Chaithra Kopparam Cheluvaiah
SUID- 326926205
ckoppara@syr.edu

HW08 - Naive Bayes and Decision tree for handwriting recognition

Section 01: Introduction

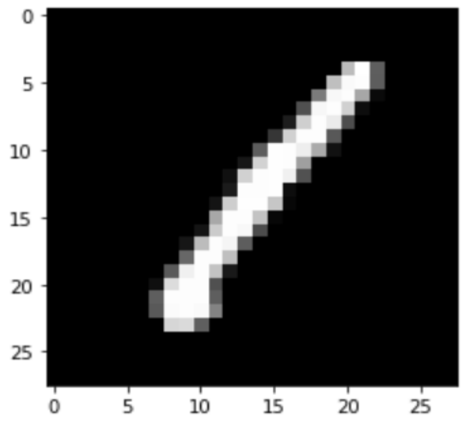
About Data

Data is taken from the Kaggle Digit Recognizer competition https://www.kaggle.com/c/digit-recognizer/data. The data files train.csv and test.csv contain gray-scale images of hand-drawn digits, from zero through nine. Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255, inclusive.

The training data set, (train.csv), has 785 columns. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel values of the associated image. The test data set, (test.csv), is the same as the training set, except that it does not contain the "label" column. Out of this complete dataset, we will be taking out a random image for our prediction.

```
# visualizing the digits for few training example
print(Y_train[0])
training_example = X_train.iloc[0]
training_example = training_example.to_numpy() # converting series to numpy array
training_example = training_example.reshape((28,28)) # converting 1d array of pixels to 2d (28X28) pixels matrix
plt.imshow(training_example, cmap='gray')
```

```
1
<matplotlib.image.AxesImage at 0x1a9007d7400>
```

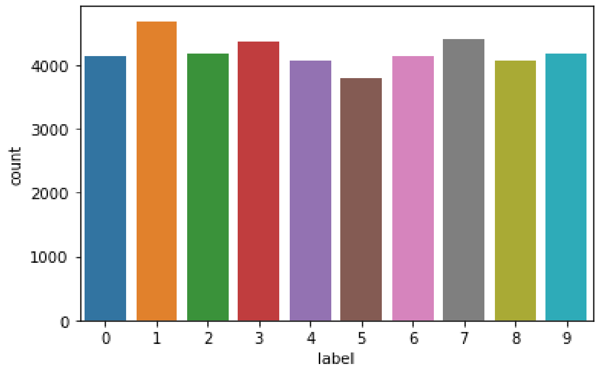


Classification Problem

The aim is to identify the digits from the images. We will be using the Naive Bayes classification algorithms (Gaussian Naive Bayes and MultiNominal Naive Bayes) to classify each image to its true digit value. In addition, we will be comparing its performance with another classification algorithm - Decision Trees.

Data Pre-Processing

In the first step, We checked for any imbalances in the dataset.



We have similar counts for the 10 digits.

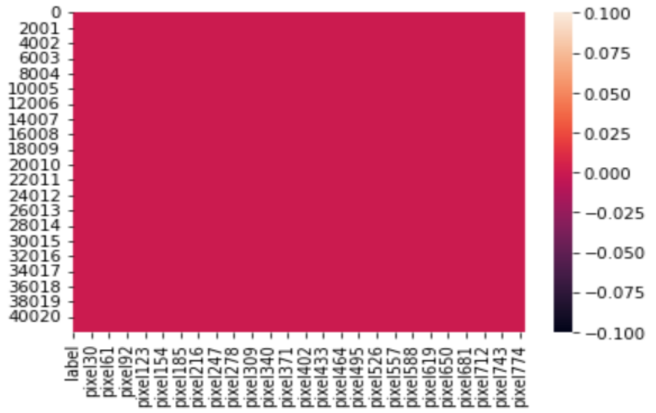
Dataset is already divided into training and test dataset. However, we would be splitting the dataset using the `train_test_split` function. We are going to divide the dataset into a 60/40 ratio.

```
# Split the train and the validation set for the fitting
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train, test_size = 0.4, random_state=30)
```

Another important step is to check for missing and null values in the dataset. Data looks very clean from the below visualizations - there are no missing values in training and testing datasets.

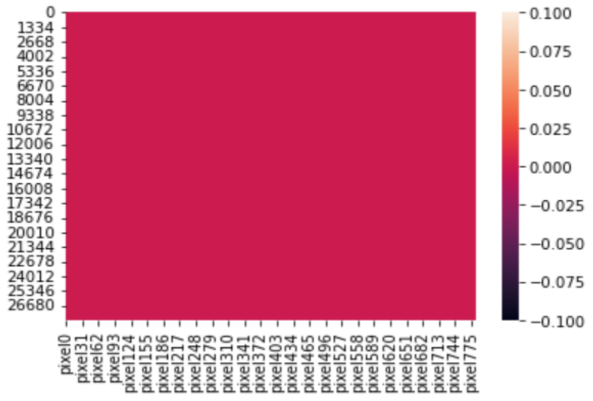
```
#sns.heatmap(train.isnull()) # no null or NA values in training dat
```

```
<AxesSubplot:>
```



```
#sns.heatmap(test.isnull()) # no null or NA values in testing data
```

```
<AxesSubplot:>
```



