Machine Learning Assignment-2

Team Members:

1. Ritvik 2020A7PS1723H
2. Dhruv Merchant 2020A7PS2063H
3. Ishaan Srivastava 2020A7PS2071H

Part A - Naive Bayes Classifier to predict income

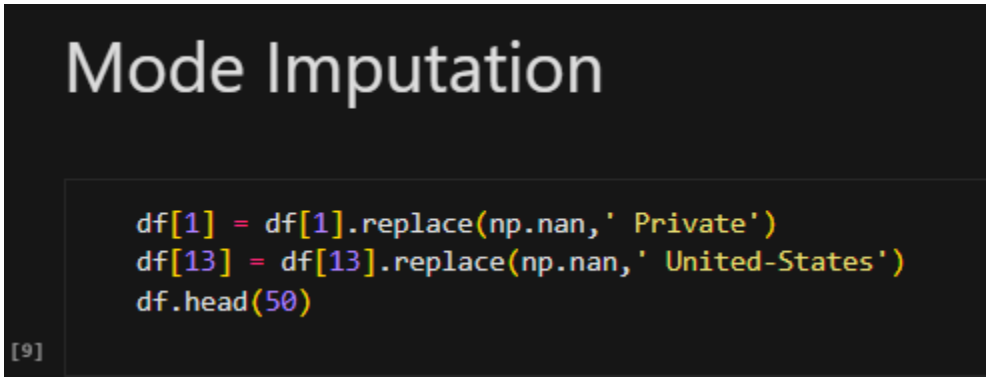
Task 1: Data Preprocessing

After loading the dataset, the first thing we did was find the missing values. Since missing values were represented as' ?' we converted them to numpy NaN values to make the work easier.

```
df=df.replace(to_replace=' ?',value=np.nan)
df.head(50)
```

```
[7]  df.isnull().sum()

...  0        0
     1     1836
     2        0
     3        0
     4        0
     5        0
     6     1843
     7        0
     8        0
     9        0
     10       0
     11       0
     12       0
     13     583
     14       0
     dtype: int64
```

Upon observation we saw that for column 1 and 13, one feature value appeared majority of the time so we implemented mode imputation on these columns and for column 6 we used proportional imputation where we provided probabilities to some feature values which were imputed according to the probability.




```
# Define the frequencies
frequencies = {
    ' Prof-specialty': 4140,
    ' Craft-repair': 4099,
    ' Exec-managerial': 4066,
    ' Adm-clerical': 3770,
    ' Sales': 3650,
    ' Other-service': 3295,
    ' Machine-op-inspct': 2002,
    ' Transport-moving': 1597,
    ' Handlers-cleaners': 1370,
    ' Farming-fishing': 994,
    ' Tech-support': 928,
    ' Protective-serv': 649,
    ' Priv-house-serv': 149,
    ' Armed-Forces': 9
}
```

```
# Calculate the total count
total_count = sum(frequencies.values())

# Calculate the proportional frequencies
proportional_frequencies = {k: v / total_count for k, v in frequencies.items()}

# Replace the missing values in column 6 with the proportional frequencies
df[6] = df[6].apply(lambda x: np.random.choice(list(proportional_frequencies.keys()), p=list(proportional_frequencies.values())) if pd.isna(x) else x)
```

```
df.head(50)
```

In the end, we imputed all the missing values.

```
[14]  df.isnull().sum()

...   0     0
      1     0
      2     0
      3     0
      4     0
      5     0
      6     0
      7     0
      8     0
      9     0
      10    0
      11    0
      12    0
      13    0
      14    0
      dtype: int64
```

Then to make the prediction smoother and more accurate we applied normalization and standardization on the columns which had discrete values.

Normalization: The goal of normalization is to transform features to be on a similar scale. This improves the performance and training stability of the model.

```
# Select columns to normalize
cols_to_normalize = [0, 2, 4, 10, 11, 12]

print(df.iloc[:, cols_to_normalize].values)
print("----------------------------------------")

# Normalize selected columns
df.iloc[:, cols_to_normalize] = (df.iloc[:, cols_to_normalize] - df.iloc[:, cols_to_normalize].mean(axis=0)) / df.iloc[:, cols_to_normalize].std(axis=0)

print(df.iloc[:, cols_to_normalize].values)
```

```
[15]
...  [[    39  77516     13   2174      0     40]
      [    50  83311     13      0      0     13]
      [    38 215646      9      0      0     40]
      ...
      [    58 151910      9      0      0     40]
      [    22 201490      9      0      0     20]
      [    52 287927      9  15024      0     40]]
     ----------------------------------------
     [[ 0.03067009 -1.06359441  1.13472134  0.14845062 -0.2166562  -0.0354289 ]
      [ 0.83709613 -1.00869151  1.13472134 -0.14591824 -0.2166562  -2.222119  ]
      [-0.04264137  0.24507474 -0.42005317 -0.14591824 -0.2166562  -0.0354289 ]
      ...
      [ 1.42358779 -0.3587719  -0.42005317 -0.14591824 -0.2166562  -0.0354289 ]
      [-1.2156247   0.11095818 -0.42005317 -0.14591824 -0.2166562  -1.65519934]
      [ 0.98371904  0.9298783  -0.42005317  1.88839534 -0.2166562  -0.0354289 ]]
```

Standardization: Standardization is another scaling method where the values are centered around the mean with a unit standard deviation. This means that the mean of the attribute becomes zero, and the resultant distribution has a unit standard deviation.

$$X' = \frac{X - \mu}{\sigma}$$

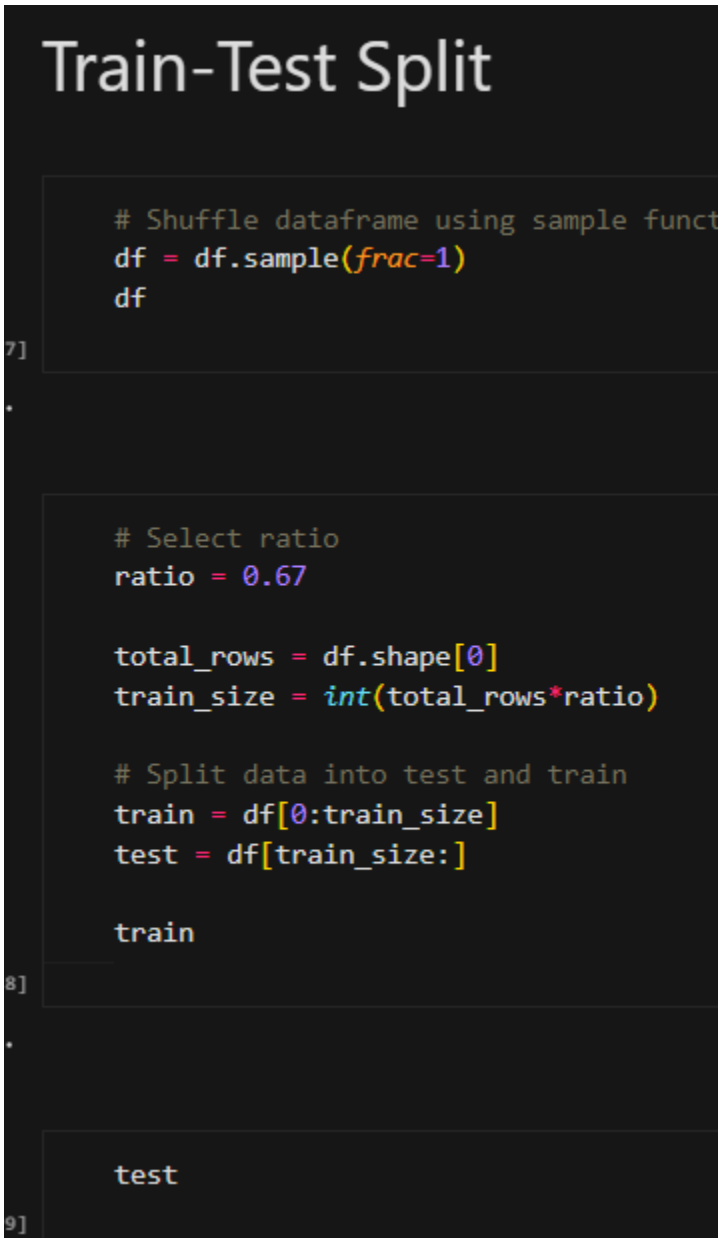
Standardize the Data

```
print(df.iloc[:, cols_to_normalize].values)
print("----------------------------------------")

selected_data = df.iloc[:, cols_to_normalize]

# standardize selected columns using NumPy
selected_data = (selected_data - np.mean(selected_data, axis=0)) / np.std(selected_data, axis=0, ddof=1)

# replace selected columns with standardized values
df.iloc[:, cols_to_normalize] = selected_data


print(df.iloc[:, cols_to_normalize].values)
```

[16]

```
[[ 0.03067009 -1.06359441  1.13472134  0.14845062 -0.2166562  -0.0354289 ]
 [ 0.83709613 -1.00869151  1.13472134 -0.14591824 -0.2166562  -2.222119  ]
 [-0.04264137  0.24507474 -0.42005317 -0.14591824 -0.2166562  -0.0354289 ]
 ...
 [ 1.42358779 -0.3587719  -0.42005317 -0.14591824 -0.2166562  -0.0354289 ]
 [-1.2156247   0.11095818 -0.42005317 -0.14591824 -0.2166562  -1.65519934]
 [ 0.98371904  0.9298783  -0.42005317  1.88839534 -0.2166562  -0.0354289 ]]
----------------------------------------
[[ 0.03067009 -1.06359441  1.13472134  0.14845062 -0.2166562  -0.0354289 ]
 [ 0.83709613 -1.00869151  1.13472134 -0.14591824 -0.2166562  -2.222119  ]
 [-0.04264137  0.24507474 -0.42005317 -0.14591824 -0.2166562  -0.0354289 ]
 ...
 [ 1.42358779 -0.3587719  -0.42005317 -0.14591824 -0.2166562  -0.0354289 ]
 [-1.2156247   0.11095818 -0.42005317 -0.14591824 -0.2166562  -1.65519934]
 [ 0.98371904  0.9298783  -0.42005317  1.88839534 -0.2166562  -0.0354289 ]]
```

Finally we performed the train-test split where 67% was train and 33% was test.



Task 2: Naive Bayes Classifier Implementation

Implement a function to calculate the prior probability of each class (benign and malignant) in the training set

This code is dividing a pandas DataFrame named "train" into two separate DataFrames based on the values in column 14, which seems to be a binary class label indicating whether an individual's income is above or below $50k.
The first line creates a new DataFrame named "df_less_than_equal_50k" by selecting only the rows in the original DataFrame where the value in column 14 is " <=50K". Similarly, the second line creates a new DataFrame named "df_more_than_50k" by selecting only the rows where the value in column 14 is " >50K".
The next three lines calculate the total number of rows in the original DataFrame (count_train) and the number of rows in each of the two new DataFrames (num_less_than_equal_50k and num_more_than_50k), which will be used to calculate the prior probabilities of each class.
The following two lines calculate the prior probabilities of the two classes by dividing the number of rows in each new DataFrame by the total number of rows in the original DataFrame. These prior probabilities are stored in a dictionary named "priorProbDict" with the class labels as keys and the probabilities as values.
Finally, the code prints out the prior probabilities for each class and the entire "priorProbDict" dictionary.

Implement a function to calculate the prior probability of each class

```
#dividing into 2 data frames for each class label >50k and <=50k
df_less_than_equal_50k = train[train[14] == " <=50K"]
df_more_than_50k = train[train[14] == " >50K"]

#length of train set
count_train = len(train)

#taking count of dataframe <=50k
num_less_than_equal_50k = len(df_less_than_equal_50k)

#taking count of dataframe >50k
num_more_than_50k = len(df_more_than_50k)

#Calculation prior probability
prob_more_50k = float(num_more_than_50k)/float(count_train)
prob_less_equal_50k = float(num_less_than_equal_50k)/float(count_train)
#Storing it in a dictionary
priorProbDict = {" >50K": prob_more_50k, " <=50K": prob_less_equal_50k }
print(prob_more_50k)
print(prob_less_equal_50k)
print("------------------------------")
print(priorProbDict)
```

```
0.24134769653907862
0.7586523034609214
------------------------------
{' >50K': 0.24134769653907862, ' <=50K': 0.7586523034609214}
```

Implement a function to calculate the conditional probability of each feature given to each class in the training set.

