Notebook Link:
https://colab.research.google.com/drive/1rFPa8QgWvX14T9wMAs9KTQ1evAB0Qe7P?usp=sharing

Mete Kerem Berk

CS412 Machine Learning Homework 1

February 22, 2025

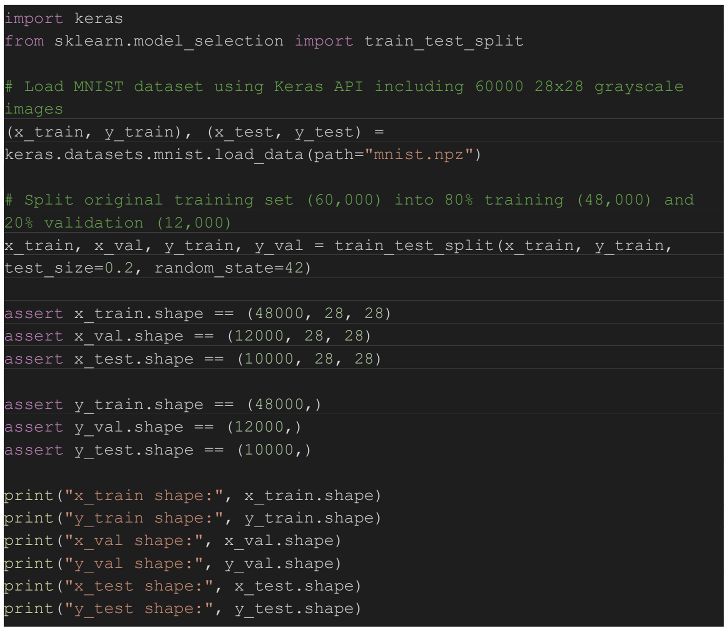
1. Overview

In this assignment, k-Nearest Neighbors (k-NN) and Decision Tree classifiers are implemented on the MNIST dataset. The MNIST dataset contains 28×28 grayscale images of handwritten digits (0-9), where each pixel value ranges from 0 to 255.

2. Dataset and Processing

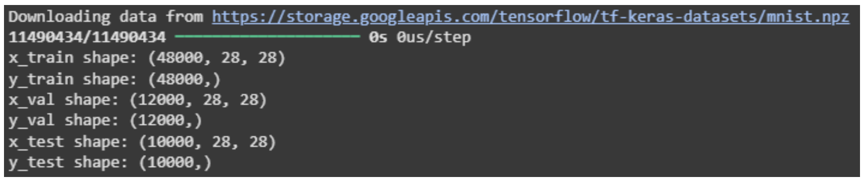
2.1. Data Loading

First, the MNIST dataset is loaded using the Keras API. The dataset is loaded as training and test sets. The initial training set is split between training and validation set 80/20 respectively and the test set is left as is.



```
import keras
from sklearn.model_selection import train_test_split

# Load MNIST dataset using Keras API including 60000 28x28 grayscale
images
(x_train, y_train), (x_test, y_test) =
keras.datasets.mnist.load_data(path="mnist.npz")

# Split original training set (60,000) into 80% training (48,000) and
20% validation (12,000)
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
test_size=0.2, random_state=42)

assert x_train.shape == (48000, 28, 28)
assert x_val.shape == (12000, 28, 28)
assert x_test.shape == (10000, 28, 28)

assert y_train.shape == (48000,)
assert y_val.shape == (12000,)
assert y_test.shape == (10000,)

print("x_train shape:", x_train.shape)
print("y_train shape:", y_train.shape)
print("x_val shape:", x_val.shape)
print("y_val shape:", y_val.shape)
print("x_test shape:", x_test.shape)
print("y_test shape:", y_test.shape)
```

Figure 1 - Data Loading Code



```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
x_train shape: (48000, 28, 28)
y_train shape: (48000,)
x_val shape: (12000, 28, 28)
y_val shape: (12000,)
x_test shape: (10000, 28, 28)
y_test shape: (10000,)
```