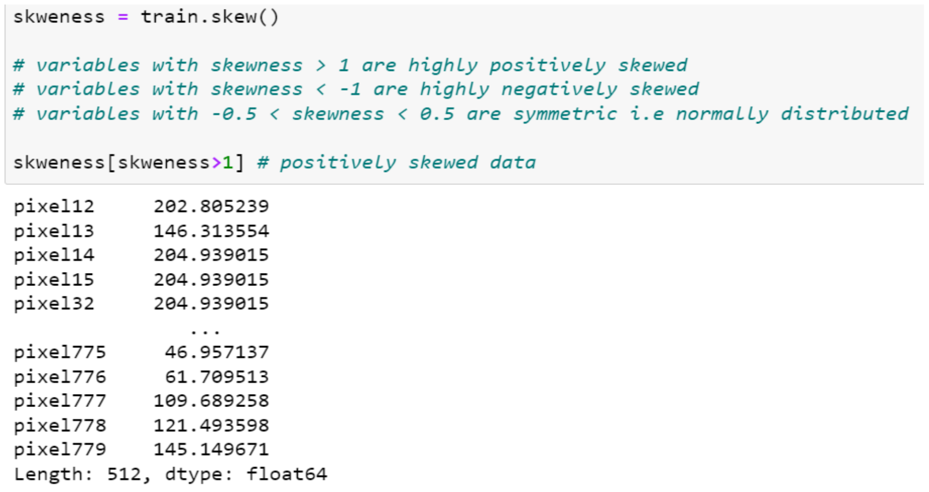
Section 2: Naive Bayes

Gaussian Naive Bayes Classifier

Gaussian Naive Bayes algorithm is a special type of NB algorithm. It's specifically used when the features have continuous values. It's also assumed that all the features are following a Gaussian distribution i.e, normal distribution.

Data Pre-Processing - Gaussian distribution assumes all the features to be normally distributed. There are various ways in which we can check the distribution of the variables - histogram plot, Skewness and Kurtosis, etc., We calculate the skewness of all the features in the training data and check whether there are any highly skewed distributions.

- variables with skewness > 1 are highly positively skewed.
- variables with skewness < -1 are highly negatively skewed.
- variables with 0.5 < skewness < 1 are moderately positively skewed.
- variables with -0.5 < skewness < -1 are moderately negatively skewed.
- And, the variables with -0.5 < skewness < 0.5 are symmetric i.e normally distributed.



```
skweness = train.skew()

# variables with skewness > 1 are highly positively skewed
# variables with skewness < -1 are highly negatively skewed
# variables with -0.5 < skewness < 0.5 are symmetric i.e normally distributed

skweness[skweness>1] # positively skewed data
```

```
pixel12     202.805239
pixel13     146.313554
pixel14     204.939015
pixel15     204.939015
pixel32     204.939015
               ...
pixel775     46.957137
pixel776     61.709513
pixel777    109.689258
pixel778    121.493598
pixel779    145.149671
Length: 512, dtype: float64
```

In the training data, we have 512 features positively skewed. Hence, the Gaussian Naive Bayes classifier might not be suitable for our data.

Model Building

This is an important part of the program where the training dataset is fed into the system and it is trained based on the target value(s) of the features. We will be first training the dataset using the Naive Bayes algorithm.

```
# instantiate the model
gnb = GaussianNB()

# fit the model
gnb.fit(X_train, Y_train)
```

```
GaussianNB()
```

```
# predict
nb_pred = gnb.predict(X_val)

nb_pred
```

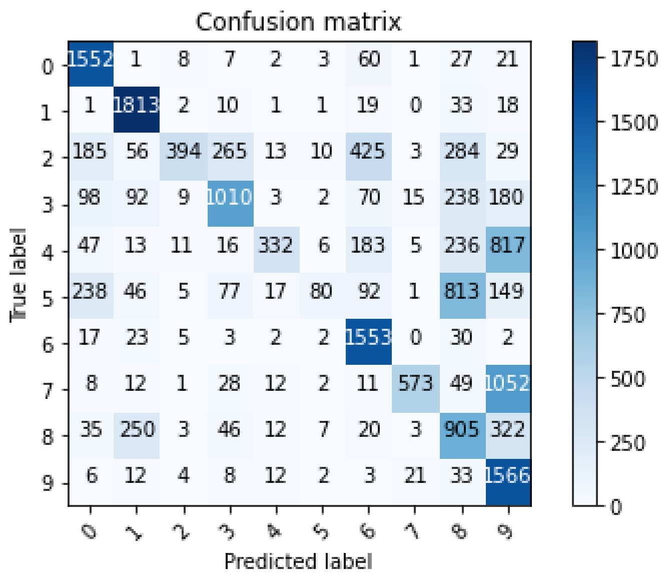
```
array([6, 1, 9, ..., 0, 2, 0], dtype=int64)
```