The code first defines a list of column indices corresponding to continuous features (columns 0, 2, 4, 10, 11, and 12). These columns will be handled differently from the categorical features, which will be handled in a separate loop.
The loop for categorical features starts by iterating over each feature (column) in the DataFrame, skipping the continuous columns. For each feature, it obtains the unique values in that column and then iterates over each class value (" <=50K" and " >50K"). For each class value, it subsets the DataFrame to only include rows with that class value and calculates the count of each unique feature value in that subset. These counts are used to calculate the conditional probabilities of each feature value given that class, and the resulting dictionary of probabilities is stored in the "likelihood" dictionary under the appropriate class and feature.
The loop for continuous features iterates over the same features as before, but instead of calculating the conditional probabilities directly from the counts, it uses a Gaussian Naive Bayes approach to estimate the probability density function of each feature given each class. For each feature value and each class value, it calculates the probability density of that feature value under a Gaussian distribution with mean and standard deviation estimated from the subset of the DataFrame with that class value. The resulting dictionary of probabilities is again stored in the "likelihood" dictionary under the appropriate class and feature.

```
feature_probs = {}
likelihood = {" <=50K": {}, " >50K": {}}

continuous_columns = [0, 2, 4, 10, 11, 12]

#Calculating conditional probability for categorical columns
for feature in train.columns[:-1]:
    if feature not in continuous_columns:
        feature_values = train[feature].unique()
        for class_value in class_values:
            class_data = train[train[14] == class_value]
            class_count = len(class_data)
            feature_probs = {}
            for feature_value in feature_values:
                feature_count = len(class_data[class_data[feature] == feature_value])
                feature_probs[feature_value] = feature_count / class_count
            likelihood[class_value][feature] = feature_probs

#Calculating conditional probability for continuous columns using gaussian naive bayes classifier
for feature_idx in continuous_columns:
    feature = train.columns[feature_idx]
    feature_values = train[feature].unique()
    for class_value in class_values:
        class_data = train[train[14] == class_value]
        class_count = len(class_data)
        feature_probs = {}
        for feature_value in feature_values:
            if feature_idx == 0:
                # handle the case where feature is the target class column
                feature_count = len(class_data[class_data[feature] == feature_value])
            else:
                # use Gaussian Naive Bayes to calculate feature probability
                class_feature_data = class_data.iloc[:, [feature_idx, -1]]
                class_feature_values = class_feature_data.iloc[:, 0].values
                mean = np.mean(class_feature_values)
                std_dev = np.std(class_feature_values)
                prob = (1 / (np.sqrt(2 * np.pi) * std_dev)) * np.exp(-(feature_value - mean)**2 / (2 * std_dev**2))
                feature_probs[feature_value] = prob
            likelihood[class_value][feature] = feature_probs
```

Implement a function to predict the class of a given instance using the Naive Bayes algorithm.

```
predicted_labels = []
for index, row in test.iterrows():
    post_prob_greater = 1
    post_prob_less = 1
    for col in test.columns:
        if col != 14:
            val = row[col]
            post_prob_greater *= likelihood[' >50K'][col].get(val, 1e-10)
            post_prob_less *= likelihood[' <=50K'][col].get(val, 1e-10)
    post_prob_greater *= priorProbDict[' >50K']
    post_prob_less *= priorProbDict[' <=50K']
    if post_prob_greater >= post_prob_less:
        predicted_labels.append(' >50K')
    else:
        predicted_labels.append(' <=50K')

print(predicted_labels)
```

The loop iterates over each row (data point) in the "test" DataFrame, and for each row it calculates the posterior probability of the data belonging to each class (" >50K" and " <=50K"). The posterior probabilities are calculated by multiplying the conditional probabilities of each feature value given each class (retrieved from the "likelihood" dictionary) and the prior probability of each class (retrieved from the "priorProbDict" dictionary).
For each feature value, if it has not been seen in the training data (i.e., its probability in the "likelihood" dictionary is 0), a small value of 1e-10 is used instead to avoid a probability of 0 in the posterior probability calculation.
Finally, the predicted class label for each data point is determined by comparing the two posterior probabilities and selecting the class with the higher probability. If the posterior probability of " >50K" is greater than or equal to the posterior probability of " <=50K", the predicted class is " >50K", and vice versa.

Implement a function to calculate the accuracy of your Naive Bayes classifier on the testing set.

```
true_labels = test[14].values
num_correct = np.sum(true_labels == predicted_labels)
accuracy = num_correct / len(true_labels)
print(accuracy)
```

The accuracy we got was around `0.8309138284012656`

Task 3: Evaluation and Improvement

Evaluate the performance of your Naive Bayes classifier using accuracy, precision, recall, and F1-score.

performance_check function takes in two arguments: true_labels and predicted_labels, which are lists of true class labels and predicted class labels, respectively. It calculates a confusion matrix that shows the number of true positives, false positives, true negatives, and false negatives. Then it calculates precision, recall, and F1-score for each class label, and returns them as arrays. The precision is the ratio of true positives to the total number of positive predictions. The recall is the ratio of true positives to the total number of actual positives. The F1-score is the harmonic mean of precision and recall, and is a measure of the overall performance of the classifier. Finally, the function prints the confusion matrix, precision, recall, and F1-score.

These are the following figures we got for Naive Baye's Classifier.

```
<class 'numpy.ndarray'>
[[7621  536]
 [1281 1308]]

Precision, Recall, and F1 Score:
 Class    | Precision | Recall | F1 Score|
 <=50k    |   0.86    |  0.93  |  0.89   |
  >50k    |   0.71    |  0.51  |  0.59   |
```

2. Experiment with different smoothing techniques to improve the performance of your classifier. You need to study and understand the different smoothing techniques on your own.

We implemented the Laplace smoothening method.

Laplace smoothing, also known as add-one smoothing, is a technique used in probabilistic modeling and machine learning to smooth categorical data. It is used when calculating conditional probabilities in Bayesian inference, especially in naive Bayes classifiers.
In simple terms, Laplace smoothing adds a small positive constant to the numerator and a constant multiple of that constant to the denominator when calculating the probabilities, which helps avoid zero probability estimates for features that are not present in the training set. The added constant is usually 1, hence the name add-one smoothing.

We have defined a function called laplace_smoothing which performs Laplace smoothing on the likelihood probabilities obtained from the likelihoodfn function. The Laplace smoothing formula is applied for each alpha value in the list. We start by dividing the training set into two dataframes - one for the records where the target class is "<=50K" and the other for records where the target class is ">50K". We calculate the total count of records in the training set and the count of records for each class. We then calculate the prior probabilities for each class using the counts. The likelihoodfn function is called to obtain the initial likelihood probabilities.
For each class, feature and feature value combination, we apply the Laplace smoothing formula, which involves adding 1 to the count of occurrences of the feature value in the corresponding class, and dividing by the total count of unique feature values in the training set for that feature plus 1. The result is stored back in the likelihood dictionary.
Finally, we call the testing function to predict the class labels for the test set using the updated likelihood probabilities and prior probabilities. We then calculate the accuracy of the predictions using the calculate_accuracy function and print it to the console along with the value of alpha used for Laplace smoothing

Why the accuracy is reduced?
In Naive Bayes, the assumption is made that all features are conditionally independent given the class variable. However, in reality, there may be correlations among features which violate this assumption. As a result, the model may not be able to capture these relationships and may lead to reduced accuracy on large datasets.

```
Accuracy with alpha = 1: 0.7708914945095849
```

```
def laplace_smoothing(train,test):
    df_less_than_equal_50k = train[train[14] == " <=50K"]
    df_more_than_50k = train[train[14] == " >50K"]
    count_train = len(train)
    num_less_than_equal_50k = len(df_less_than_equal_50k)
    num_more_than_50k = len(df_more_than_50k)
    prob_more_50k = float(num_more_than_50k)/float(count_train)
    prob_less_equal_50k = float(num_less_than_equal_50k)/float(count_train)
    priorProbDict = {" >50K": prob_more_50k, " <=50K": prob_less_equal_50k }
    likelihood = likelihoodfn(train,test)
    # perform Laplace smoothing for each alpha value in the list
    for class_value in likelihood:
        for feature in likelihood[class_value]:
            for feature_value in likelihood[class_value][feature]:
                    # apply Laplace smoothing formula
                likelihood[class_value][feature][feature_value] = (likelihood[class_value][feature][feature_value] + 1) / \
                    ((df_less_than_equal_50k[feature].nunique() + df_more_than_50k[feature].nunique() + 1 ))
    predicted_labels = testing(train,test,priorProbDict,likelihood)
    accuracy = calculate_accuracy(test, predicted_labels)
    print(f"Accuracy with alpha = {1}: {accuracy}")
    return likelihood
```

```
likelihood = laplace_smoothing(train, test)
```

```
Accuracy with alpha = 1: 0.7708914945095849
```

3. Compare the performance of your Naive Bayes classifier with other classification algorithms like logistic regression and k-nearest neighbors.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_recall_fscore_support
from sklearn.preprocessing import LabelEncoder

categorical_cols = [1, 3, 5, 6, 7, 8, 9, 13]
continuous_cols = [0, 2, 4, 10, 11, 12]

# label encoding
le = LabelEncoder()
for col in categorical_cols:
    train.iloc[:, col] = le.fit_transform(train.iloc[:, col])
    test.iloc[:, col] = le.transform(test.iloc[:, col])
```

Label encoding is a preprocessing step used to transform categorical features into numerical values, which can be used in machine learning models. The process involves assigning each unique category in a categorical feature with a unique integer label.
In this code snippet, the LabelEncoder class from scikit-learn is used to label encode the categorical features in both the training and test datasets. The fit_transform() method of the LabelEncoder class is used to transform the values of each categorical column in the training dataset into numerical values, and the transform() method is used to transform the values of each categorical column in the test dataset using the same labels learned from the training dataset.
After label encoding, the categorical features are represented as integers, and can be used as input to machine learning models.

Making Logstic Regression

In the code above, the LogisticRegression class from the sklearn.linear_model module is used to build a logistic regression model. The max_iter parameter is set to a large value to ensure that the model converges. The fit method is used to train the model on the training data, and the predict method is used to make predictions on the test data.
The performance of the model is evaluated using various metrics such as accuracy, confusion matrix, precision, recall, and F1-score. The accuracy_score, confusion_matrix, precision_recall_fscore_support functions from the sklearn.metrics module are used to calculate these metrics. The average=None parameter is used to calculate these metrics for each class separately.

Making Logistic Regression using sklearn

```
# separating target and predictors
X_train = train.iloc[:, :-1].values
X_test = test.iloc[:, :-1].values
Y_train = train.iloc[:, -1].values
Y_test = test.iloc[:, -1].values

