<u>**BITS F464 MACHINE LEARNING**</u>

<u>**Assignment-2**</u>

Group details:

Dhairya Agrawal (2020A7PS0130H)

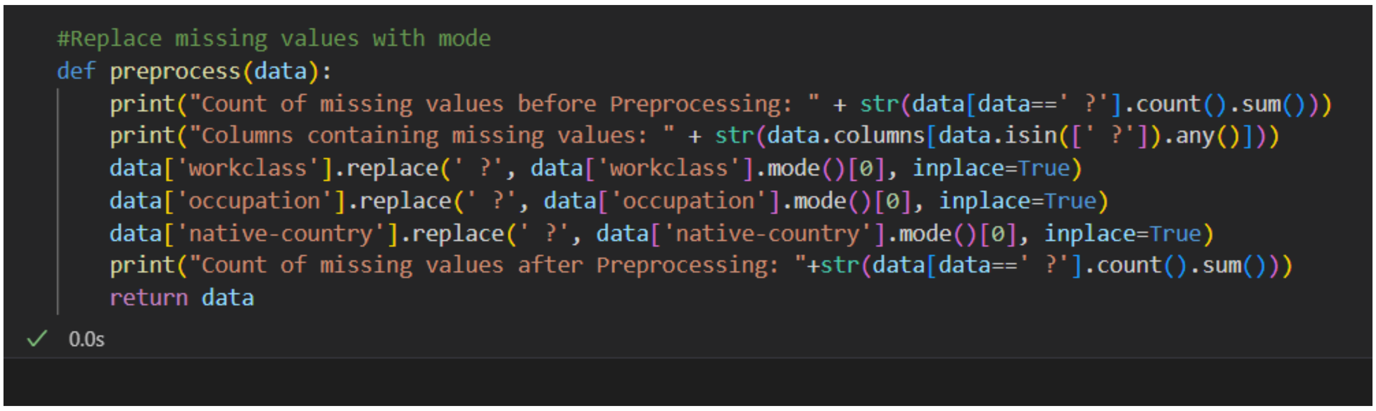
Pulkit Agrawal (2020A7PS2072H)

Mohit Agarwal (2020A7PS0189H)

PART A-Naive Bayes Classifier to predict income

Task 1: Data Preprocessing

The given task required us to import the dataset and handle the missing values and split the data into training and testing data, however, since the original data was already divided into the testing and training datasets so we used the already split dataset for our implementation. On inspection, we found that the missing values were only categorical and were present in only the work class, occupation, and native-country features/columns. So we replaced the missing values in them with the mode (the most frequently occurring value).

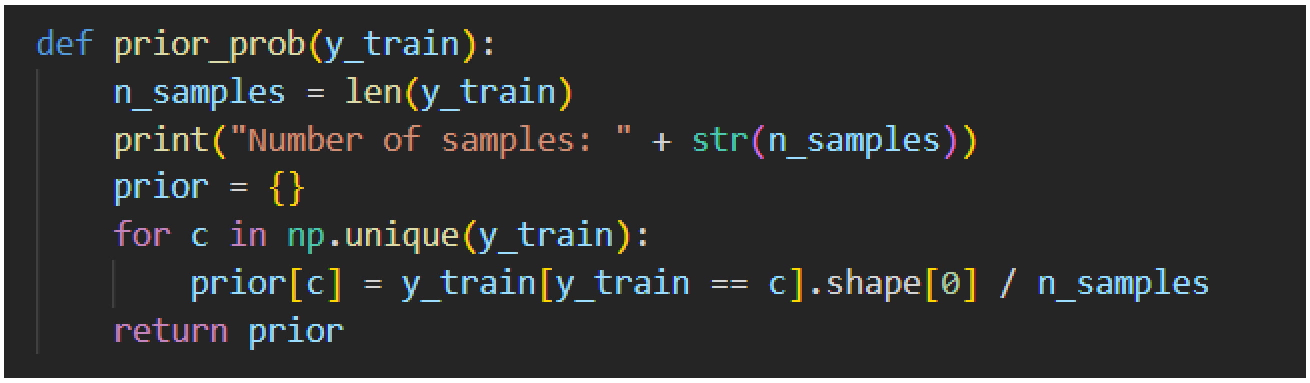


```python
#Replace missing values with mode
def preprocess(data):
    print("Count of missing values before Preprocessing: " + str(data[data==' ?'].count().sum()))
    print("Columns containing missing values: " + str(data.columns[data.isin([' ?']).any()]))
    data['workclass'].replace(' ?', data['workclass'].mode()[0], inplace=True)
    data['occupation'].replace(' ?', data['occupation'].mode()[0], inplace=True)
    data['native-country'].replace(' ?', data['native-country'].mode()[0], inplace=True)
    print("Count of missing values after Preprocessing: "+str(data[data==' ?'].count().sum()))
    return data
```

✓ 0.0s

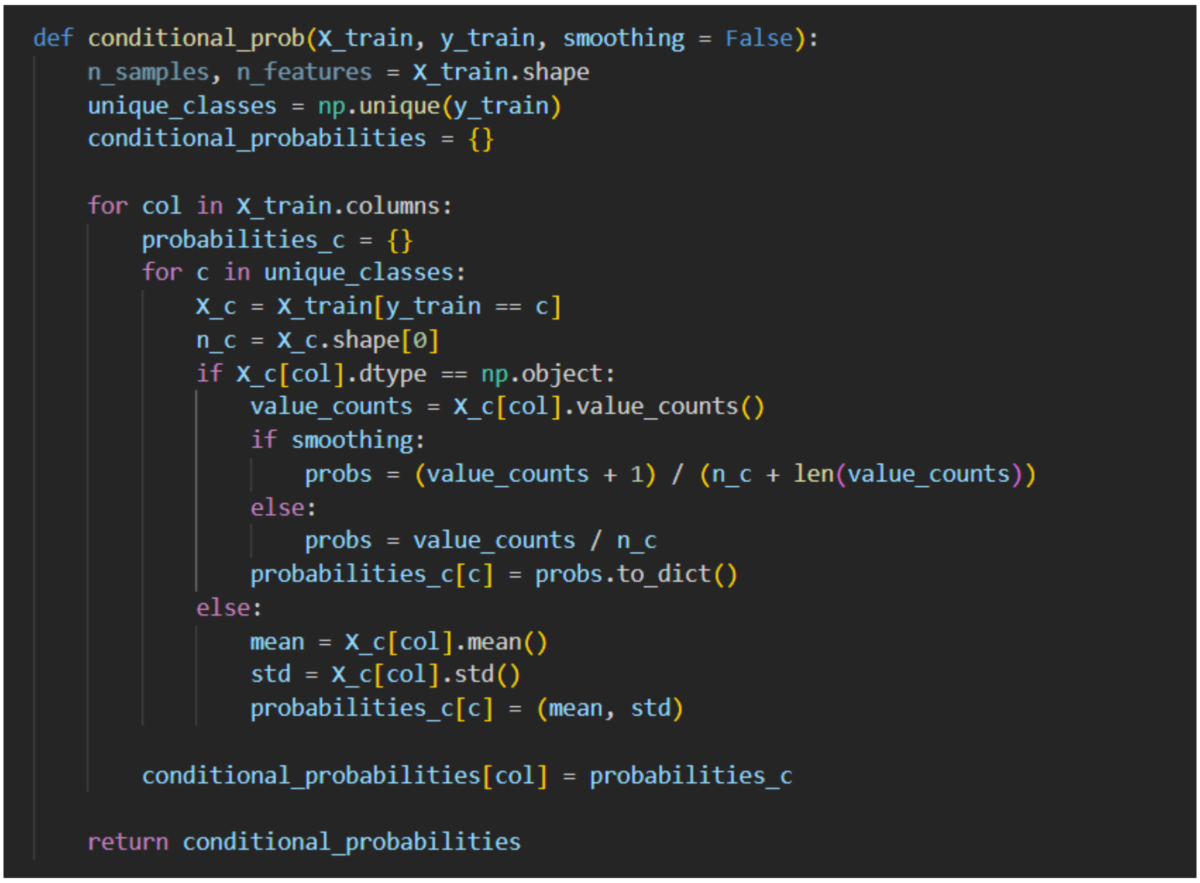
Task-2:Naive Bayes Classifier Implementation