Model Evaluation and Conclusion

As expected, performance of GNB was very poor. The accuracy of the model is 0.58. The classifier is not able to accurately predict digits - `2,4,5,7,` and `8` which is very evident in the F1 - scores of these digits and the confusion matrix. As mentioned previously, the reason for poor performance can be due to highly skewed features present in the data. We can improve the performance of GNB by applying transformations like log transformation, inverse transformation, box-cox transformation, etc. However, we have another classifier - Multinominal Naive Bayes which might be appropriate to our dataset.

```
print(classification_report(Y_val, nb_pred))
```

```
              precision    recall  f1-score   support

           0       0.71      0.92      0.80      1682
           1       0.78      0.96      0.86      1898
           2       0.89      0.24      0.37      1664
           3       0.69      0.59      0.63      1717
           4       0.82      0.20      0.32      1666
           5       0.70      0.05      0.10      1518
           6       0.64      0.95      0.76      1637
           7       0.92      0.33      0.48      1748
           8       0.34      0.56      0.43      1603
           9       0.38      0.94      0.54      1667

    accuracy                           0.58     16800
   macro avg       0.69      0.57      0.53     16800
weighted avg       0.69      0.58      0.54     16800
```



	0	1	2	3	4	5	6	7	8	9
0	1552	1	8	7	2	3	60	1	27	21
1	1	1813	2	10	1	1	19	0	33	18
2	185	56	394	265	13	10	425	3	284	29
3	98	92	9	1010	3	2	70	15	238	180
4	47	13	11	16	332	6	183	5	236	817
5	238	46	5	77	17	80	92	1	813	149
6	17	23	5	3	2	2	1553	0	30	2
7	8	12	1	28	12	2	11	573	49	1052
8	35	250	3	46	12	7	20	3	905	322
9	6	12	4	8	12	2	3	21	33	1566

Multinomial Naive Bayes Classifier

The multinomial Naive Bayes classifier is suitable for classification with discrete features. The multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as TF-IDF may also work.

Multinomial Naive Bayes

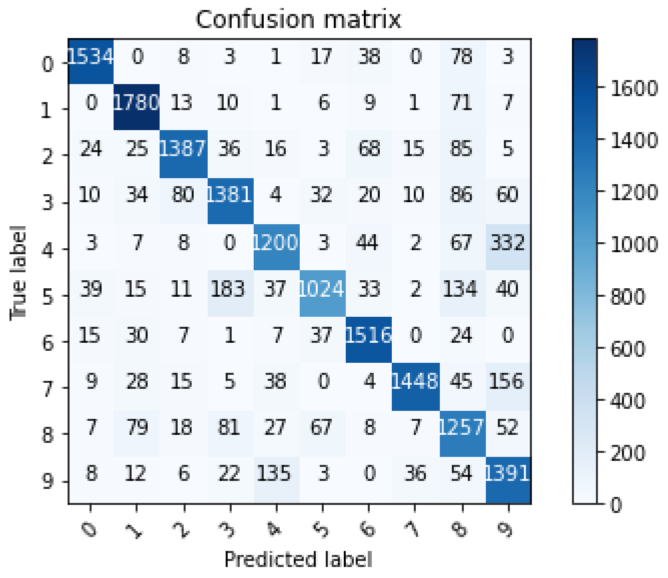
```
clf_mnb = MultinomialNB()
clf_mnb.fit(X_train, Y_train)
pred_mnb = clf_mnb.predict(X_val)
pred_mnb
```

```
array([3, 1, 7, ..., 0, 2, 0], dtype=int64)
```

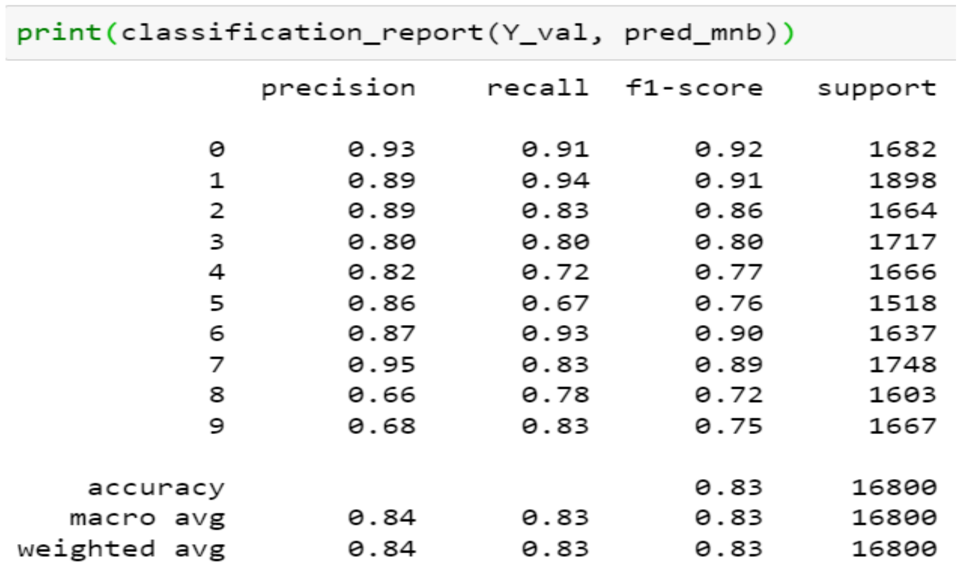
Training the model

We will be using the same dataset and training it using the Multinominal Naive Bayes classifier.