# building the model
model = LogisticRegression(max_iter=100000)
model.fit(X_train, Y_train)

# making predictions on the test set
Y_pred = model.predict(X_test)

# evaluating performance
acc_log = accuracy_score(Y_test, Y_pred)
cm_log = confusion_matrix(Y_test, Y_pred)
precision_log, recall_log, f1_log, support_log = precision_recall_fscore_support(Y_test, Y_pred, average=None)

print('Accuracy:', acc_log)
print('Confusion Matrix:\n', cm_log)
print('Precision:', precision_log)
print('Recall:', recall_log)
print('F1-score:', f1_log)
```

Making KNN

In the code above, the LogisticRegression class from the sklearn.linear_model module is used to build a logistic regression model. The max_iter parameter is set to a large value to ensure that the model converges. The fit method is used to train the model on the training data, and the predict method is used to make predictions on the test data.
The performance of the model is evaluated using various metrics such as accuracy, confusion matrix, precision, recall, and F1-score. The accuracy_score, confusion_matrix,

precision_recall_fscore_support functions from the sklearn.metrics module are used to calculate these metrics. The average=None parameter is used to calculate these metrics for each class separately.

Making KNN model from sklearn

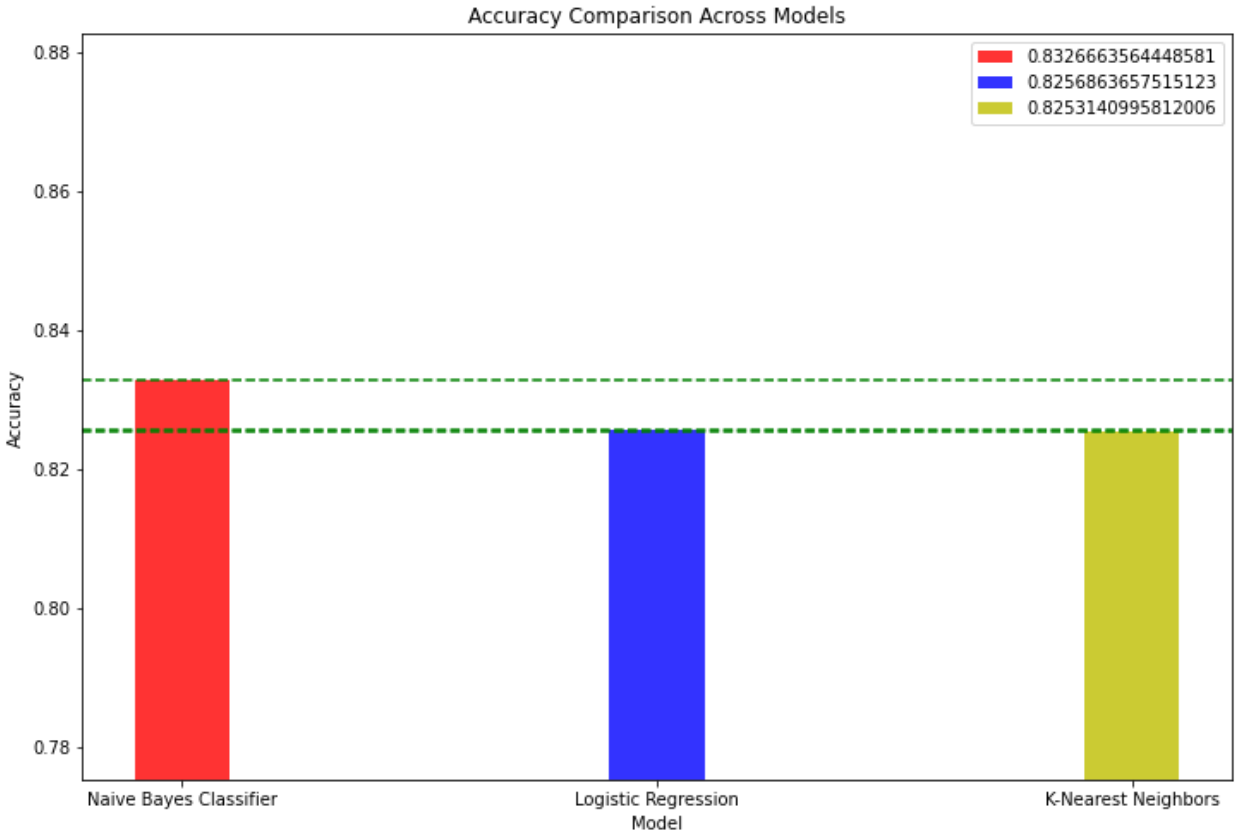
```
# building the model
model = KNeighborsClassifier(n_neighbors=5)
model.fit(X_train, Y_train)

# making predictions on the test set
Y_pred = model.predict(X_test)

# evaluating performance
acc_knn = accuracy_score(Y_test, Y_pred)
cm_knn = confusion_matrix(Y_test, Y_pred)
precision_knn, recall_knn, f1_knn, support_knn = precision_recall_fscore_support(Y_test, Y_pred, average=None)

print('Accuracy:', acc_knn)
print('Confusion Matrix:\n', cm_knn)
print('Precision:', precision_knn)
print('Recall:', recall_knn)
print('F1-score:', f1_knn)
```

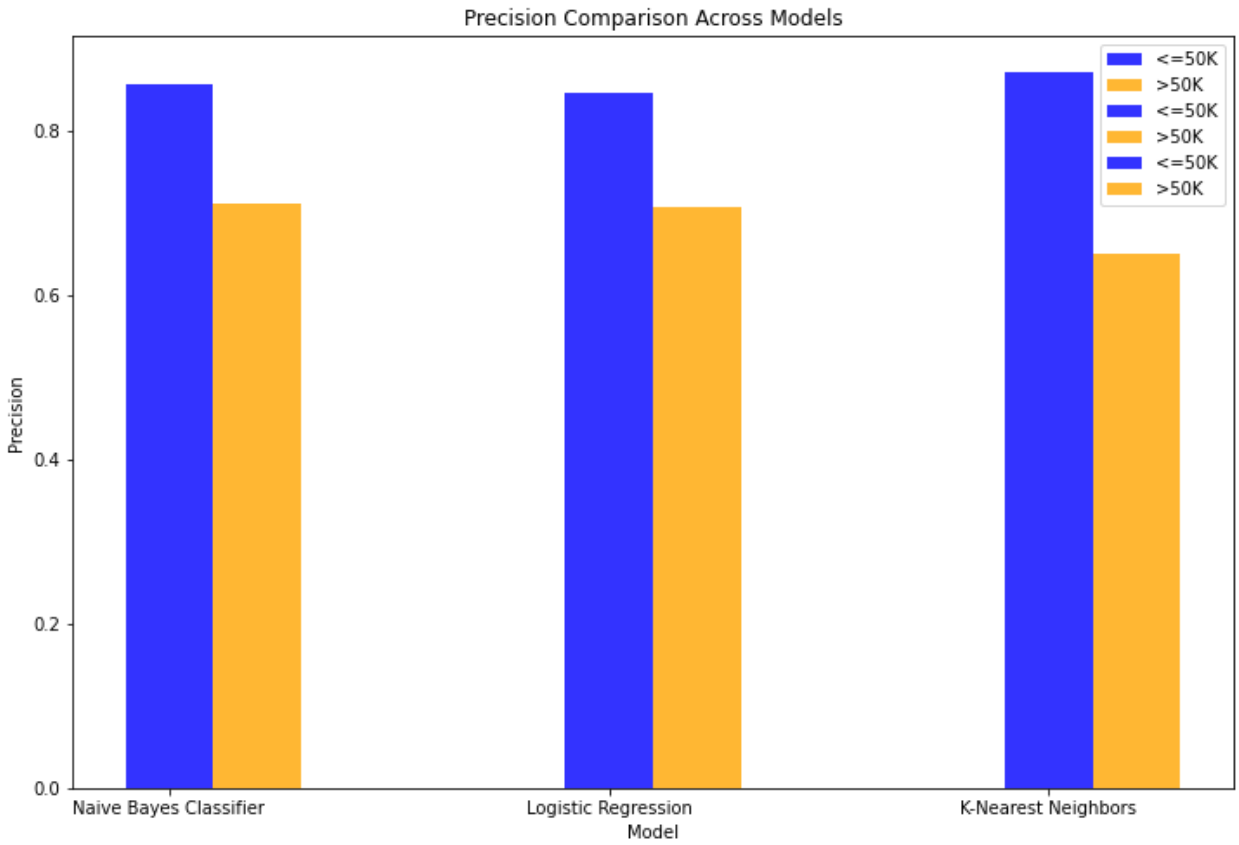
The code above creates a bar chart comparison of performance metrics for three models: Naive Bayes Classifier, Logistic Regression, and K-Nearest Neighbors. The metrics compared include accuracy, precision, recall, and F1-score.
The code first defines a dictionary called metrics where each key is a metric name (e.g., accuracy, precision) and each value is a list of scores for each model. For example, metrics['accuracy'] is a list containing the accuracy scores for the three models.
Next, the code loops through each metric in metrics and creates a bar chart for that metric. For each model, the code adds two bars to the chart: one for the score for '<=50K' predictions and one for the score for '>50K' predictions. For the accuracy metric, the code also adds a horizontal line at the maximum score across all models. The code then adds labels and a legend to the chart before displaying it using plt.show().

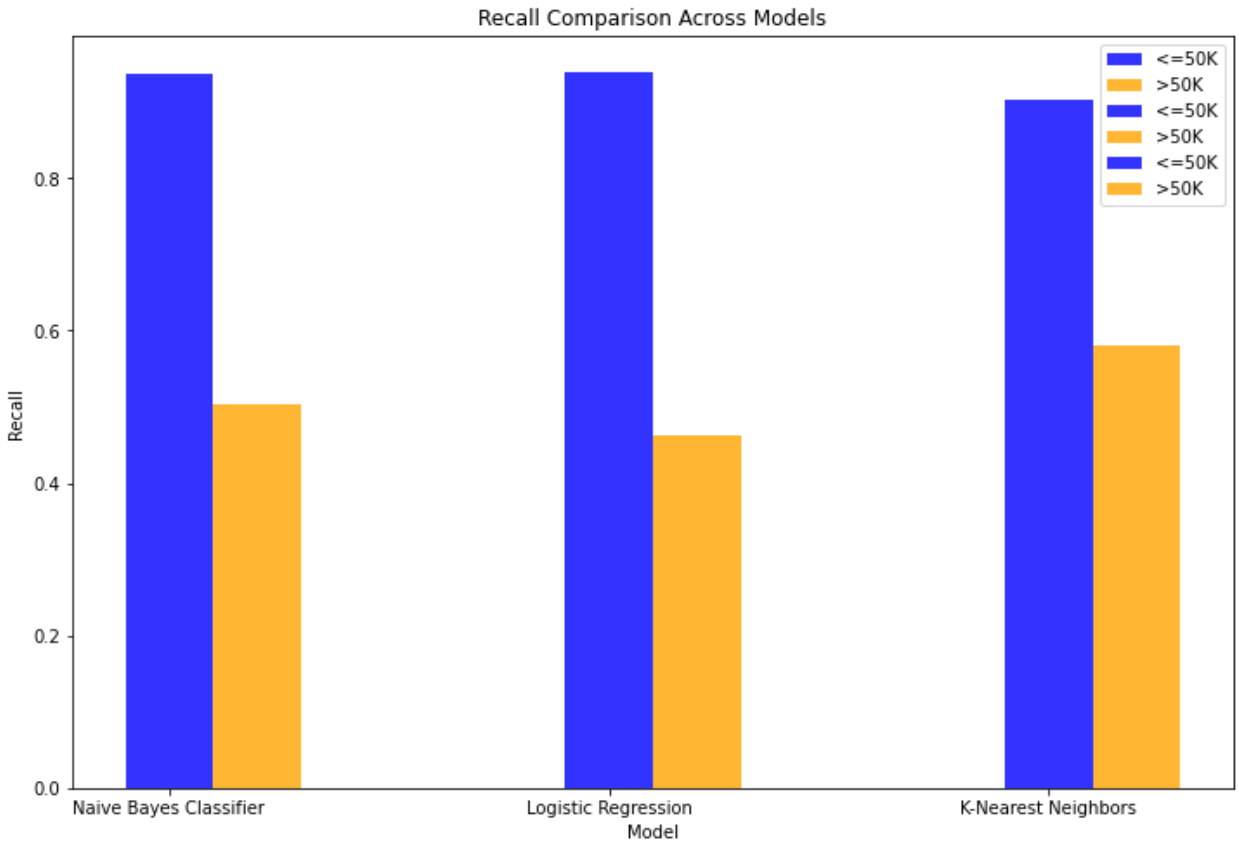
Results

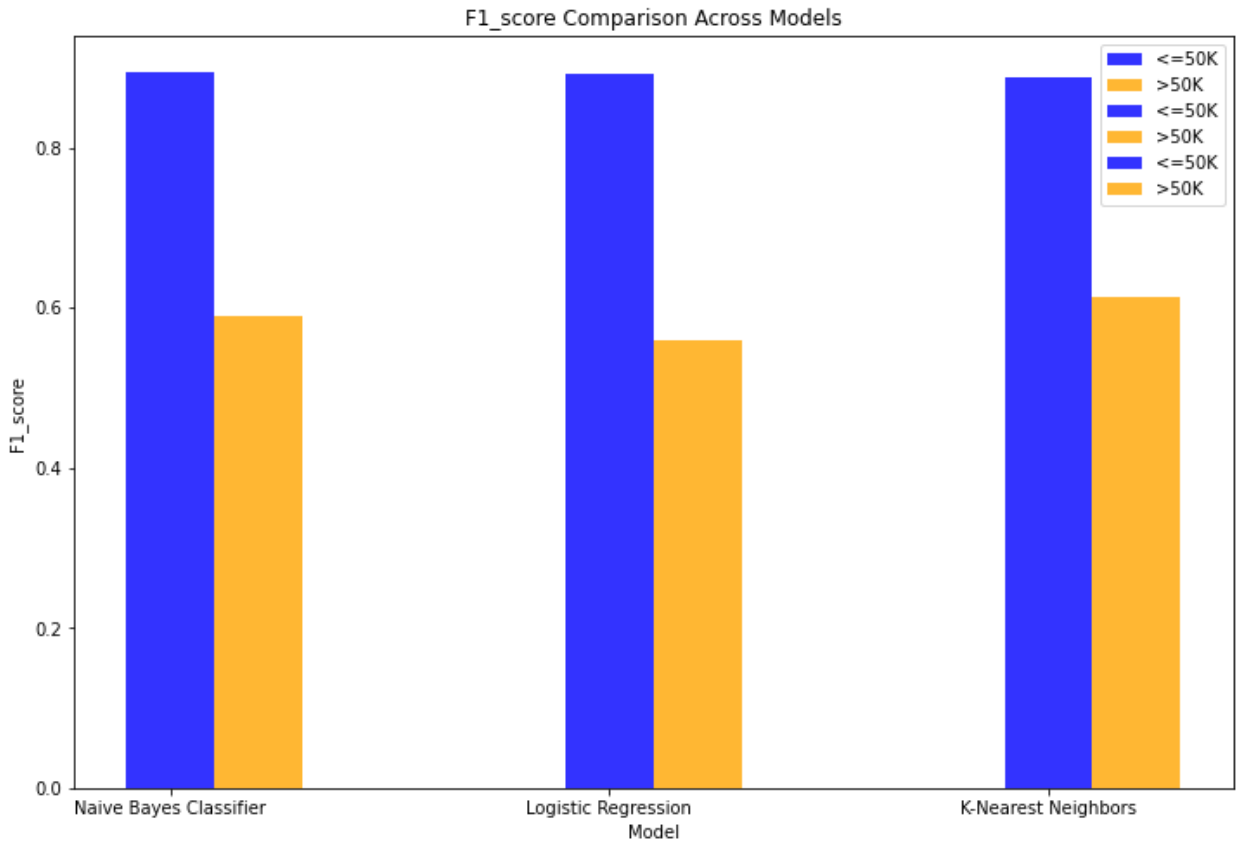
```
Precision, Recall, and F1 Score:
 Class    | Precision | Recall | F1 Score|
 <=50k    |   0.86    |  0.94  |  0.89   |
  >50k    |   0.71    |  0.50  |  0.59   |
```