The first given sub-task required us to implement a function to calculate the prior probability of each class. Our dataset has only two classes, ‘ <=50K’ and ‘ >50K’. So for the respective classes, we calculated the number of occurrences of that class divided by the total number of examples.



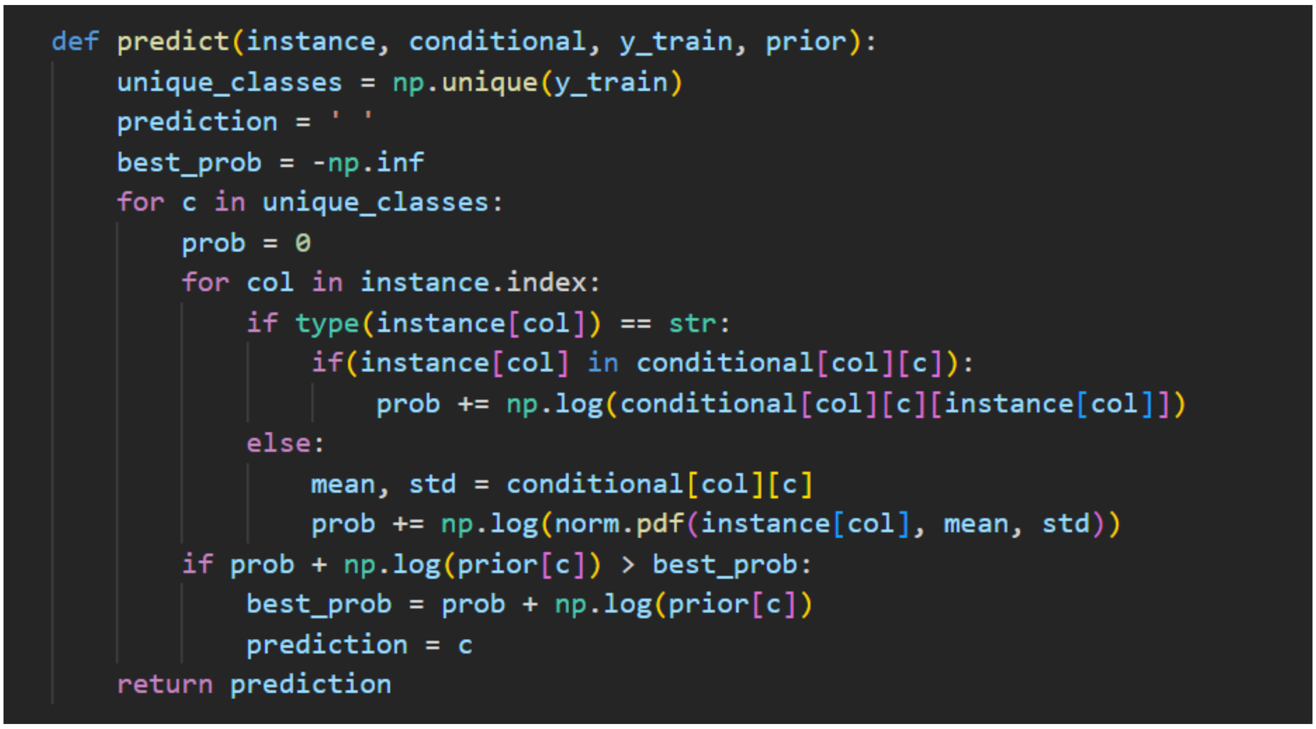
```
def prior_prob(y_train):
    n_samples = len(y_train)
    print("Number of samples: " + str(n_samples))
    prior = {}
    for c in np.unique(y_train):
        prior[c] = y_train[y_train == c].shape[0] / n_samples
    return prior
```

The given sub-task required us to implement a function to calculate the conditional probability of each feature in the training set. For this purpose, for each feature, we must calculate the conditional probability for each feature given a particular class. The dataset has two types of features: continuous features and categorical ones. For categorical features, we found the conditional probability for each unique feature's unique value. For continuous features, we found the mean and standard deviation of the values to create a normal distribution and finally predict a probability given a value.



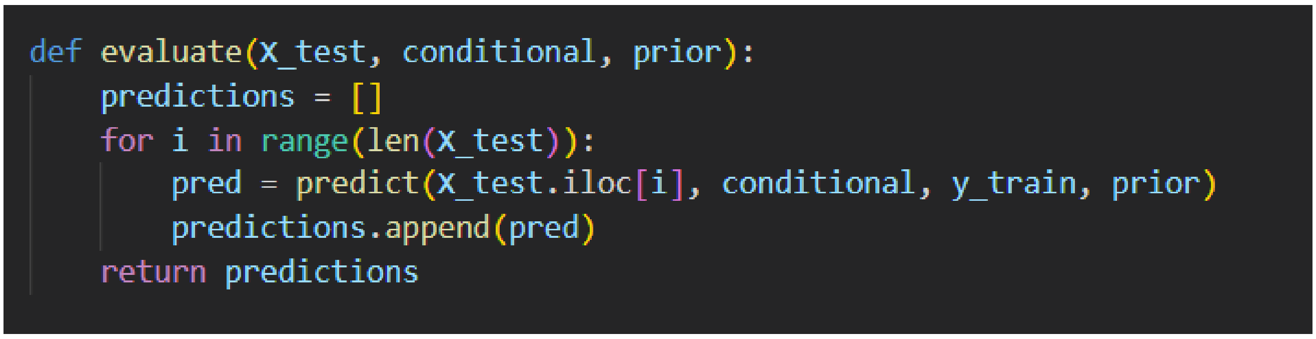
```python
def conditional_prob(X_train, y_train, smoothing = False):
    n_samples, n_features = X_train.shape
    unique_classes = np.unique(y_train)
    conditional_probabilities = {}

    for col in X_train.columns:
        probabilities_c = {}
        for c in unique_classes:
            X_c = X_train[y_train == c]
            n_c = X_c.shape[0]
            if X_c[col].dtype == np.object:
                value_counts = X_c[col].value_counts()
                if smoothing:
                    probs = (value_counts + 1) / (n_c + len(value_counts))
                else:
                    probs = value_counts / n_c
                probabilities_c[c] = probs.to_dict()
            else:
                mean = X_c[col].mean()
                std = X_c[col].std()
                probabilities_c[c] = (mean, std)

        conditional_probabilities[col] = probabilities_c

    return conditional_probabilities
```