We can clearly observe that performance of the MNB algorithm is far better than GNB. The accuracy of the model is 0.83. However, the prediction of a few digits - `3,4,8,` and 9 needs to be improved. This can be clearly seen in the classification report f1-scores and the confusion matrix of these digits.



True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	1534	0	8	3	1	17	38	0	78	3
1	0	1780	13	10	1	6	9	1	71	7
2	24	25	1387	36	16	3	68	15	85	5
3	10	34	80	1381	4	32	20	10	86	60
4	3	7	8	0	1200	3	44	2	67	332
5	39	15	11	183	37	1024	33	2	134	40
6	15	30	7	1	7	37	1516	0	24	0
7	9	28	15	5	38	0	4	1448	45	156
8	7	79	18	81	27	67	8	7	1257	52
9	8	12	6	22	135	3	0	36	54	1391



```
print(classification_report(Y_val, pred_mnb))
```

```
              precision    recall  f1-score   support

           0       0.93      0.91      0.92      1682
           1       0.89      0.94      0.91      1898
           2       0.89      0.83      0.86      1664
           3       0.80      0.80      0.80      1717
           4       0.82      0.72      0.77      1666
           5       0.86      0.67      0.76      1518
           6       0.87      0.93      0.90      1637
           7       0.95      0.83      0.89      1748
           8       0.66      0.78      0.72      1603
           9       0.68      0.83      0.75      1667

    accuracy                           0.83     16800
   macro avg       0.84      0.83      0.83     16800
weighted avg       0.84      0.83      0.83     16800
```

Section 3: Decision Tree

Decision Trees

```
# training the model
clf = DecisionTreeClassifier(random_state=0, criterion='entropy')
clf.fit(X_train,Y_train)
```

```
DecisionTreeClassifier(criterion='entropy', random_state=0)
```

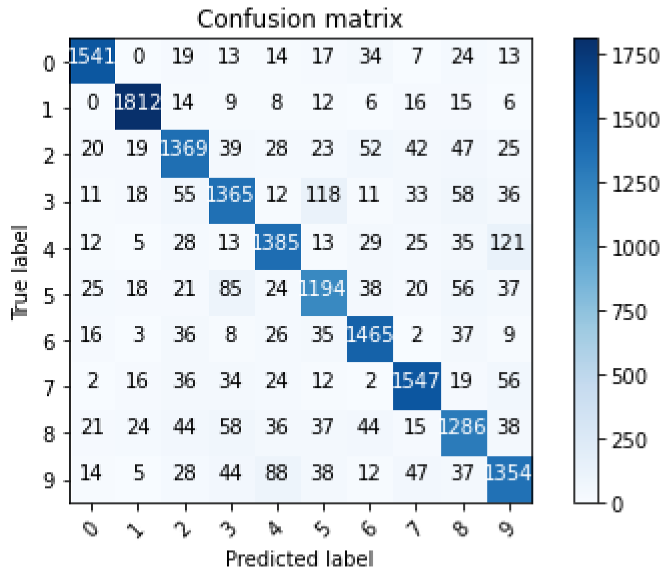
```
# predicting on test data
dt_pred=clf.predict(X_val)
dt_pred
```

```
array([5, 1, 7, ..., 0, 2, 0], dtype=int64)
```

Now, we will be using the same dataset and training it using the Decision Tree algorithm. The decision tree model predicts the value of a target variable by learning simple decision rules inferred from the data features.

Conclusion

We have used the entropy criterion in this model to calculate the information gain. The accuracy of the model is 0.85 which is better when compared to the Multinominal Naive Bayes classifier. Prediction of the digit - 5 is still low compared to other digits.