```
Accuracy with alpha = 1: 0.7696603071195905
```




Precision Comparison Across Models
<=50K
>50K
<=50K
>50K
<=50K
>50K
0.8
0.6
0.4
0.2
0.0
Precision
Naive Bayes Classifier
Logistic Regression
K-Nearest Neighbors
Model

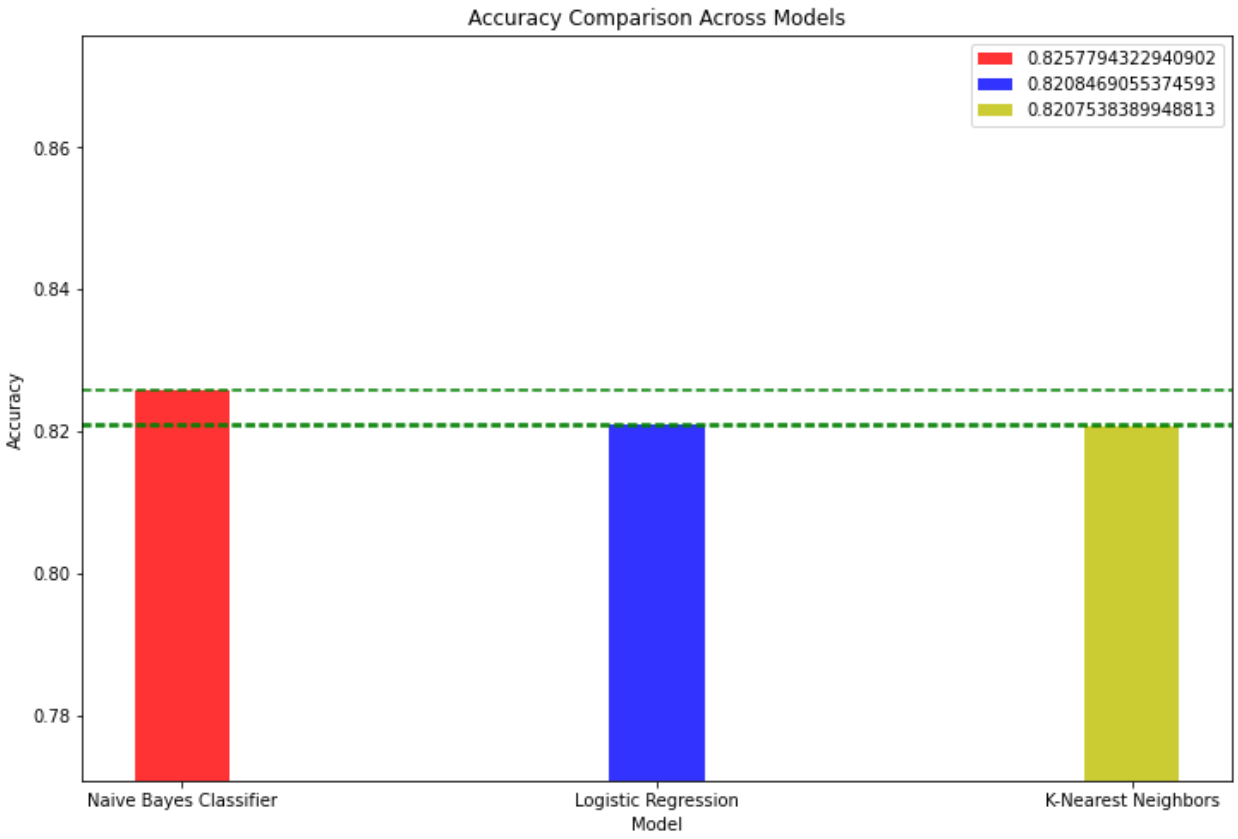

Recall Comparison Across Models
<=50K
>50K
<=50K
>50K
<=50K
>50K
Recall
0.0
0.2
0.4
0.6
0.8
Naive Bayes Classifier
Logistic Regression
K-Nearest Neighbors
Model

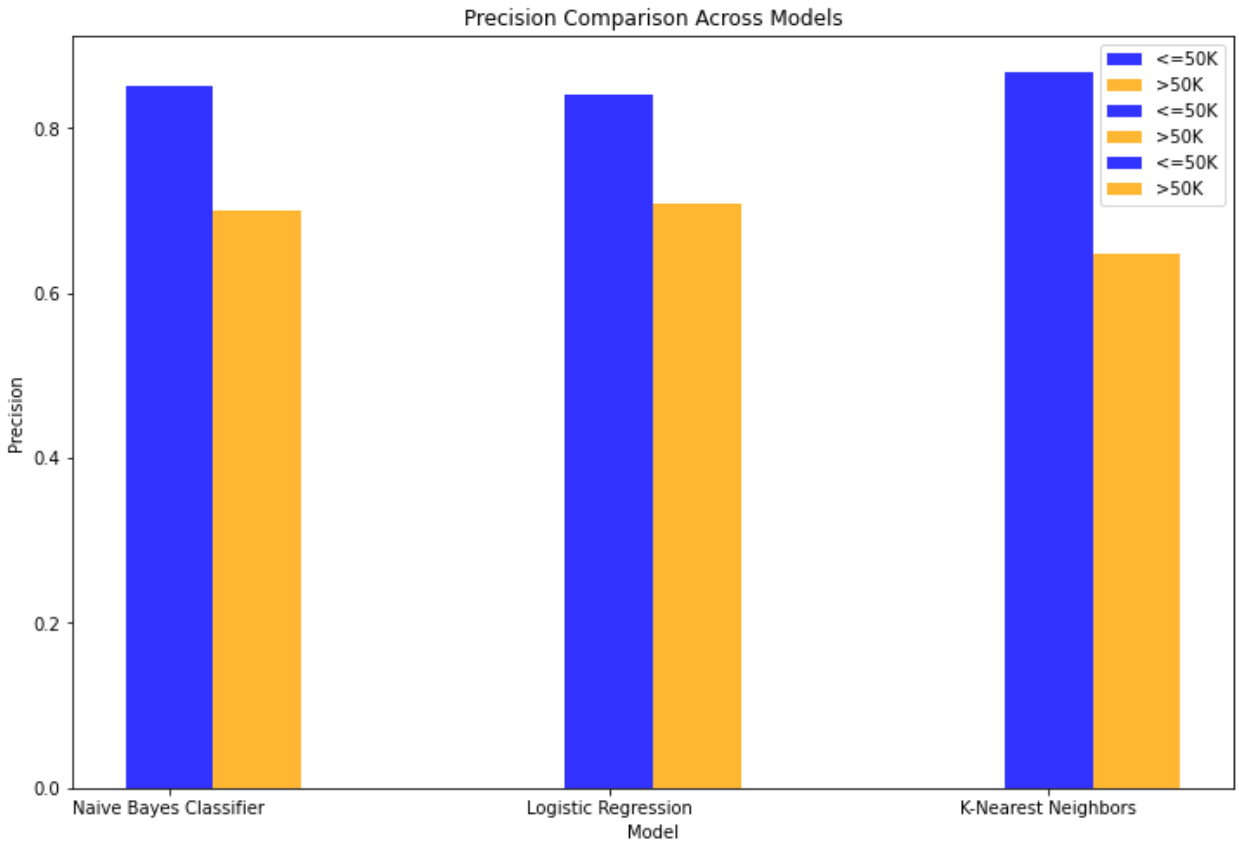


Run 2

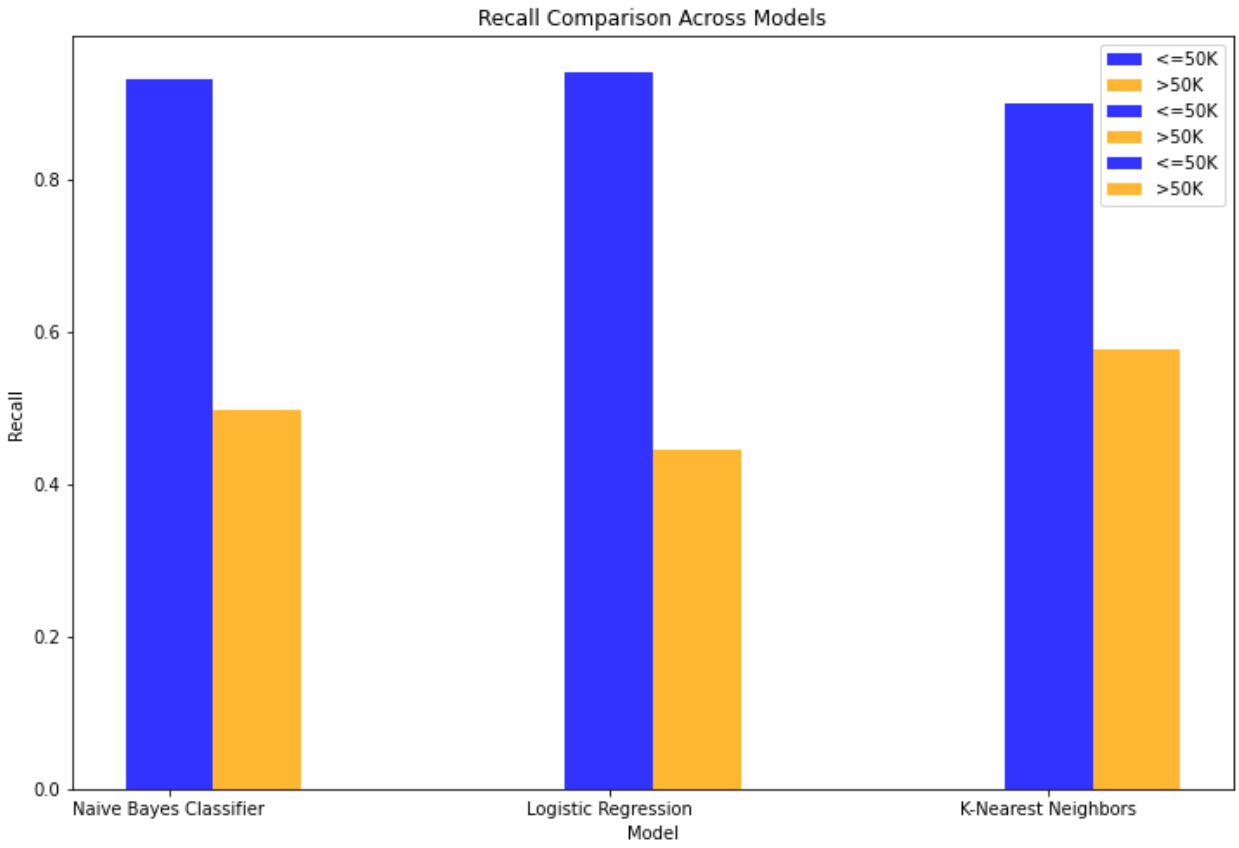
```
Precision, Recall, and F1 Score:
 Class    | Precision | Recall | F1 Score|
 <=50k    |   0.85    |  0.93  |  0.89   |
  >50k    |   0.70    |  0.50  |  0.58   |
```