The given sub-task required us to implement a function to predict the class of a given instance using the Naive Bayes algorithm. For this sub-task, we multiplied each of the values of the conditional probability of the instance for categorical. We found out the probability using the normal distribution for the continuous features of the instance.



```python
def predict(instance, conditional, y_train, prior):
    unique_classes = np.unique(y_train)
    prediction = ' '
    best_prob = -np.inf
    for c in unique_classes:
        prob = 0
        for col in instance.index:
            if type(instance[col]) == str:
                if(instance[col] in conditional[col][c]):
                    prob += np.log(conditional[col][c][instance[col]])
            else:
                mean, std = conditional[col][c]
                prob += np.log(norm.pdf(instance[col], mean, std))
        if prob + np.log(prior[c]) > best_prob:
            best_prob = prob + np.log(prior[c])
            prediction = c
    return prediction
```

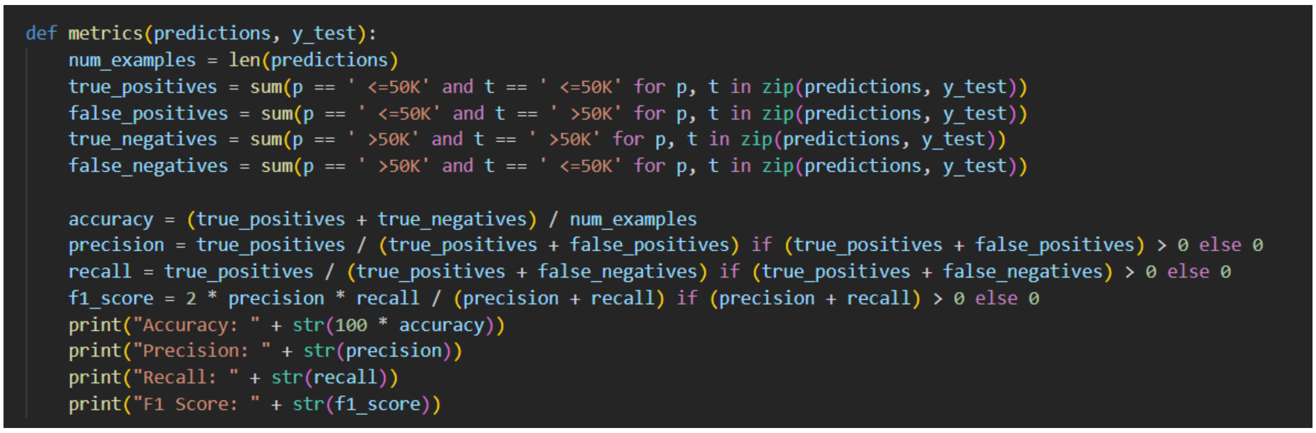
The given sub-task required us to Implement a function to calculate the accuracy of the Naive Bayes classifier on the testing set. In this sub-task, we called the prediction function for each testing example in the dataset. The predictions are stored in the predictions list and will be later used to calculate the accuracies and other performance metrics.



```python
def evaluate(X_test, conditional, prior):
    predictions = []
    for i in range(len(X_test)):
        pred = predict(X_test.iloc[i], conditional, y_train, prior)
        predictions.append(pred)
    return predictions
```

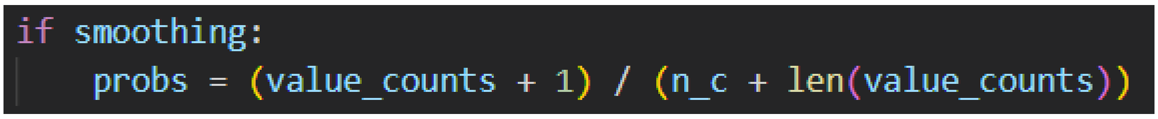
Task-3: Evaluation and Improvement

The given subtask required us to evaluate the performance of the Naive Bayes classifier using accuracy, precision, recall, and F1-score.