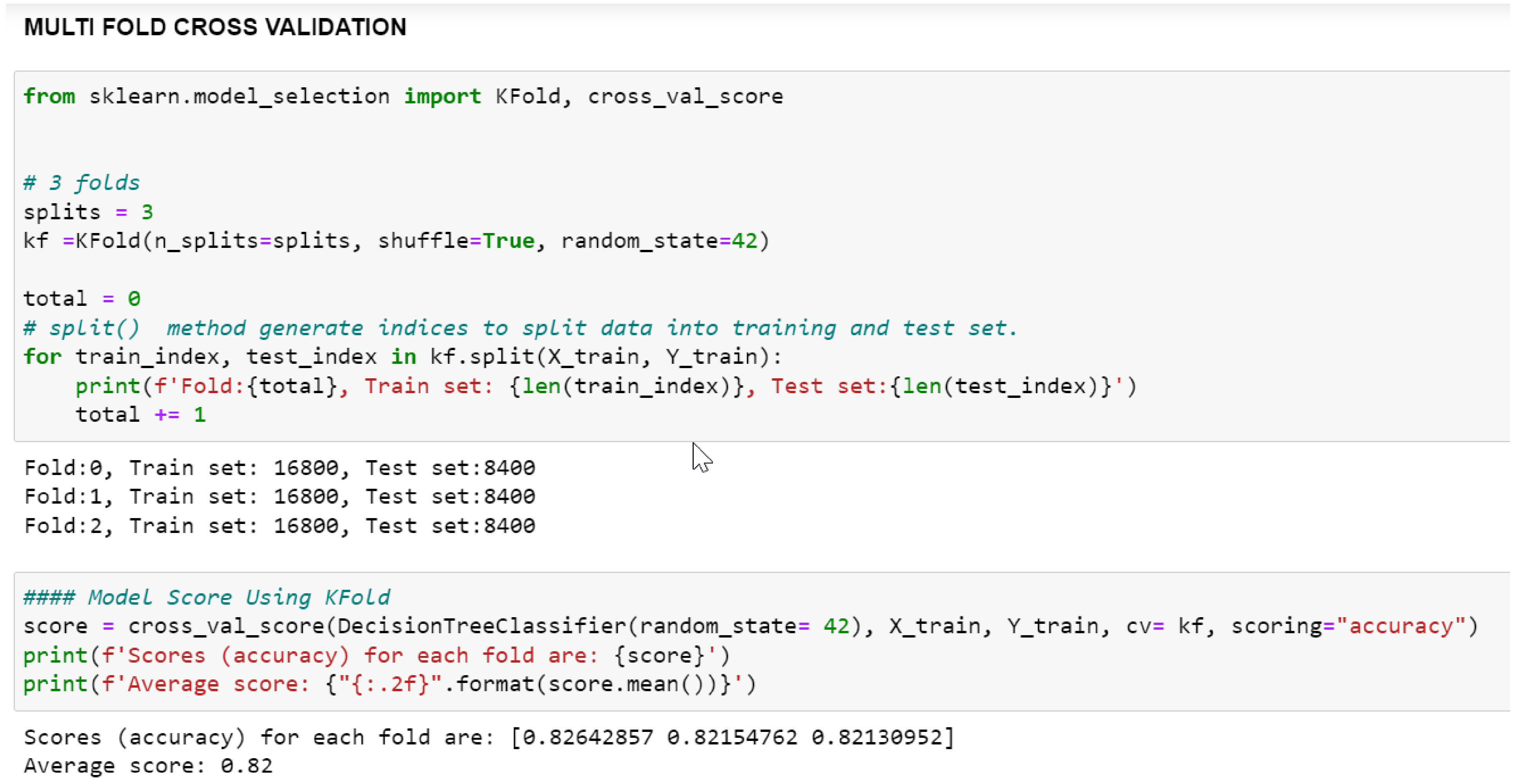


```
print(classification_report(Y_val, dt_pred))
```

```
              precision    recall  f1-score   support

           0       0.93      0.92      0.92      1682
           1       0.94      0.95      0.95      1898
           2       0.83      0.82      0.83      1664
           3       0.82      0.79      0.81      1717
           4       0.84      0.83      0.84      1666
           5       0.80      0.79      0.79      1518
           6       0.87      0.89      0.88      1637
           7       0.88      0.89      0.88      1748
           8       0.80      0.80      0.80      1603
           9       0.80      0.81      0.81      1667

    accuracy                           0.85     16800
   macro avg       0.85      0.85      0.85     16800
weighted avg       0.85      0.85      0.85     16800
```

3 Fold Cross-Validation of Decision Tree Classifier

Till now, we have divided the input data into train and test datasets and used them for model training and testing respectively. This method is not very reliable as train and test data do not always have the same variation as original data, which will affect the accuracy of the model.
Cross-validation solves this problem by dividing the input data into multiple groups instead of just two groups.