```
Accuracy with alpha = 1: 0.7652861796184272
```


Accuracy Comparison Across Models
0.8257794322940902
0.8208469055374593
0.8207538389948813
0.86
0.84
0.82
0.80
0.78
Accuracy
Naive Bayes Classifier
Logistic Regression
K-Nearest Neighbors
Model


Precision Comparison Across Models
<=50K
>50K
<=50K
>50K
<=50K
>50K
Precision
0.0
0.2
0.4
0.6
0.8
Naive Bayes Classifier
Logistic Regression
K-Nearest Neighbors
Model

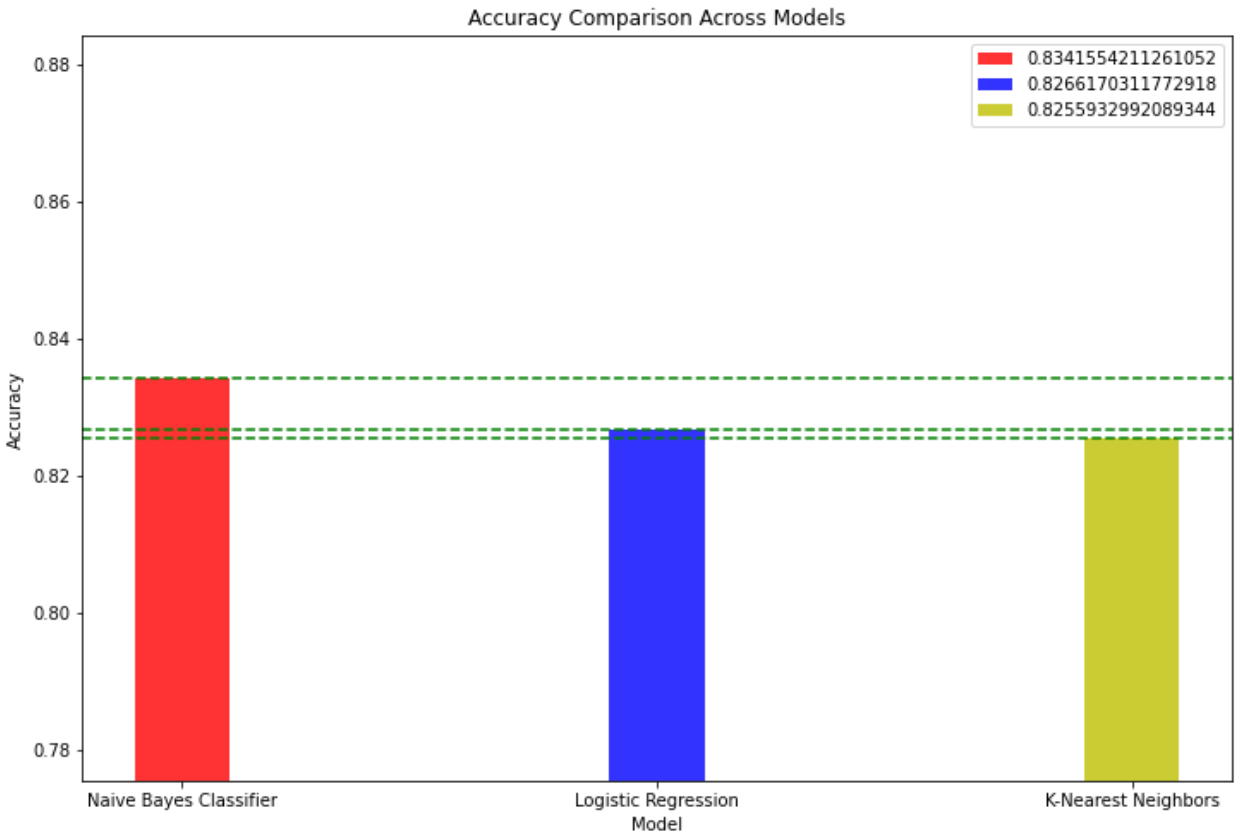
Rrun 3

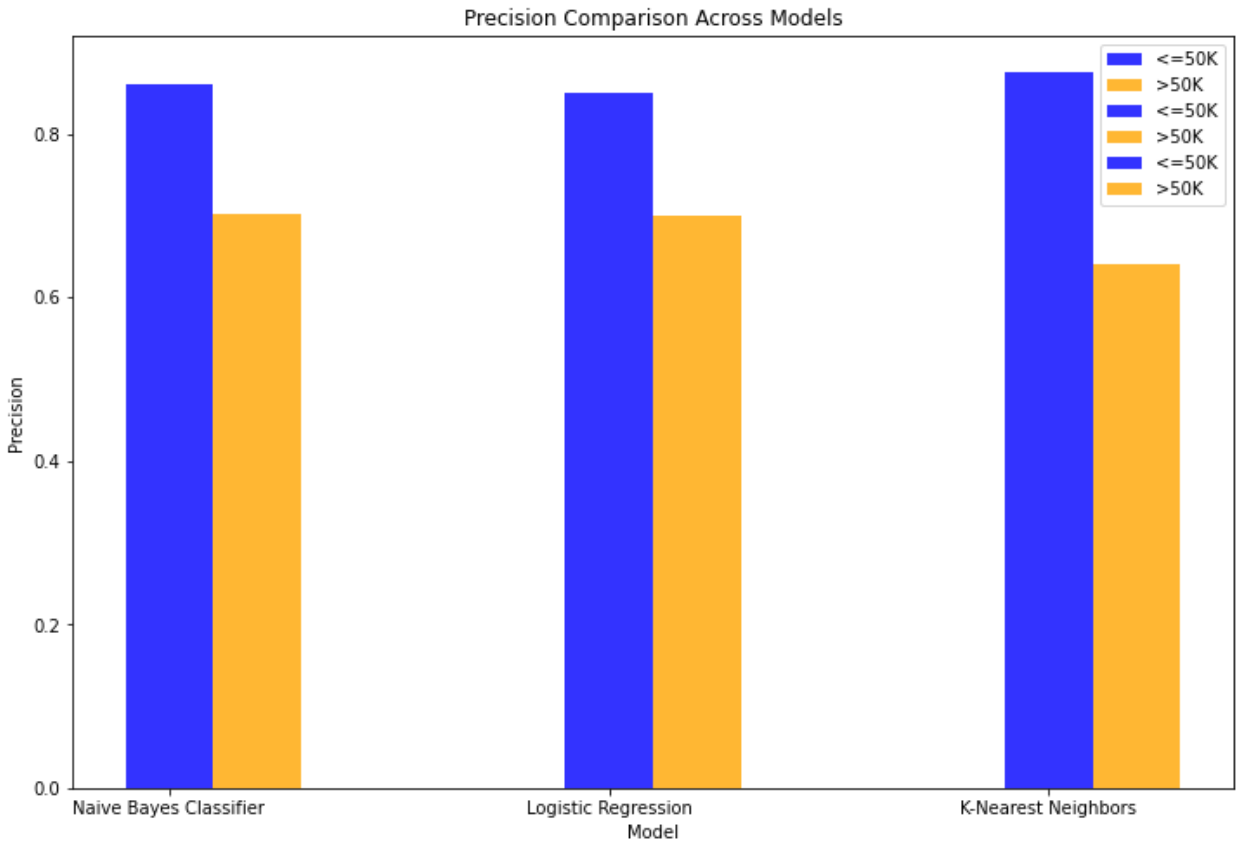


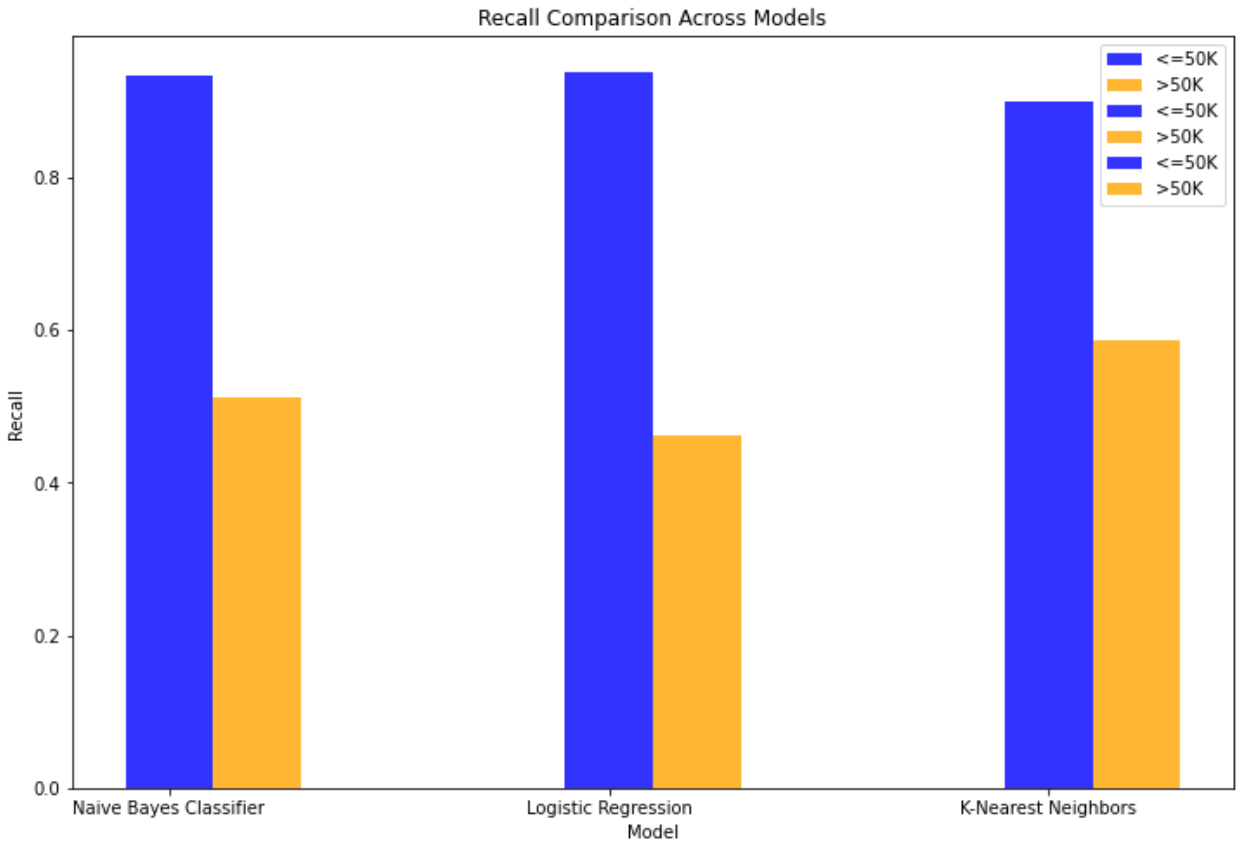
Run 3

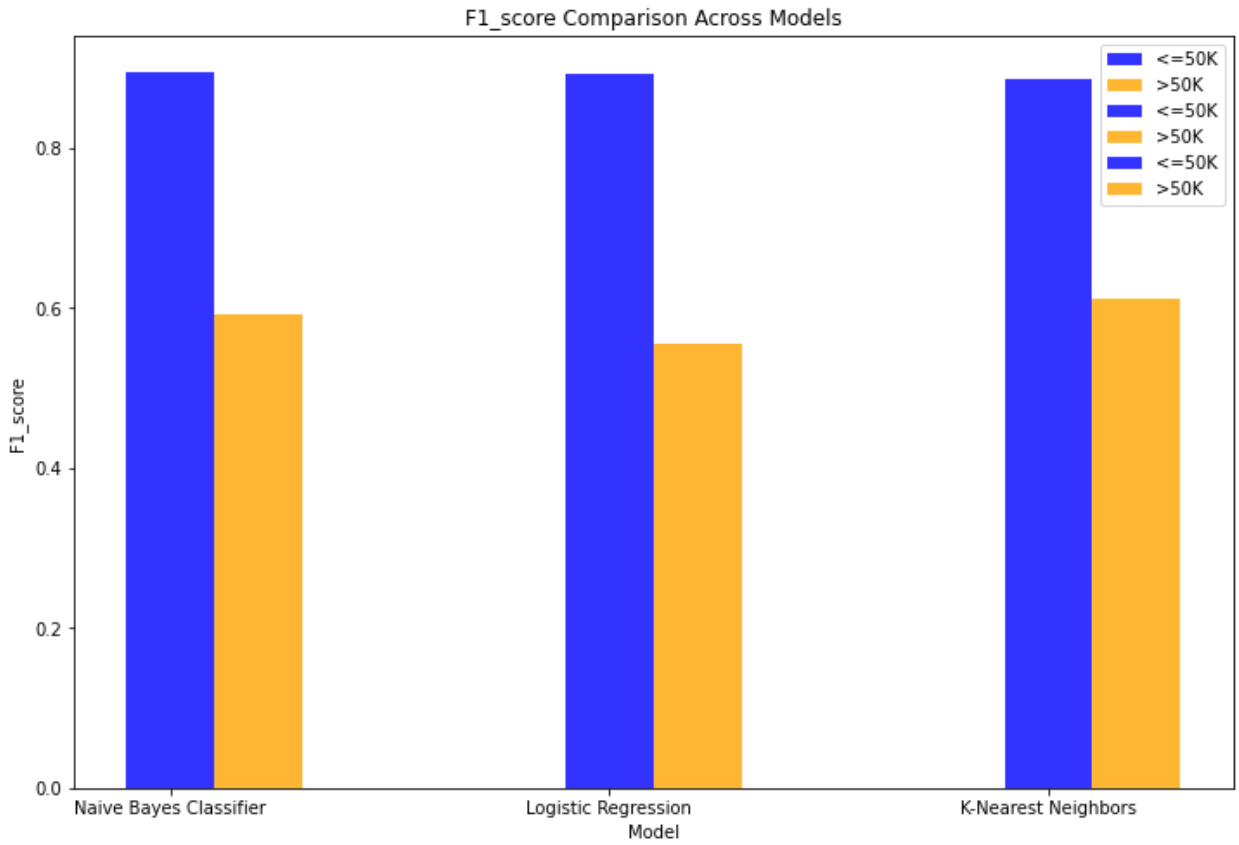
```
Precision, Recall, and F1 Score:
 Class     | Precision | Recall | F1 Score|
 <=50k     |   0.86    |  0.93  |  0.90   |
  >50k     |   0.70    |  0.51  |  0.59   |
```