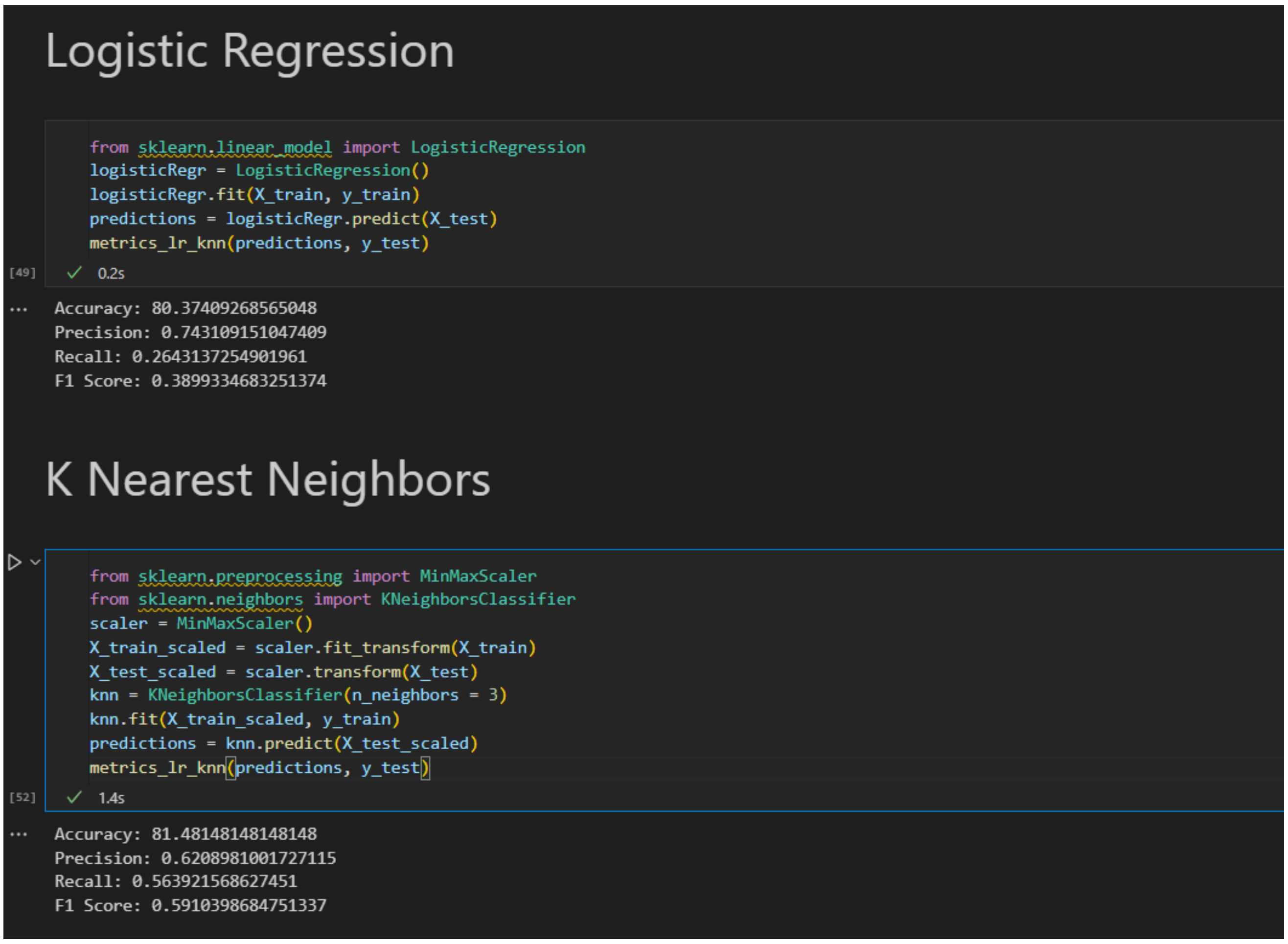
```python
def metrics(predictions, y_test):
    num_examples = len(predictions)
    true_positives = sum(p == ' <=50K' and t == ' <=50K' for p, t in zip(predictions, y_test))
    false_positives = sum(p == ' <=50K' and t == ' >50K' for p, t in zip(predictions, y_test))
    true_negatives = sum(p == ' >50K' and t == ' >50K' for p, t in zip(predictions, y_test))
    false_negatives = sum(p == ' >50K' and t == ' <=50K' for p, t in zip(predictions, y_test))

    accuracy = (true_positives + true_negatives) / num_examples
    precision = true_positives / (true_positives + false_positives) if (true_positives + false_positives) > 0 else 0
    recall = true_positives / (true_positives + false_negatives) if (true_positives + false_negatives) > 0 else 0
    f1_score = 2 * precision * recall / (precision + recall) if (precision + recall) > 0 else 0
    print("Accuracy: " + str(100 * accuracy))
    print("Precision: " + str(precision))
    print("Recall: " + str(recall))
    print("F1 Score: " + str(f1_score))
```

The given subtask required us to experiment with different smoothing techniques to improve the performance of our classifier. The smoothing technique we used was Laplace's smoothing.



```python
if smoothing:
    probs = (value_counts + 1) / (n_c + len(value_counts))
```

The given subtask required us to compare the performance of the Naive Bayes classifier with logistic regression and k-nearest neighbors. We used sklearn to train and find the performance metrics using logistic regression and k-nearest neighbors.



Logistic Regression

```
from sklearn.linear_model import LogisticRegression
logisticRegr = LogisticRegression()
logisticRegr.fit(X_train, y_train)
predictions = logisticRegr.predict(X_test)
metrics_lr_knn(predictions, y_test)
```

[49] ✓ 0.2s

```
Accuracy: 80.37409268565048
Precision: 0.743109151047409
Recall: 0.2643137254901961
F1 Score: 0.3899334683251374
```

K Nearest Neighbors

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
knn = KNeighborsClassifier(n_neighbors = 3)
knn.fit(X_train_scaled, y_train)
predictions = knn.predict(X_test_scaled)
metrics_lr_knn(predictions, y_test)
```

[52] ✓ 1.4s