MULTI FOLD CROSS VALIDATION

```python
from sklearn.model_selection import KFold, cross_val_score


# 3 folds
splits = 3
kf =KFold(n_splits=splits, shuffle=True, random_state=42)

total = 0
# split()  method generate indices to split data into training and test set.
for train_index, test_index in kf.split(X_train, Y_train):
    print(f'Fold:{total}, Train set: {len(train_index)}, Test set:{len(test_index)}')
    total += 1
```

```
Fold:0, Train set: 16800, Test set:8400
Fold:1, Train set: 16800, Test set:8400
Fold:2, Train set: 16800, Test set:8400
```

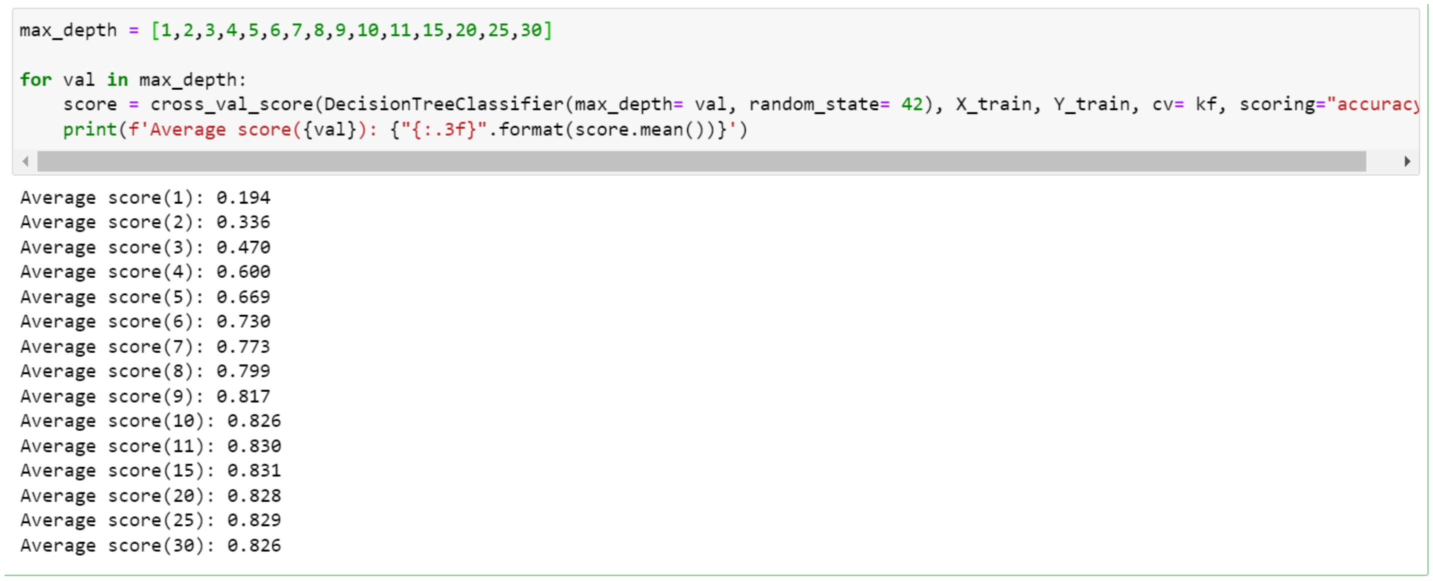
```python
#### Model Score Using KFold
score = cross_val_score(DecisionTreeClassifier(random_state= 42), X_train, Y_train, cv= kf, scoring="accuracy")
print(f'Scores (accuracy) for each fold are: {score}')
print(f'Average score: {"{:.2f}".format(score.mean())}')
```

```
Scores (accuracy) for each fold are: [0.82642857 0.82154762 0.82130952]
Average score: 0.82
```

The average accuracy of 3 fold cross-validation of the decision tree classifier is 0.82.

Hyperparameter Tuning of max_depth of the tree

Here we are going to do tuning based on ‘max_depth’. We will try with max depth starting from 1 to 10 and depending on the final ‘accuracy’ score, we can choose the optimal max_depth for the model.



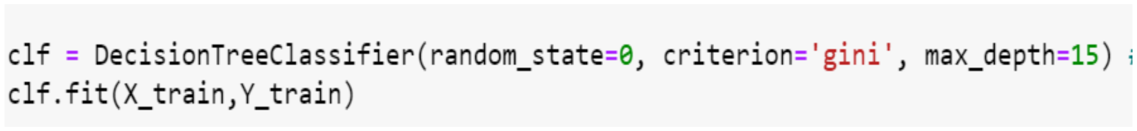
```
max_depth = [1,2,3,4,5,6,7,8,9,10,11,15,20,25,30]

for val in max_depth:
    score = cross_val_score(DecisionTreeClassifier(max_depth= val, random_state= 42), X_train, Y_train, cv= kf, scoring="accuracy
    print(f'Average score({val}): {"{:.3f}".format(score.mean())}')
```

```
Average score(1): 0.194
Average score(2): 0.336
Average score(3): 0.470
Average score(4): 0.600
Average score(5): 0.669
Average score(6): 0.730
Average score(7): 0.773
Average score(8): 0.799
Average score(9): 0.817
Average score(10): 0.826
Average score(11): 0.830
Average score(15): 0.831
Average score(20): 0.828
Average score(25): 0.829
Average score(30): 0.826
```

With `max_depth` = 15, the model has the highest accuracy of 0.83. We are going to use this value for training our final model.