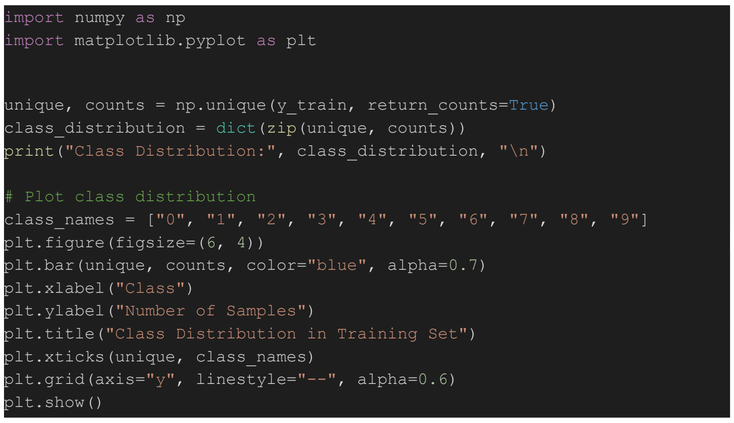
Figure 2 - Data Loading Output

The initial training set is split between the training set and the validation set to help in hyperparameter training and prevent overfitting. The validation set acts as a checkpoint to detect problems before testing.

2.2. Data Analysis

After loading the dataset, data analysis is done to identify dataset characteristics and necessary preprocessing steps.

First, class distribution is controlled. The number of samples per digit is computed and displayed to check for imbalances.



```python
import numpy as np
import matplotlib.pyplot as plt


unique, counts = np.unique(y_train, return_counts=True)
class_distribution = dict(zip(unique, counts))
print("Class Distribution:", class_distribution, "\n")

# Plot class distribution
class_names = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
plt.figure(figsize=(6, 4))
plt.bar(unique, counts, color="blue", alpha=0.7)
plt.xlabel("Class")
plt.ylabel("Number of Samples")
plt.title("Class Distribution in Training Set")
plt.xticks(unique, class_names)
plt.grid(axis="y", linestyle="--", alpha=0.6)
plt.show()
```

Figure 3 - Class Distribution Code

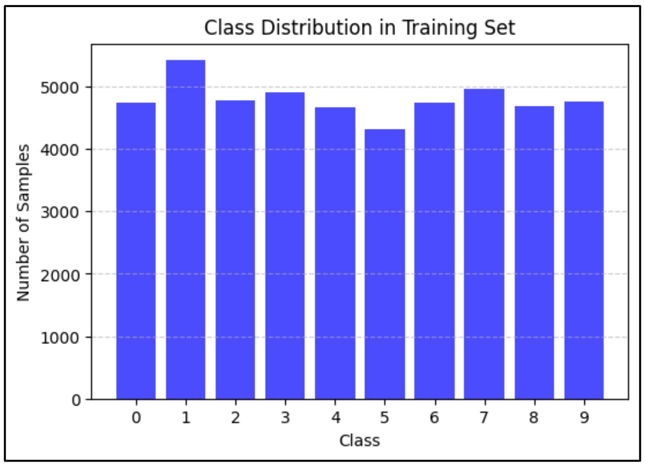
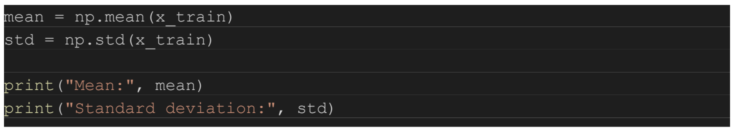


Figure 4 - Class Distribution

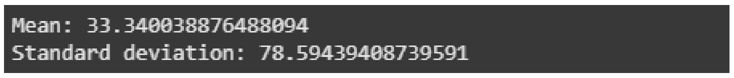
From Figure 4, it is seen that the samples are balanced in the dataset. This shows that preprocessing techniques like resampling or class weighting are not necessary.

Then, the mean and the standard deviation of the pixel values are computed. The mean helps understand the general brightness level of images and the standard deviation shows whether the images have high contrast or are mostly uniform in brightness.



```
mean = np.mean(x_train)
std = np.std(x_train)

print("Mean:", mean)
print("Standard deviation:", std)
```