```
Accuracy with alpha = 1: 0.7746859004187995
```


Accuracy Comparison Across Models
0.8341554211261052
0.8266170311772918
0.8255932992089344
0.88
0.86
0.84
0.82
0.80
0.78
Accuracy
Naive Bayes Classifier
Logistic Regression
K-Nearest Neighbors
Model


Precision Comparison Across Models
<=50K
>50K
<=50K
>50K
<=50K
>50K
Precision
0.0
0.2
0.4
0.6
0.8
Naive Bayes Classifier
Logistic Regression
K-Nearest Neighbors
Model

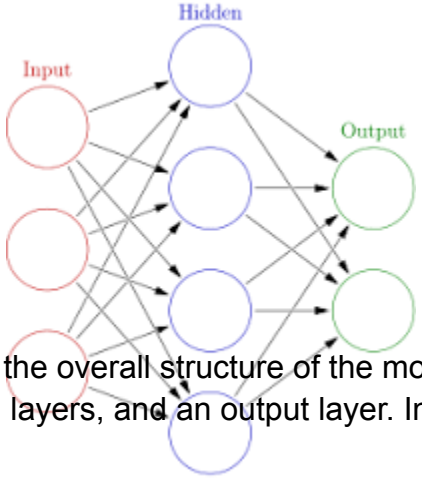

Recall Comparison Across Models
<=50K
>50K
<=50K
>50K
<=50K
>50K
Recall
0.0
0.2
0.4
0.6
0.8
Naive Bayes Classifier
Logistic Regression
K-Nearest Neighbors
Model



And so on….. We have made 10 runs

Part B: Building a Basic Neural Network for Image Classification

A neural network is a method in artificial intelligence that teaches computers to process data in a way that is inspired by the human brain. It is a type of machine learning process, called deep learning, that uses interconnected nodes or neurons in a layered structure that resembles the human brain.



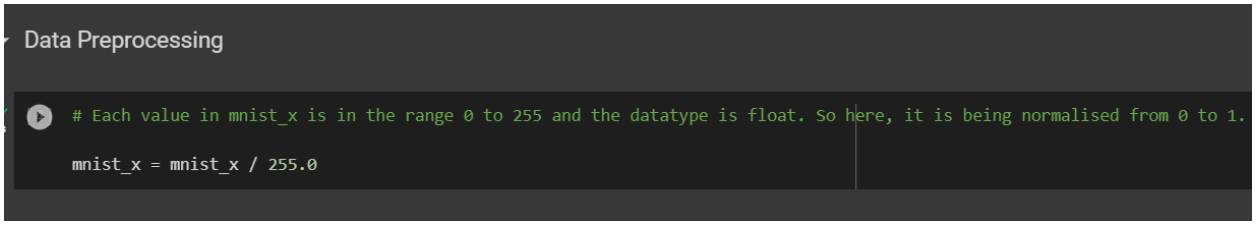
The architecture of a neural network refers to the overall structure of the model, which consists of an input layer, one or more hidden layers, and an output layer. In this project,

there are variations in the number of hidden layers and neurons used in the models. Specifically, there are two variations for the number of hidden layers, which are either 2 or 3. There are also two variations for the total number of neurons in the hidden layer, which are either 100 or 150. The input layer consists of neurons that represent the input features, while the output layer produces the predicted output. The hidden layers, which come between the input and output layers, perform computations to extract relevant features from the input and provide a way to model complex relationships between the input and output variables. By varying the number of hidden layers and neurons, different architectures can be tested to determine the optimal configuration for the given task.

Code walkthrough

Data-Preprocessing:Normalizing dataset

mnist_x is being normalized from 0 to 1 by dividing each value by 255.0. The values in mnist_x are originally in the range of 0 to 255, which represents the pixel intensity of the grayscale images in the MNIST dataset. By dividing each pixel intensity value by 255.0, we are scaling the values to a range between 0 and 1.



Train-Test Split

```
[5]  # List of Train-Test Splits
     mnist_train_x = []
     mnist_train_y = []

     mnist_test_x = []
     mnist_test_y = []
```

```
     num_splits = 10

     for i in range(0, 10):
         x_train, x_test, y_train, y_test = train_test_split(mnist_x, mnist_y, test_size=0.33, random_state=57+i)

         mnist_train_x.append(x_train)
         mnist_test_x.append(x_test)

         mnist_train_y.append(y_train)
         mnist_test_y.append(y_test)
```

```
[7]  print(f"Number of Images (Data) in Training Set: {mnist_train_x[0].shape[0]}")
     print(f"Number of Images (Data) in Testing Set: {mnist_test_x[0].shape[0]}")

     Number of Images (Data) in Training Set: 46900
     Number of Images (Data) in Testing Set: 23100
```

Creating ANN model

We are trying to create a dictionary of artificial neural network models in TensorFlow. We have specified some hyperparameters such as the number of hidden layers, the total number of neurons, and activation functions for each layer. Then, we are creating a random architecture for each model by selecting values for these hyperparameters randomly.
After creating each model, we compile it with the Adam optimizer and sparse categorical cross-entropy loss function and then append it to the corresponding split in the dictionary of models.
However, we noticed that we have used the same variable name "split_num," in the outer and inner loops. This may cause some issues as the inner loop will overwrite the value of "split_num" in the outer loop. We may want to use a different variable name for the inner loop. Also, we see that the variable "num_splits" is not defined in our code snippet, so it may cause an error. We may want to define this variable and specify the number of splits we want to create. Other than that, the code looks fine, and it should create a dictionary of 15 artificial neural network models, each with a random architecture.

```
epochs = 5

for split_num in range(num_splits):
    print(f"Split Number: {split_num+1}")
    for i, ann_model in enumerate(ann_models[split_num]):
        print(f'Training model {i+1}...')
        ann_model.fit(mnist_train_x[split_num], mnist_train_y[split_num], epochs=epochs, validation_data=(mnist_test_x[split_num], mnist_test_y[split_num]))
```

Now, we are ready to train the artificial neural network models in the dictionary "ann_models". The training will be done in a nested loop, where the outer loop will iterate over the splits, and the inner loop will iterate over the models in each split.
For each model, we will fit it to the training data using the fit() method in TensorFlow. We have specified the number of epochs for training to be 5, using the variable "epochs". Additionally, we will provide the validation data to the model during training.
During training, we will print the split number and the current model number to keep track of the progress.
Overall, this code should train the neural network models for 5 epochs using the specified training and validation data.

```
Training model 8...
Epoch 1/5
1466/1466 [==============================] - 7s 4ms/step - loss: 0.5144 - accuracy: 0.8799 - val_loss: 0.2259 - val_accuracy: 0.9383
Epoch 2/5
1466/1466 [==============================] - 6s 4ms/step - loss: 0.1723 - accuracy: 0.9516 - val_loss: 0.1533 - val_accuracy: 0.9552
Epoch 3/5
1466/1466 [==============================] - 6s 4ms/step - loss: 0.1202 - accuracy: 0.9655 - val_loss: 0.1384 - val_accuracy: 0.9587
Epoch 4/5
1466/1466 [==============================] - 6s 4ms/step - loss: 0.0931 - accuracy: 0.9730 - val_loss: 0.1306 - val_accuracy: 0.9612
Epoch 5/5
1466/1466 [==============================] - 5s 4ms/step - loss: 0.0752 - accuracy: 0.9782 - val_loss: 0.1184 - val_accuracy: 0.9648
Training model 9...
```

Now, we will evaluate the performance of the trained neural network models in the dictionary "ann_models" on the test data. The evaluations will be done in a nested loop, where the outer loop will iterate over the splits, and the inner loop will iterate over the models in each split.
For each model, we will evaluate its performance using the evaluate() method in TensorFlow. The method returns the loss and the accuracy of the model on the specified test data. Since we are only interested in the accuracy, we will append only the second element of the returned value to the list "results[split_num]".
We will also create a dictionary "results", where each key will represent a split, and the corresponding value will be a list of accuracies of the models in that split.
During evaluation, we will print the split number to keep track of the progress.
Overall, this code should evaluate the performance of the trained neural network models on the test data and store the accuracy results in the dictionary "results".

```
results = {}

for split_num in range(num_splits):
    print(f"For Split {split_num+1}: ")
    results[split_num] = []
    for model in ann_models[split_num]:
        results[split_num].append(model.evaluate(mnist_test_x[split_num], mnist_test_y[split_num])[1])
```

In this code, we calculate the best accuracy and average accuracy across all the trained models in each split.
We initialize three lists: "best_accuracy", "average_accuracy", and "best_accuracy_split" to keep track of the best accuracy, average accuracy, and the split number where the best accuracy was obtained, respectively.
Then, we iterate over the trained models using a for loop, and for each model, we check the accuracy achieved in each split. If the accuracy is better than the previous best accuracy for that model, we update the best_accuracy, best_accuracy_split, and average_accuracy for that model. We also update the average_accuracy for each model by adding the accuracy of that model in the current split divided by the total number of models.
At the end of this code, we have three lists: "best_accuracy", "average_accuracy", and "best_accuracy_split". The list "best_accuracy" contains the best accuracy achieved for each model, the list "average_accuracy" contains the average accuracy achieved for each model across all the splits, and the list "best_accuracy_split" contains the split number where the best accuracy was obtained for each model.