```
Accuracy: 81.48148148148148
Precision: 0.6208981001727115
Recall: 0.563921568627451
F1 Score: 0.5910398684751337
```

Part B – Building a Basic Neural Network for Image Classification

The goal is to build a basic neural network that can classify images of handwritten digits from the MNIST dataset.

We have built 12 distinct artificial neural network classifiers by varying various following parameters -

(i). A number of hidden layers – 2 or 3

(ii) Total number of neurons in the hidden layer is 100 or 150

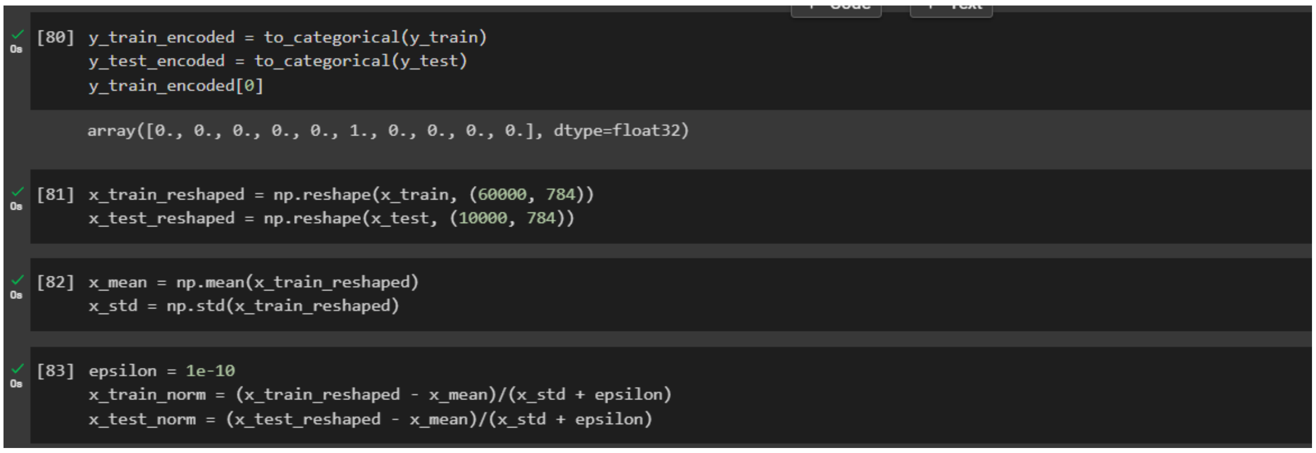
(iii) Activation function is from any of the following functions: tanh, sigmoid, ReLu

We ran the 12 different models on 2 different optimizers that is Adam and SGD.

Code Snippets for the model:

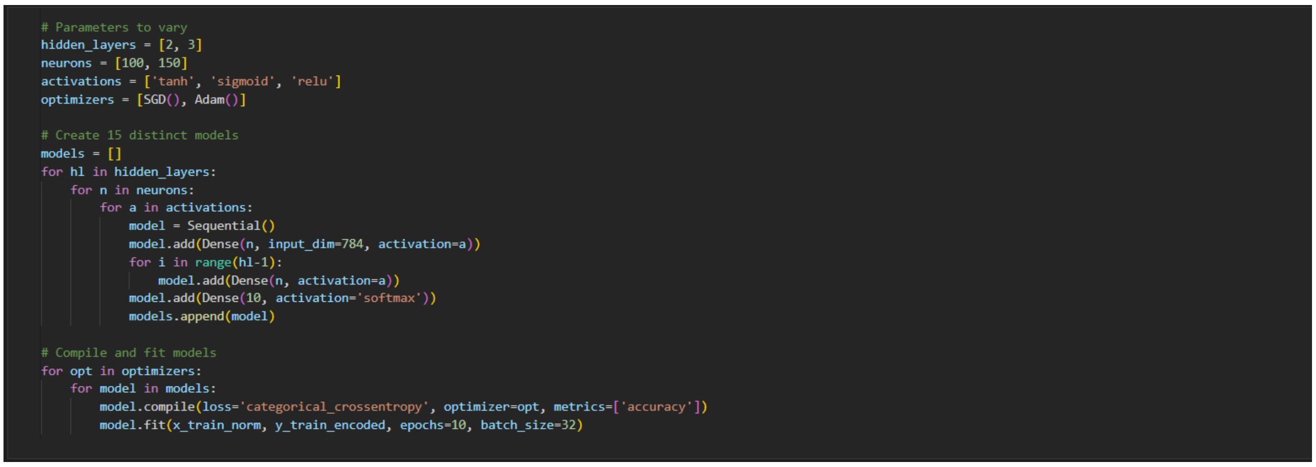
Training and Testing:

We have trained the neural network on the MNIST dataset and used Adam and SGD as an optimization algorithm. The different accuracy and confusion matrices of the best model are shown below.



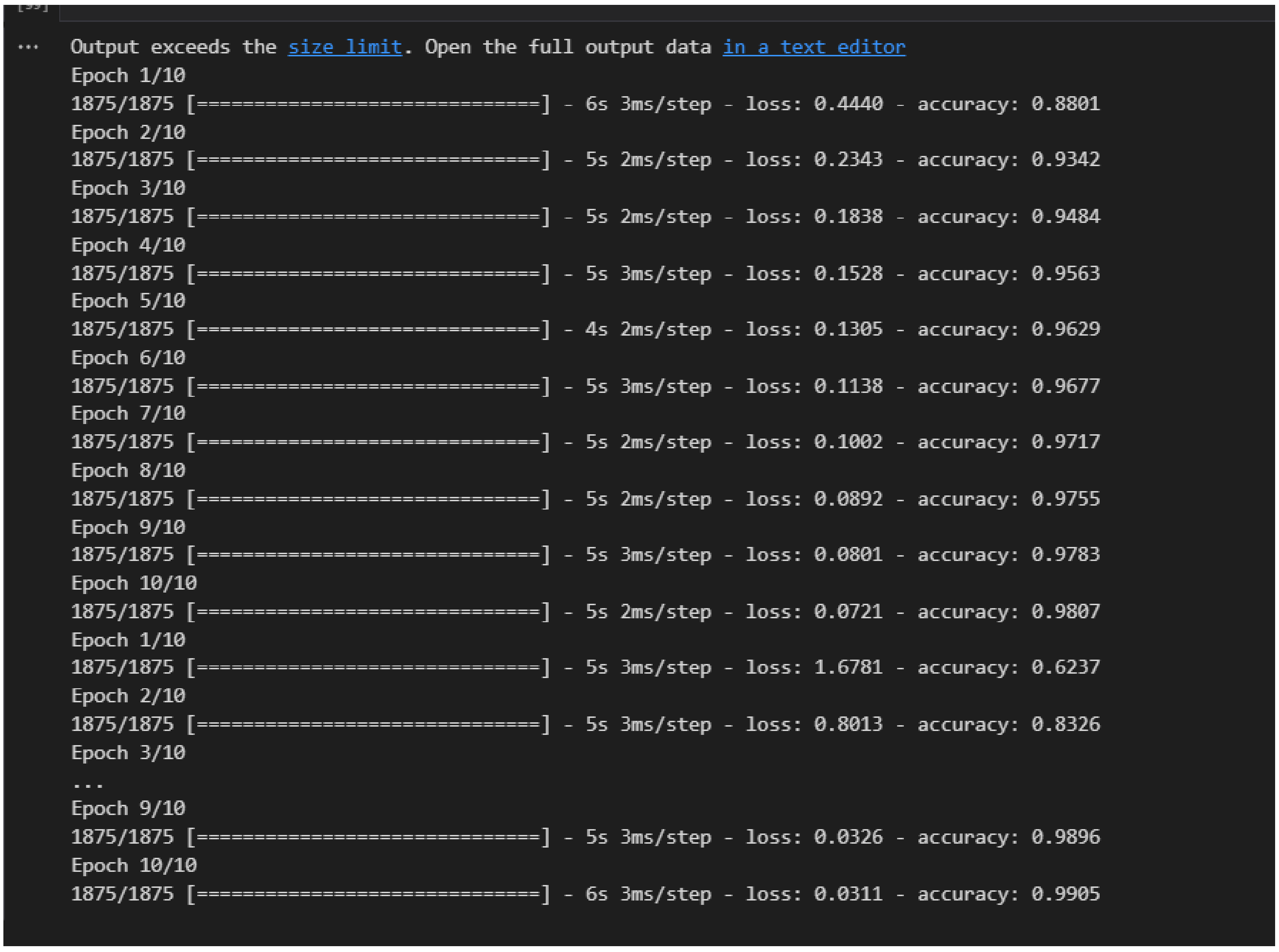
```
[80] y_train_encoded = to_categorical(y_train)
     y_test_encoded = to_categorical(y_test)
     y_train_encoded[0]

     array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)

[81] x_train_reshaped = np.reshape(x_train, (60000, 784))
     x_test_reshaped = np.reshape(x_test, (10000, 784))

[82] x_mean = np.mean(x_train_reshaped)
     x_std = np.std(x_train_reshaped)