max_depth = 15 seems to be optimal from the above results

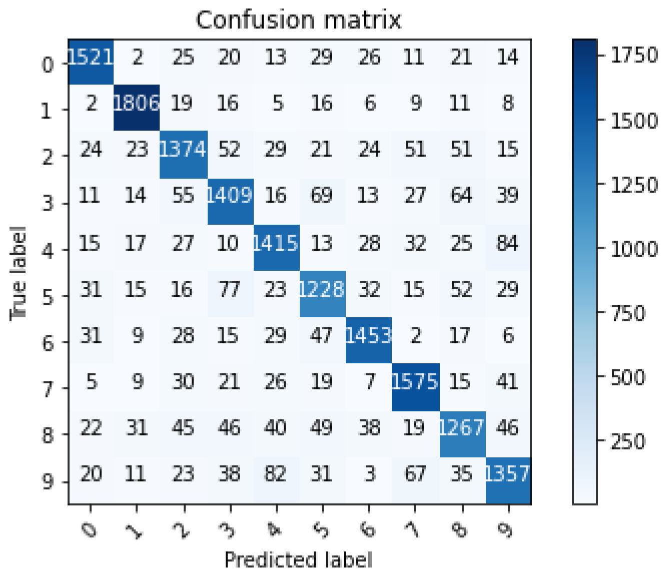
Decision Tree Classifier (max_depth 15 and Gini Impurity)



```
clf = DecisionTreeClassifier(random_state=0, criterion='gini', max_depth=15)
clf.fit(X_train,Y_train)
```

Training the model with the same dataset but with appropriate parameters obtained from hyperparameter tuning. We have pruned the tree to have a max depth of 15 and in this model, we tried to use Gini impurity to measure the purity of the split instead of Entropy. The accuracy of this model is 0.86. The accuracy of the model improved by 0.1 because of hyper-parameter tuning.

Confusion matrix and classification report of Decision Model after hyperparameter tuning and pruning the tree:



	0	1	2	3	4	5	6	7	8	9
0	1521	2	25	20	13	29	26	11	21	14
1	2	1806	19	16	5	16	6	9	11	8
2	24	23	1374	52	29	21	24	51	51	15
3	11	14	55	1409	16	69	13	27	64	39
4	15	17	27	10	1415	13	28	32	25	84
5	31	15	16	77	23	1228	32	15	52	29
6	31	9	28	15	29	47	1453	2	17	6
7	5	9	30	21	26	19	7	1575	15	41
8	22	31	45	46	40	49	38	19	1267	46
9	20	11	23	38	82	31	3	67	35	1357

```
              precision    recall  f1-score   support

           0       0.90      0.90      0.90      1682
           1       0.93      0.95      0.94      1898
           2       0.84      0.83      0.83      1664
           3       0.83      0.82      0.82      1717
           4       0.84      0.85      0.85      1666
           5       0.81      0.81      0.81      1518
           6       0.89      0.89      0.89      1637
           7       0.87      0.90      0.89      1748
           8       0.81      0.79      0.80      1603
           9       0.83      0.81      0.82      1667

    accuracy                           0.86     16800
   macro avg       0.86      0.86      0.86     16800
weighted avg       0.86      0.86      0.86     16800
```

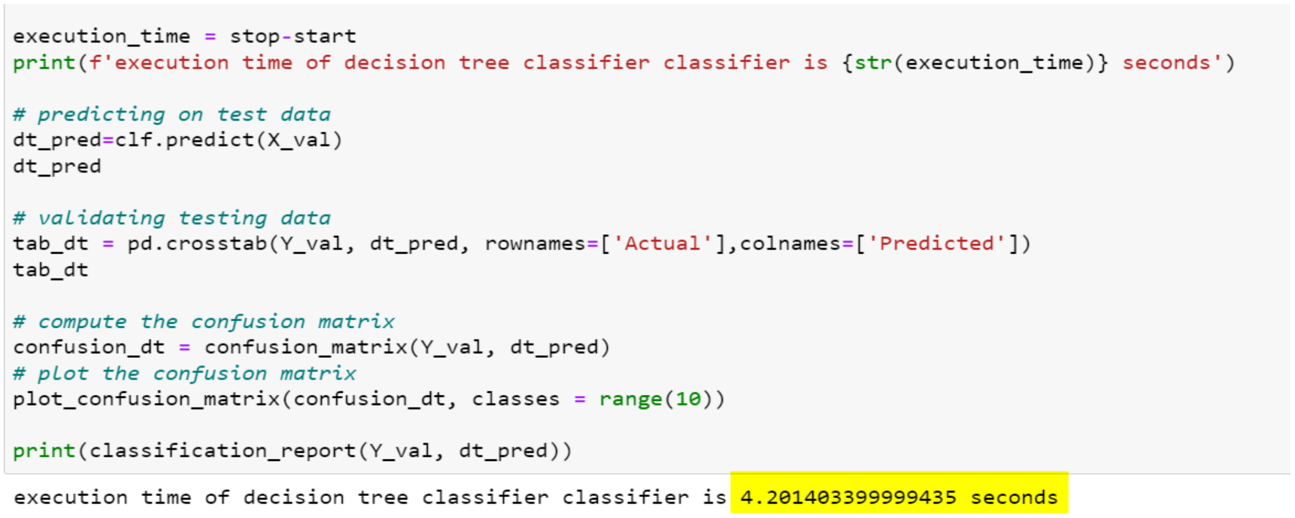
Section 4: Algorithm performance comparison

