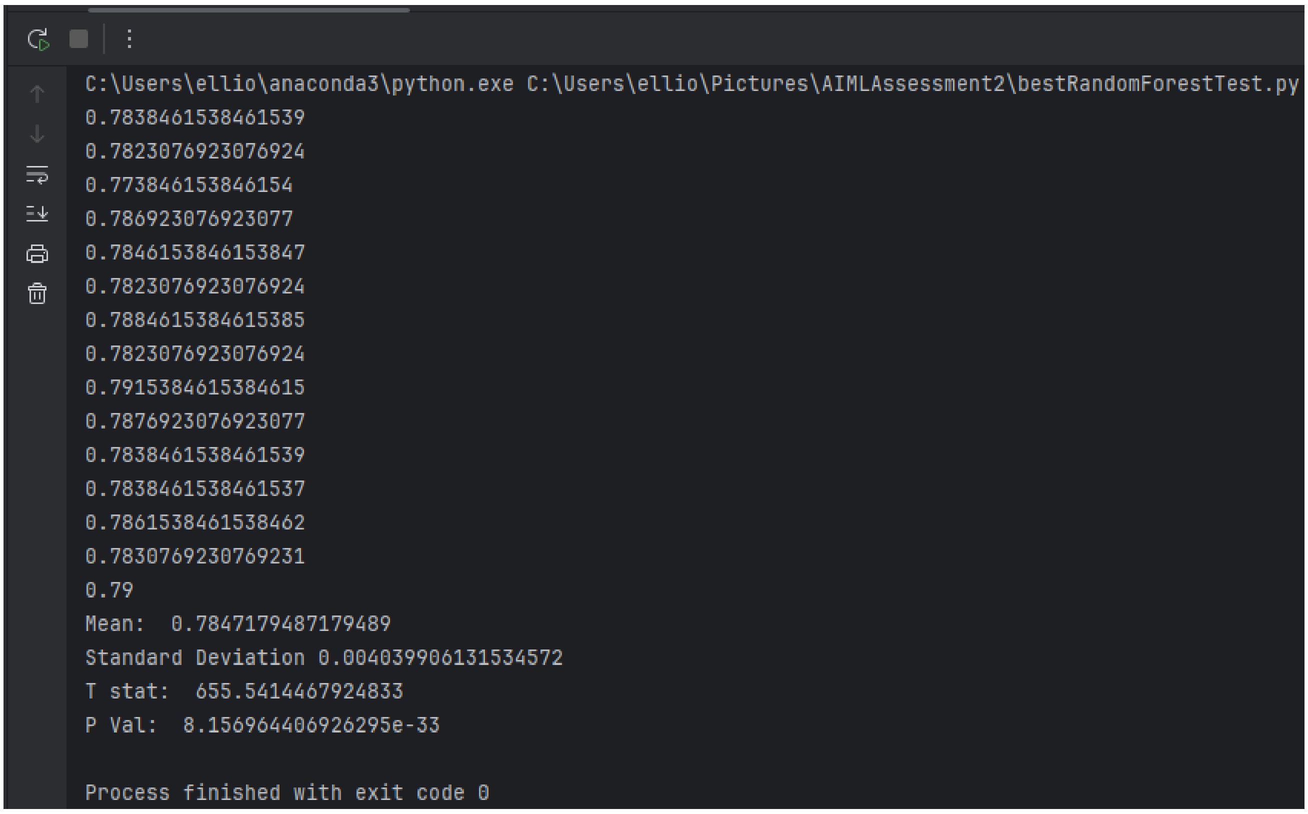
Our code is much the same as last time, the key difference being that we save our cross-validated accuracies, calculate the mean and standard deviation using numpy, and then conduct a hypothesis test using scipy's ttest_1samp (we set the popmean to 1/13 to represent random guessing).

Here are our null an alternate hypothesis:

H0: Our model's predictions will be successful 1/13th or less of the time

HA: Our model's prediction will be successful more than 13/th of the time

We will set our significance level to 0.05.



```
C:\Users\ellio\anaconda3\python.exe C:\Users\ellio\Pictures\AIMLAssessment2\bestRandomForestTest.py
0.7838461538461539
0.7823076923076924
0.773846153846154
0.786923076923077
0.7846153846153847
0.7823076923076924
0.7884615384615385
0.7823076923076924
0.7915384615384615
0.7876923076923077
0.7838461538461539
0.7838461538461537
0.7861538461538462
0.7830769230769231
0.79
Mean:  0.7847179487179489
Standard Deviation 0.004039906131534572
T stat:  655.5414467924833
P Val:  8.156964406926295e-33

Process finished with exit code 0
```

Our mean is quite strong, at 0.78, with an extremely low standard deviation. This indicates our model has strong, consistent performance in terms of identifying conducting the 13 way classification. Given our T-statistic is incredibly large (655) and our p value is far, far below our significance level, we can safely reject the null hypothesis in favour of the alternate hypothesis and conclude that our model classifies the categories significantly better than random chance.

Conclusions

In this analysis, I have arrived at several conclusions. Letters vs non-letters can be categorized consistently with logistic regression, as our final logistic regression model demonstrates both strong accuracy and an ability to separate the two classes (shown by it's ROC curve). A higher accuracy in a logisitic regression model can result in a lower roc_curve, and a worse model. Therefore, it's better to determine the best logistic regression model with the area under the curve. The optimal number of k for four way classification in our dataset is 3. In our graphing and our analysis, this number led to the most optimal results. Beyond k = 9, all performance becomes very sharply worse. 275 trees with 6 as predictor number has the greatest performance, which is consistent when the model is trained multiple times. In general, increasing the number of trees and maximum features leads to sharp but diminishing returns. Finally, our random forest ML model performs statistically significantly better than random chance and allows us to reject the null hypothesis.

References

[1]141_logistic_regression_iris.ipynb: CSC2062: Introduction to Artificial Intelligence and Machine Learning (2241_SPR)

[2] 161_knn_crossvalidation.ipynb: CSC2062: Introduction to Artificial Intelligence and Machine Learning (2241_SPR)

[3] 151_knn_example.ipynb: CSC2062: Introduction to Artificial Intelligence and Machine Learning (2241_SPR)

[4] RandomForestClassifier — scikit-learn 1.6.1 documentation

[5] seaborn.heatmap — seaborn 0.13.2 documentation