[83] epsilon = 1e-10
     x_train_norm = (x_train_reshaped - x_mean)/(x_std + epsilon)
     x_test_norm = (x_test_reshaped - x_mean)/(x_std + epsilon)
```

Code for 12 distinct models



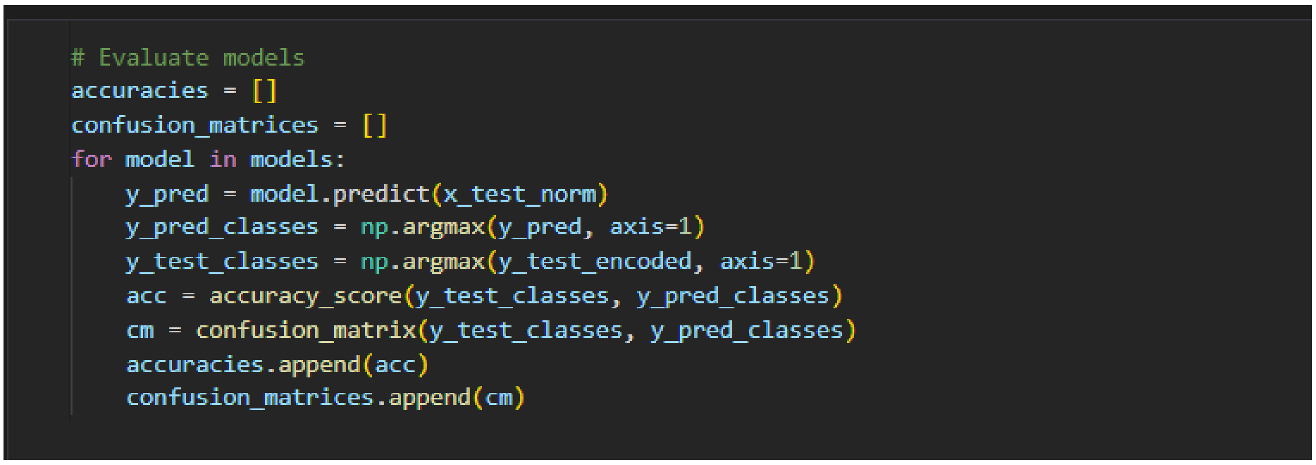
```python
# Parameters to vary
hidden_layers = [2, 3]
neurons = [100, 150]
activations = ['tanh', 'sigmoid', 'relu']
optimizers = [SGD(), Adam()]

# Create 15 distinct models
models = []
for hl in hidden_layers:
    for n in neurons:
        for a in activations:
            model = Sequential()
            model.add(Dense(n, input_dim=784, activation=a))
            for i in range(hl-1):
                model.add(Dense(n, activation=a))
            model.add(Dense(10, activation='softmax'))
            models.append(model)

# Compile and fit models
for opt in optimizers:
    for model in models:
        model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
        model.fit(x_train_norm, y_train_encoded, epochs=10, batch_size=32)
```

Output accuracies after running the above code.

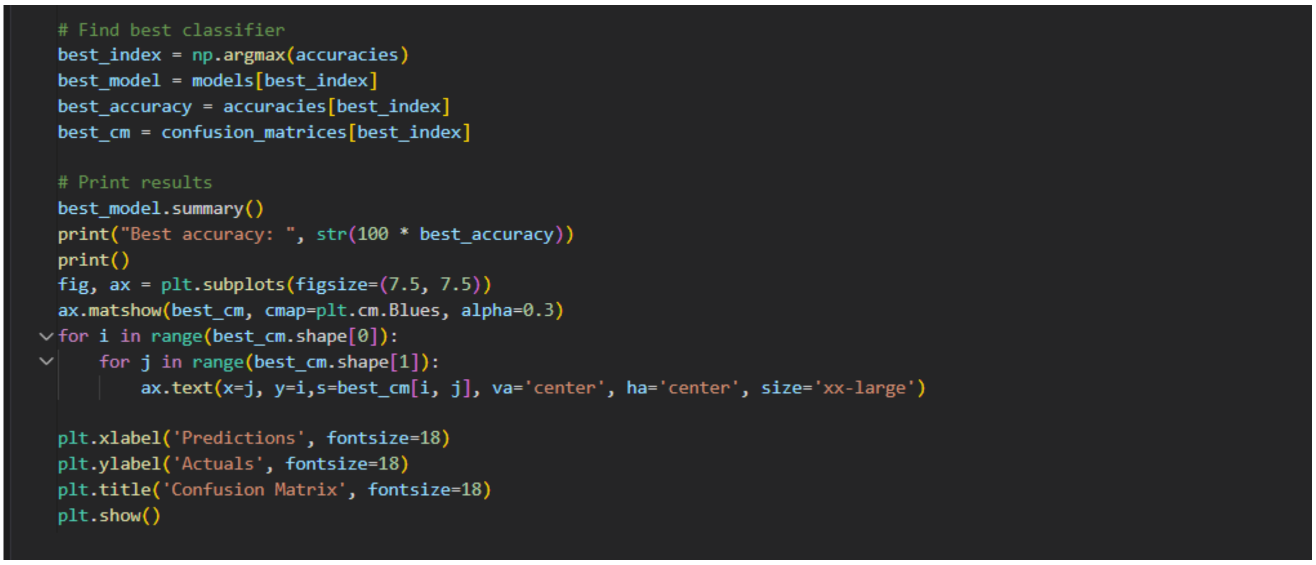


```
Output exceeds the size limit. Open the full output data in a text editor
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.4440 - accuracy: 0.8801
Epoch 2/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.2343 - accuracy: 0.9342
Epoch 3/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1838 - accuracy: 0.9484
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1528 - accuracy: 0.9563
Epoch 5/10
1875/1875 [==============================] - 4s 2ms/step - loss: 0.1305 - accuracy: 0.9629
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1138 - accuracy: 0.9677
Epoch 7/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1002 - accuracy: 0.9717
Epoch 8/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.0892 - accuracy: 0.9755
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0801 - accuracy: 0.9783
Epoch 10/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.0721 - accuracy: 0.9807
Epoch 1/10
1875/1875 [==============================] - 5s 3ms/step - loss: 1.6781 - accuracy: 0.6237
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.8013 - accuracy: 0.8326
Epoch 3/10
...
Epoch 9/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0326 - accuracy: 0.9896
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0311 - accuracy: 0.9905
```

Comparing the distinct models generated by their accuracies

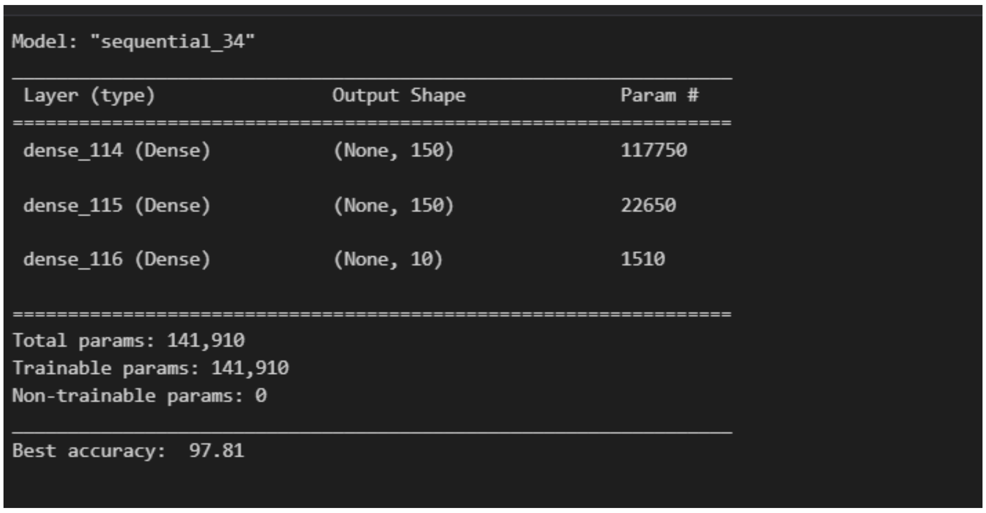