Figure 5 - Mean & Standard Deviation Code

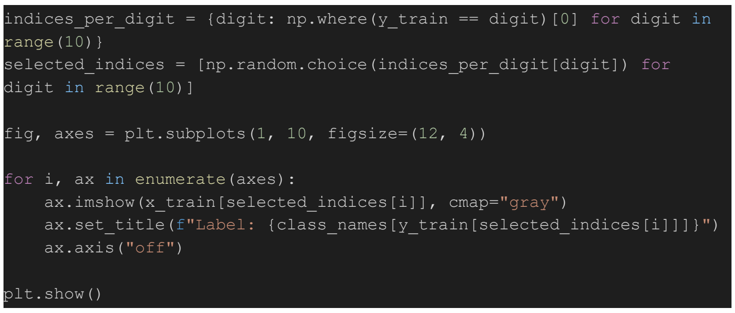


```
Mean: 33.340038876488094
Standard deviation: 78.59439408739591
```

Figure 6 - Mean & Standard Deviation

The mean and standard deviation values are moderate on a 0-255 scale. These values will need to be normalized (0-1 scale) in the preprocessing steps.

Finally, random samples from each digit are displayed to confirm data quality and structure. It also helps detect noise, mislabeling and unexpected artifacts.



```
indices_per_digit = {digit: np.where(y_train == digit)[0] for digit in
range(10)}
selected_indices = [np.random.choice(indices_per_digit[digit]) for
digit in range(10)]

fig, axes = plt.subplots(1, 10, figsize=(12, 4))

for i, ax in enumerate(axes):
    ax.imshow(x_train[selected_indices[i]], cmap="gray")
    ax.set_title(f"Label: {class_names[y_train[selected_indices[i]]]}")
    ax.axis("off")

plt.show()
```

Figure 7 - Visualization Code

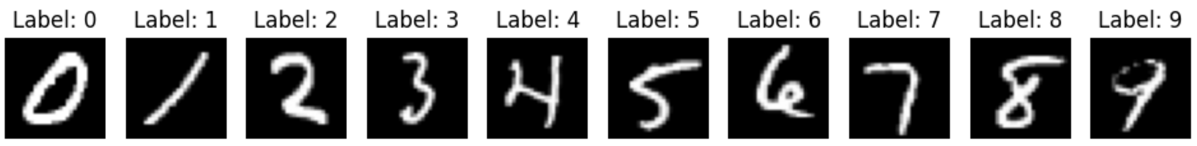
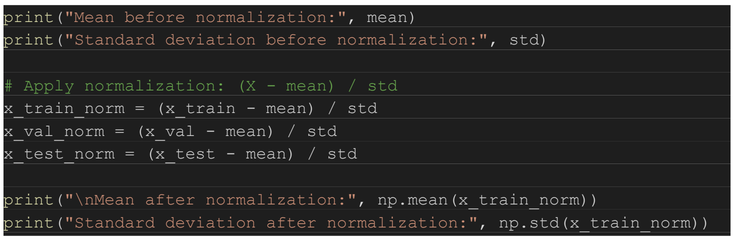


Figure 8 - Random Samples

2.3. Data Preprocessing

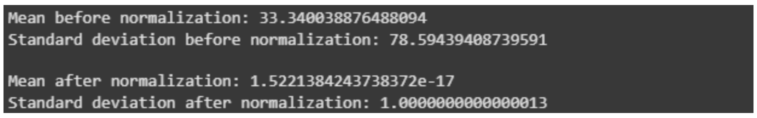
In the data preprocessing step, the images are normalized so the pixel values are scaled to the range [0,1]. To do this, Z-score normalization is used. This process standardizes the dataset so that the pixel values have a mean of 0 and a standard deviation of 1.

$$X_{normalized} = \frac{X - mean}{std}$$



```
print("Mean before normalization:", mean)
print("Standard deviation before normalization:", std)

# Apply normalization: (X - mean) / std
x_train_norm = (x_train - mean) / std
x_val_norm = (x_val - mean) / std
x_test_norm = (x_test - mean) / std

print("\nMean after normalization:", np.mean(x_train_norm))
print("Standard deviation after normalization:", np.std(x_train_norm))
```

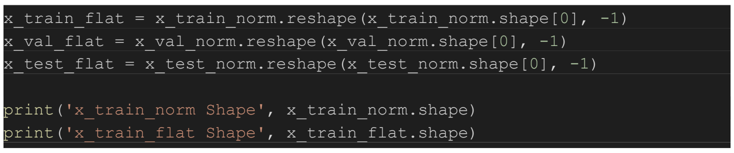
Figure 9 - Standardization Code



```
Mean before normalization: 33.340038876488094
Standard deviation before normalization: 78.59439408739591

Mean after normalization: 1.5221384243738372e-17
Standard deviation after normalization: 1.0000000000000013
```