```
best_accuracy = []
average_accuracy = []
best_accuracy_split = []

for i in range(num_models):
    best_accuracy.append(0)
    average_accuracy.append(0)
    best_accuracy_split.append(0)

    for split_num in range(num_splits):
        if results[split_num][i] > best_accuracy[i]:
            best_accuracy_split[i] = split_num
        best_accuracy[i] = max(best_accuracy[i], results[split_num][i])
        average_accuracy[i] += (results[split_num][i]/num_models)
```

Then we have plotted graphs for the best accuracy model and the average accuracy using the following functions.:

```
plt.plot(range(len(best_accuracy)), best_accuracy)
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Accuracy of ANN models')
plt.show()
```

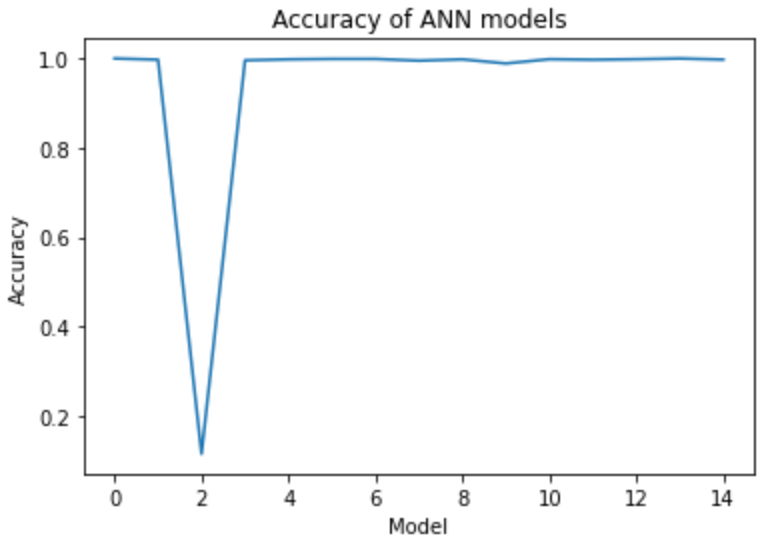
```
plt.plot(range(len(average_accuracy)), average_accuracy)
plt.xlabel('Model')
plt.ylabel('Accuracy')
plt.title('Accuracy of ANN models')
plt.show()
```

This function takes a confusion matrix and a list of classes as input and plots the confusion matrix as a heatmap.
We start by setting up the plot with appropriate labels and tick marks. We then calculate the threshold for the text color based on the maximum value in the confusion matrix. Next, we loop over each cell in the confusion matrix and add the value as text to the center of the cell. If the value is greater than the threshold, we set the text color to white, otherwise black.
Finally, we show the plot.

```
def plot_confusion_matrix(confusion_matrix, classes, title="Confusion Matrix", cmap = plt.cm.Blues):
  plt.imshow(confusion_matrix, interpolation='nearest', cmap=cmap)
  plt.title(title)
  plt.colorbar()
  tick_marks = np.arange(len(classes))
  plt.xticks(tick_marks, classes, rotation=45)
  plt.yticks(tick_marks, classes)

  thresh = confusion_matrix.max() / 2.
  for i, j in itertools.product(range(confusion_matrix.shape[0]), range(confusion_matrix.shape[1])):
    plt.text(j, i, format(confusion_matrix[i, j], 'd'), horizontalalignment = 'center', color = "white" if confusion_matrix[i, j] > thresh else "black")

  plt.tight_layout()
  plt.ylabel('True label')
  plt.xlabel('Predicted label')
  plt.show()
```

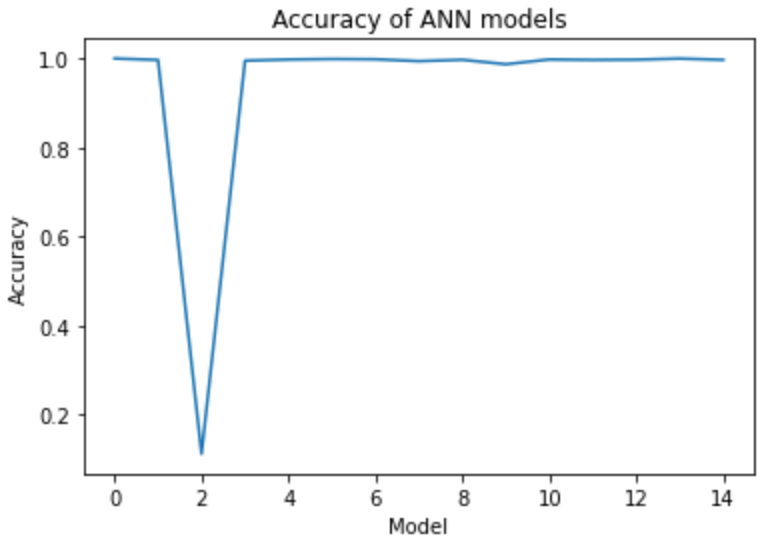
For each model, we first predict the labels of the test set using the trained ANN model. We then calculate the confusion matrix using the predicted labels and the actual labels of the test set.
Finally, we pass the confusion matrix and a list of classes to the plot_confusion_matrix function to plot the confusion matrix as a heatmap.
Note that in the loop, split_num is not defined, so it should be replaced with best_accuracy_split_ind to correctly get the split index with the best accuracy for each model.