```python
# Evaluate models
accuracies = []
confusion_matrices = []
for model in models:
    y_pred = model.predict(x_test_norm)
    y_pred_classes = np.argmax(y_pred, axis=1)
    y_test_classes = np.argmax(y_test_encoded, axis=1)
    acc = accuracy_score(y_test_classes, y_pred_classes)
    cm = confusion_matrix(y_test_classes, y_pred_classes)
    accuracies.append(acc)
    confusion_matrices.append(cm)
```



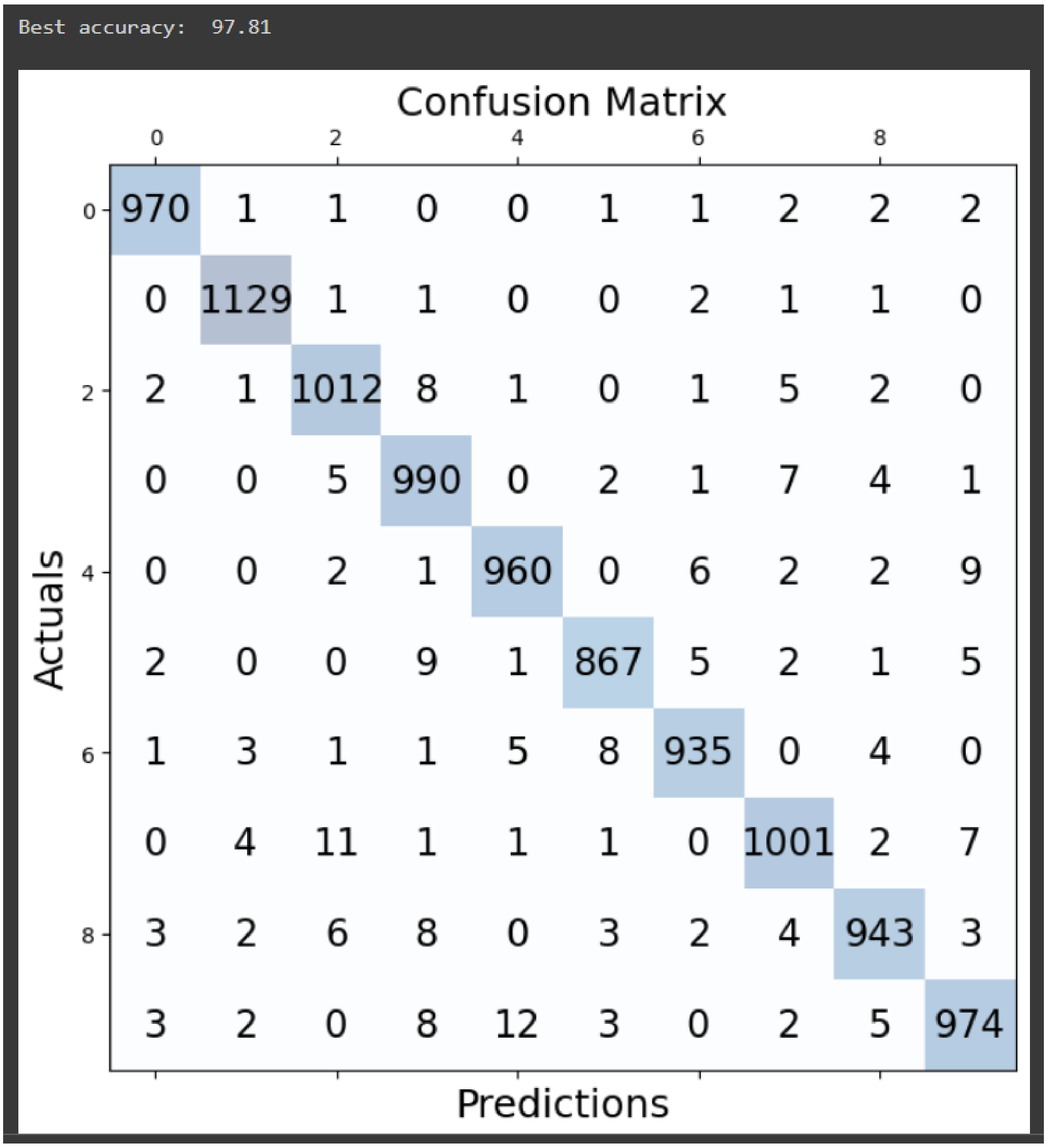
```python
# Find best classifier
best_index = np.argmax(accuracies)
best_model = models[best_index]
best_accuracy = accuracies[best_index]
best_cm = confusion_matrices[best_index]

# Print results
best_model.summary()
print("Best accuracy: ", str(100 * best_accuracy))
print()
fig, ax = plt.subplots(figsize=(7.5, 7.5))
ax.matshow(best_cm, cmap=plt.cm.Blues, alpha=0.3)
for i in range(best_cm.shape[0]):
    for j in range(best_cm.shape[1]):
        ax.text(x=j, y=i,s=best_cm[i, j], va='center', ha='center', size='xx-large')

plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
```

Results after running the above code snippet:-



```
Model: "sequential_34"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_114 (Dense)           (None, 150)               117750