Comparison between Gaussian Naive Bayes and Multinominal Naive Bayes

Among the two Naive Bayes algorithms - gaussian and multinominal, the Multinominal Naive Bayes classifier showed significant improvement. Gaussian Naive Bayes algorithm accuracy is 0.53 whereas Multinominal Naive Bayes algorithm accuracy is 0.83. GNB assumes all the features are normally distributed but pixels in our dataset is having skewed data. This can be the reason for poor performance by the GNB algorithm. On the other hand, MNB is suitable for discrete data. Pixels are discrete data hence the accuracy of the MNB algorithm is very high compared to GNB.

Comparison between Multinominal Naive Bayes and Decision Tree (after hyperparameter tuning)

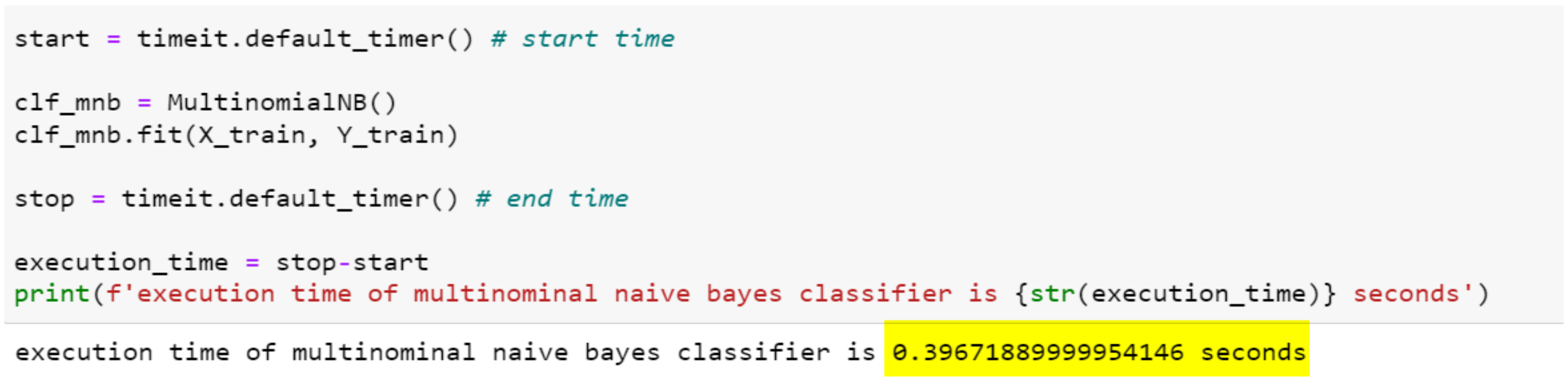
The accuracy of the decision tree is **0.86** after hyperparameter tuning and applying the pruning technique (max_depth = 15 and criterion=gini). However, the execution time of the decision tree is **4.2 seconds** which seems pretty high for a training dataset of ~20,000.



```
execution_time = stop-start
print(f'execution time of decision tree classifier classifier is {str(execution_time)} seconds')

# predicting on test data
dt_pred=clf.predict(X_val)
dt_pred

# validating testing data
tab_dt = pd.crosstab(Y_val, dt_pred, rownames=['Actual'],colnames=['Predicted'])
tab_dt

# compute the confusion matrix
confusion_dt = confusion_matrix(Y_val, dt_pred)
# plot the confusion matrix
plot_confusion_matrix(confusion_dt, classes = range(10))

print(classification_report(Y_val, dt_pred))
```

```
execution time of decision tree classifier classifier is 4.201403399999435 seconds
```

The accuracy of the Multinominal Naive Bayes classifier is **0.83.** However, the execution time of this classifier was only **0.39 seconds**.



```
start = timeit.default_timer() # start time

clf_mnb = MultinomialNB()
clf_mnb.fit(X_train, Y_train)

stop = timeit.default_timer() # end time

execution_time = stop-start
print(f'execution time of multinominal naive bayes classifier is {str(execution_time)} seconds')
```

```
execution time of multinominal naive bayes classifier is 0.39671889999954146 seconds
```

The decision tree often involves higher time to train the model because even a small change in the data can result in a large change in the structure of the decision tree.

Section 5: Kaggle test result

Test accuracy of Multinominal Naive Bayes - 0.83
Test accuracy of Decision Tree (with hyperparameter tuning and pruning) - 0.86

Kaggle Submission
https://www.kaggle.com/chaithrakc/competitions?tab=active
https://www.kaggle.com/code/chaithrakc/notebook2f176f476c