```
for i, best_accuracy_split_ind in enumerate(best_accuracy_split):
  print(f"For Model {i}")
  predicted_test = ann_models[split_num][i].predict(mnist_test_x[split_num]).argmax(axis=1)
  con_matrix = confusion_matrix(mnist_test_y[split_num], predicted_test)
  plot_confusion_matrix(con_matrix, list(range(10)))
```



This is the plot for the best accuracy, as we can see among the five models all except model 2 has been accurate at level>99% the reason might be:

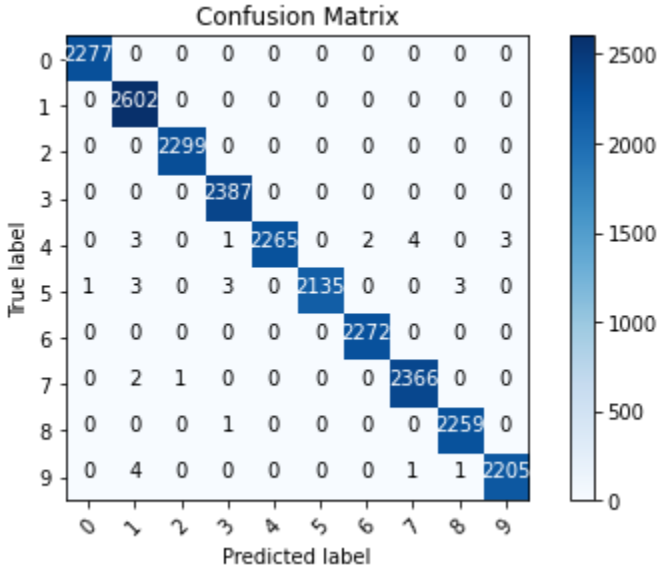
1. Hyperparameters: Hyperparameters are values that are set before training a model and can have a significant impact on its performance. It's possible that the hyperparameters chosen for the second model were not optimal, leading to lower accuracy.
2. Overfitting: Overfitting occurs when a model becomes too complex and starts to fit the noise in the training data instead of the underlying pattern. It's possible that the second model was overfitting the training data, leading to lower accuracy on new, unseen data.
3. Randomness: Many machine learning models include an element of randomness in their training process, which can lead to slightly different results each time the model is trained. It's possible that the second model was simply unlucky in the randomness of its training process and ended up with lower accuracy as a result.



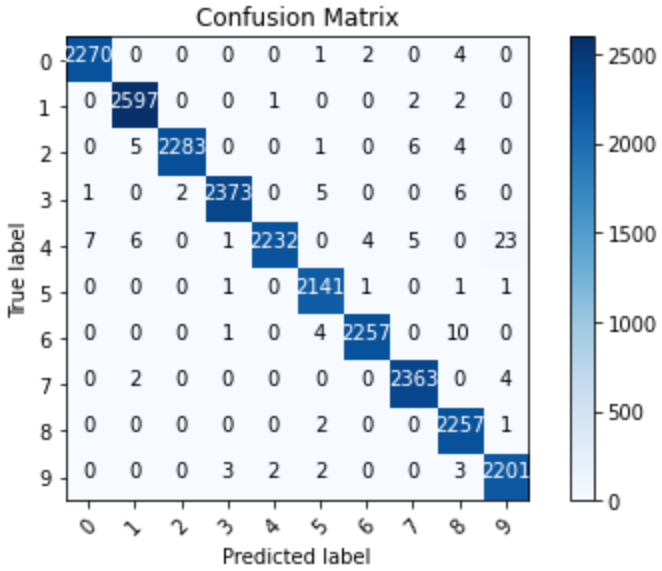
This graphs plots the average accuracy amongst all 15 models and again we can see that all models except model 2 have high accuracies.

For the best accuracy model:

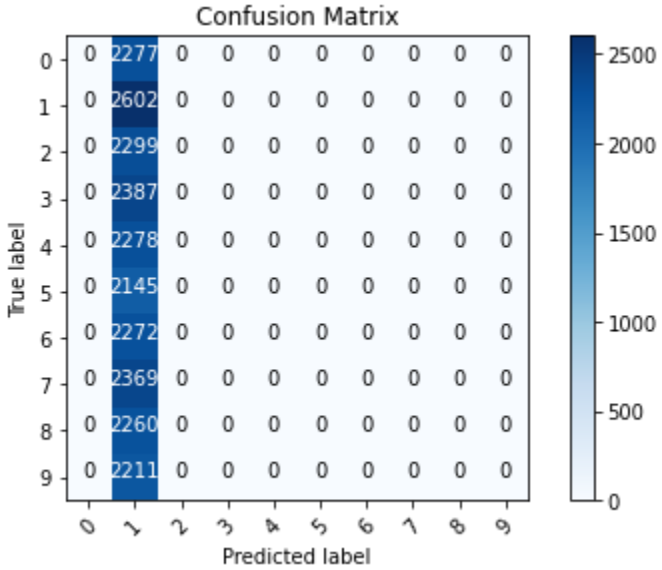
These are the following confusion matrix for each corresponding model.



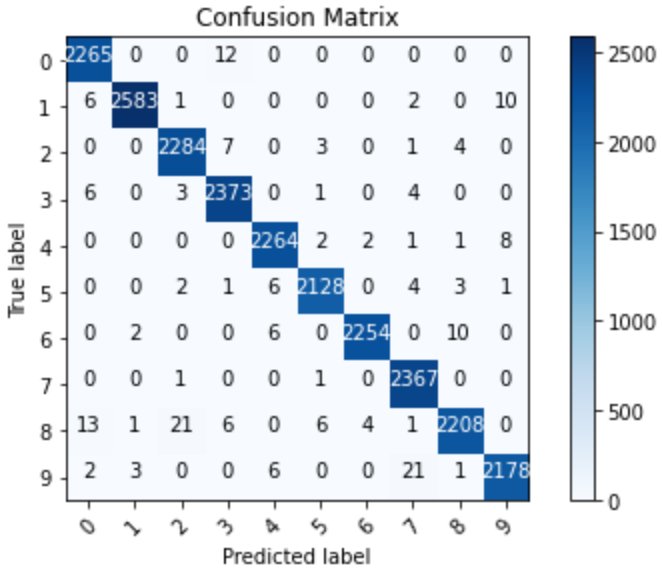
Model 0



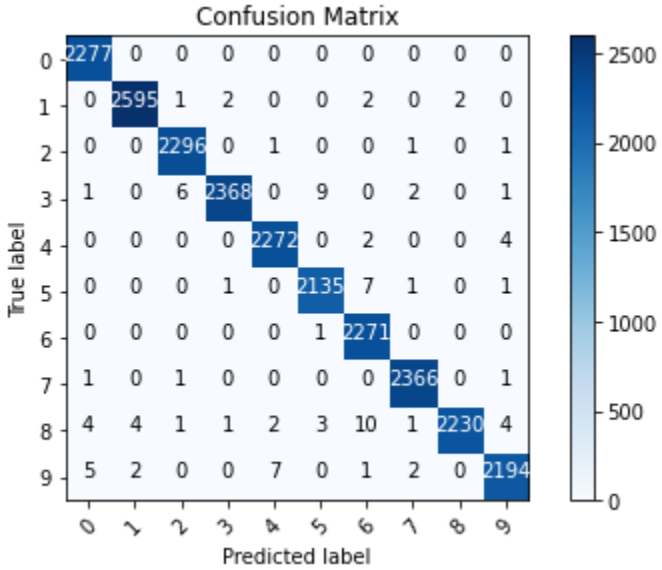
Model 1



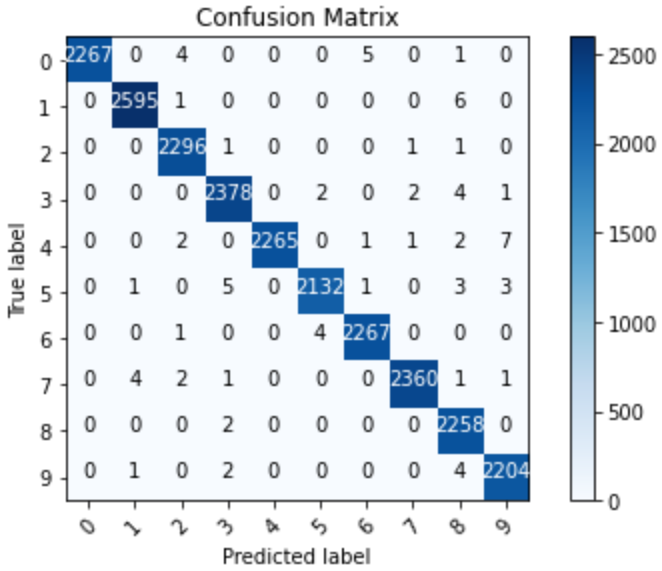
Model 2



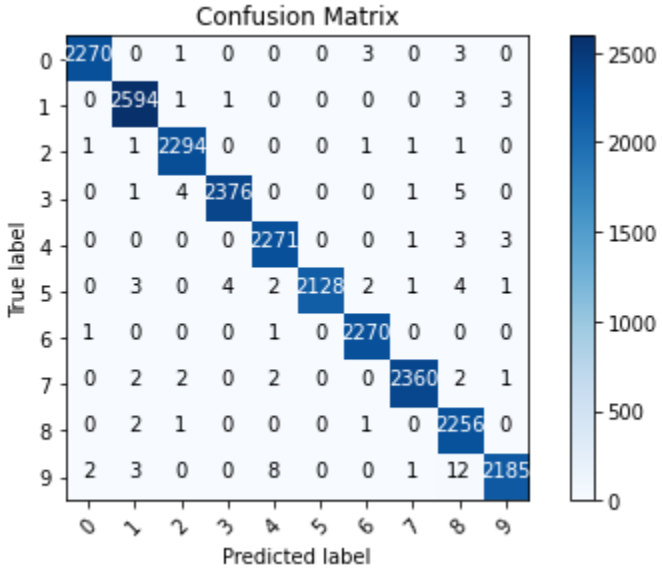
Model 3



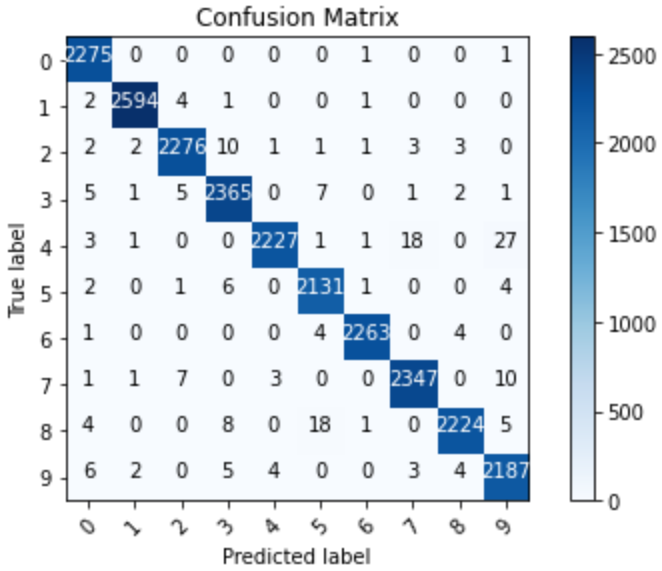
Model 4



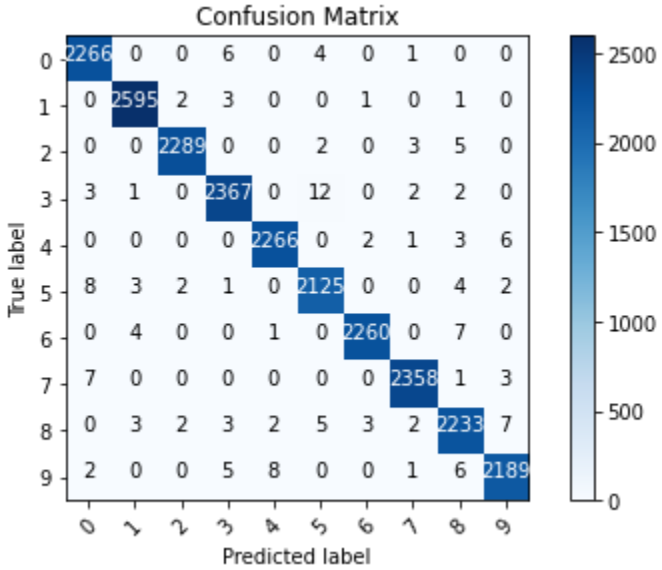
Model 5



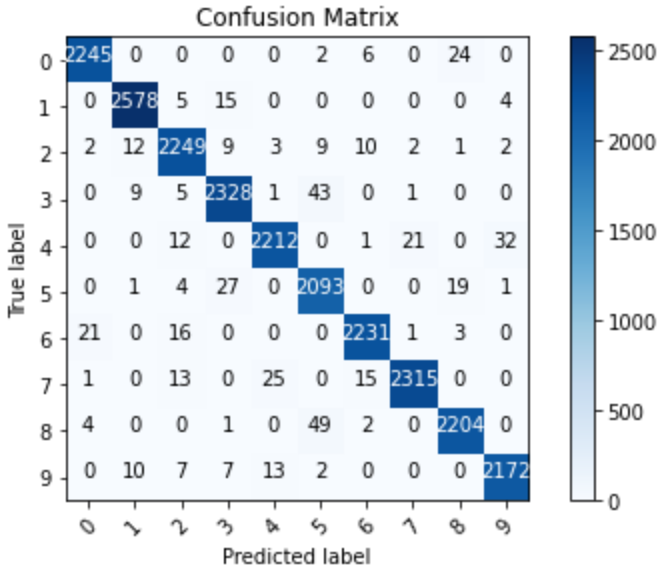
Model 6



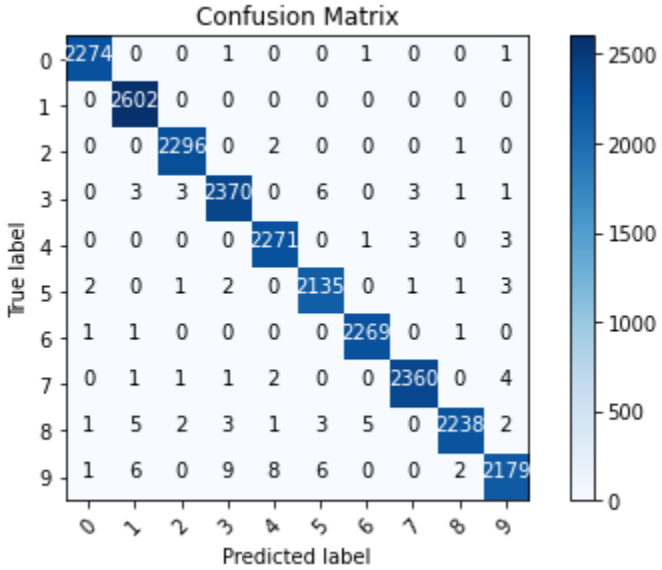
Model 7



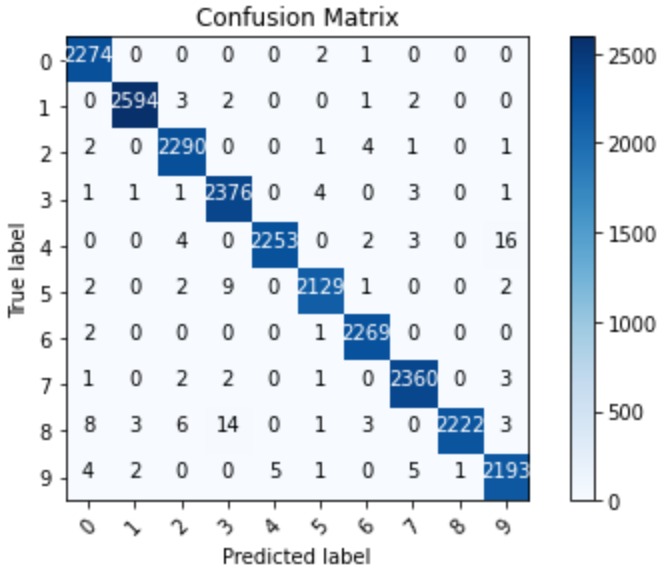
Model 8



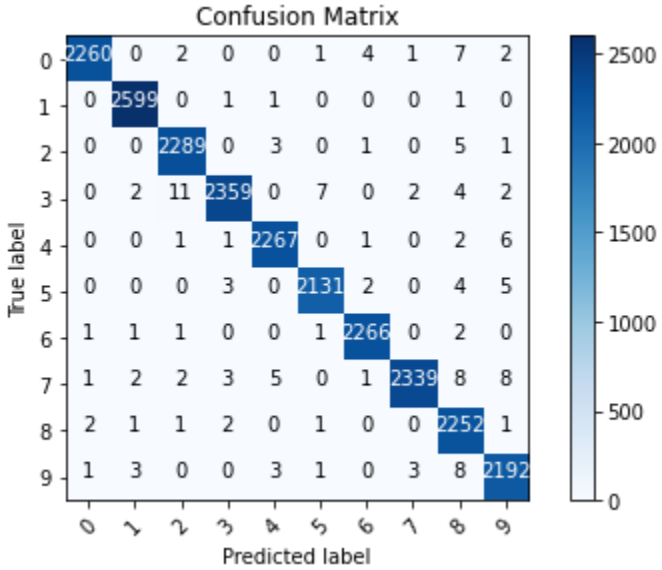
Model 9



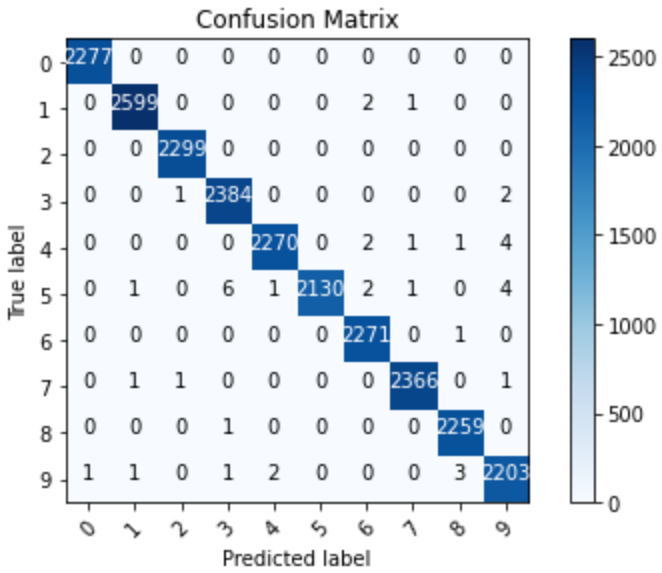
Model 10



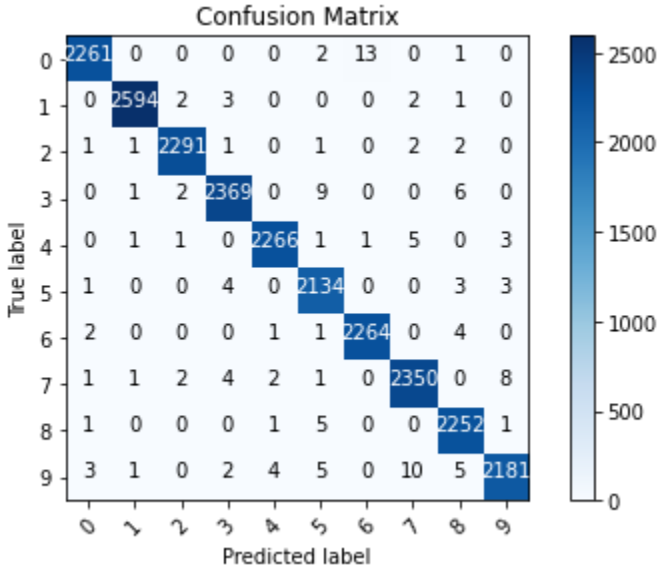
Model 11



Model 12



Model 13



Model 14

Final Conclusion:

One of the model/network has just a single neuron in the last hidden layer and hence predicts just one output which is not accurate as is tries to reduce the error in prediction by reducing the error by predicting the most likely output.

The maximum difference between the accuracies of most of the other classifiers/network is negligible.