 dense_115 (Dense)           (None, 150)               22650

 dense_116 (Dense)           (None, 10)                1510

=================================================================
Total params: 141,910
Trainable params: 141,910
Non-trainable params: 0
_________________________________________________________________
Best accuracy:  97.81
```

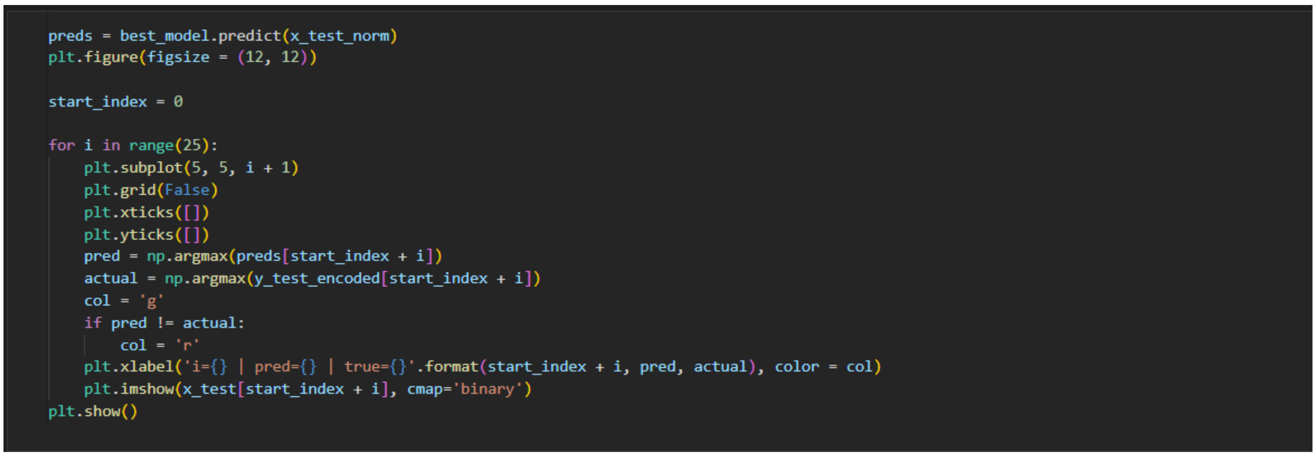
Best accuracy to be found out to be 97.81% .



```
Best accuracy:  97.81
```

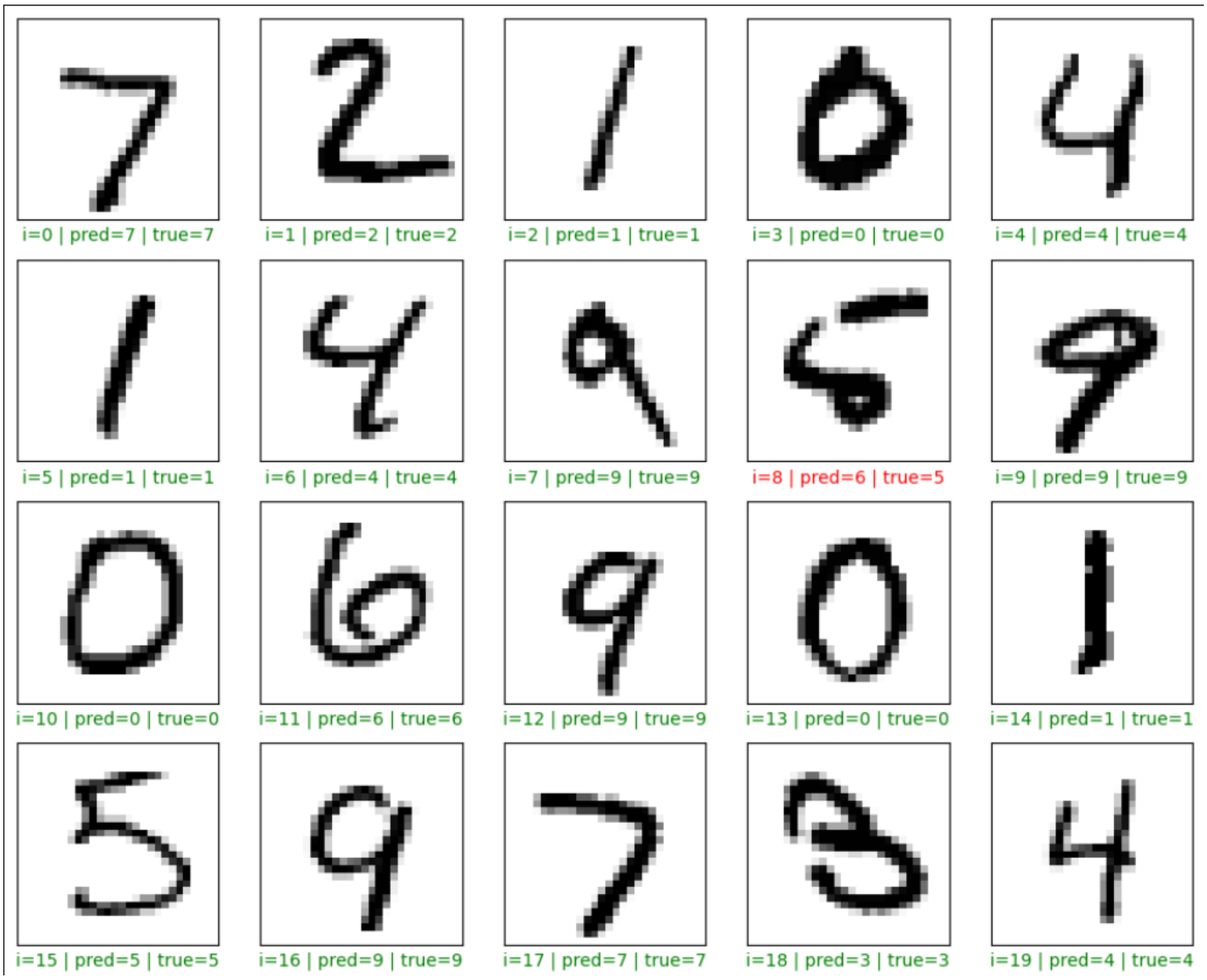
Actuals \ Predictions	0	1	2	3	4	5	6	7	8	9
0	970	1	1	0	0	1	1	2	2	2
1	0	1129	1	1	0	0	2	1	1	0
2	2	1	1012	8	1	0	1	5	2	0
3	0	0	5	990	0	2	1	7	4	1
4	0	0	2	1	960	0	6	2	2	9
5	2	0	0	9	1	867	5	2	1	5
6	1	3	1	1	5	8	935	0	4	0
7	0	4	11	1	1	1	0	1001	2	7
8	3	2	6	8	0	3	2	4	943	3
9	3	2	0	8	12	3	0	2	5	974

Running the test data set to predict the y_test with the best model



```python
preds = best_model.predict(x_test_norm)
plt.figure(figsize = (12, 12))

start_index = 0

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])
    pred = np.argmax(preds[start_index + i])
    actual = np.argmax(y_test_encoded[start_index + i])
    col = 'g'
    if pred != actual:
        col = 'r'
    plt.xlabel('i={} | pred={} | true={}'.format(start_index + i, pred, actual), color = col)
    plt.imshow(x_test[start_index + i], cmap='binary')
plt.show()
```

Output:


i=0 | pred=7 | true=7
i=1 | pred=2 | true=2
i=2 | pred=1 | true=1
i=3 | pred=0 | true=0
i=4 | pred=4 | true=4
i=5 | pred=1 | true=1
i=6 | pred=4 | true=4
i=7 | pred=9 | true=9
i=8 | pred=6 | true=5
i=9 | pred=9 | true=9
i=10 | pred=0 | true=0
i=11 | pred=6 | true=6
i=12 | pred=9 | true=9
i=13 | pred=0 | true=0
i=14 | pred=1 | true=1
i=15 | pred=5 | true=5
i=16 | pred=9 | true=9
i=17 | pred=7 | true=7
i=18 | pred=3 | true=3
i=19 | pred=4 | true=4