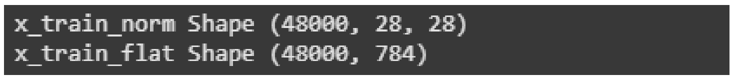
Figure 10 - Standardization Results

Then, the dataset needs to be reshaped because k-NN classifiers works with 2D inputted data. Currently, the shape of the dataset is **(number of samples, height, width)** but this shape does not work for k-NN as it is 3D. The shape will be changed to **(number of samples, height * width)** which is 2D.



```
x_train_flat = x_train_norm.reshape(x_train_norm.shape[0], -1)
x_val_flat = x_val_norm.reshape(x_val_norm.shape[0], -1)
x_test_flat = x_test_norm.reshape(x_test_norm.shape[0], -1)

print('x_train_norm Shape', x_train_norm.shape)
print('x_train_flat Shape', x_train_flat.shape)
```

Figure 11 - Reshaping Code

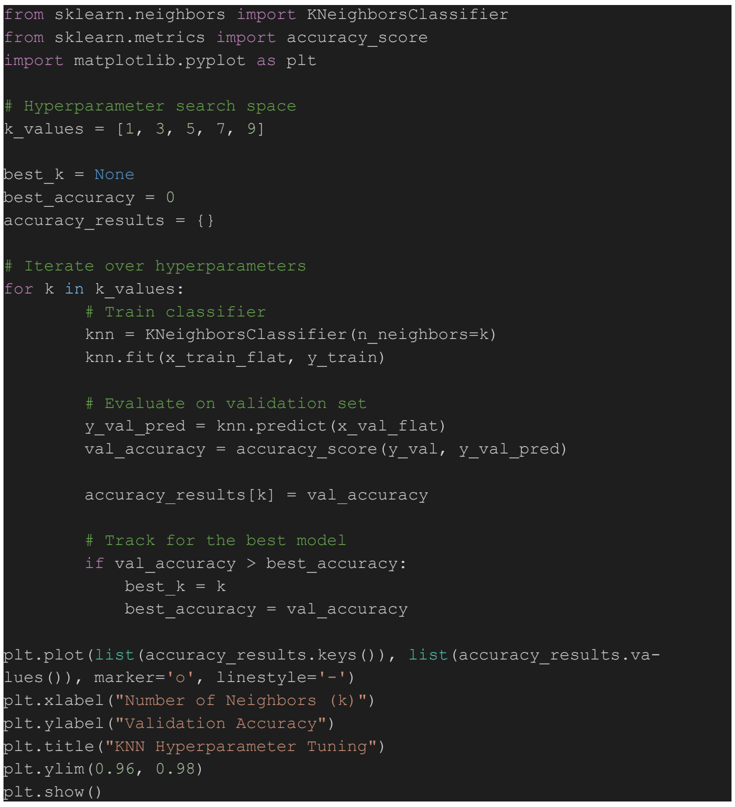


```
x_train_norm Shape (48000, 28, 28)
x_train_flat Shape (48000, 784)
```

Figure 12 - Reshaping Results

3. k-NN Classifier
3.1. Model Initialization and Hyperparameter Tuning

The k-NN classifier model is initialized and is experimented for the k values of [1, 3, 5, 7, 9]. During the experimentation, the validation set is used to determine the accuracy of the classifier.



```python
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# Hyperparameter search space
k_values = [1, 3, 5, 7, 9]

best_k = None
best_accuracy = 0
accuracy_results = {}

# Iterate over hyperparameters
for k in k_values:
        # Train classifier
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(x_train_flat, y_train)

        # Evaluate on validation set
        y_val_pred = knn.predict(x_val_flat)
        val_accuracy = accuracy_score(y_val, y_val_pred)

        accuracy_results[k] = val_accuracy

        # Track for the best model
        if val_accuracy > best_accuracy:
            best_k = k
            best_accuracy = val_accuracy

plt.plot(list(accuracy_results.keys()), list(accuracy_results.va-
lues()), marker='o', linestyle='-')
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Validation Accuracy")
plt.title("KNN Hyperparameter Tuning")
plt.ylim(0.96, 0.98)
plt.show()
```

Figure 13 - k-NN Initializer and k-Value Tester

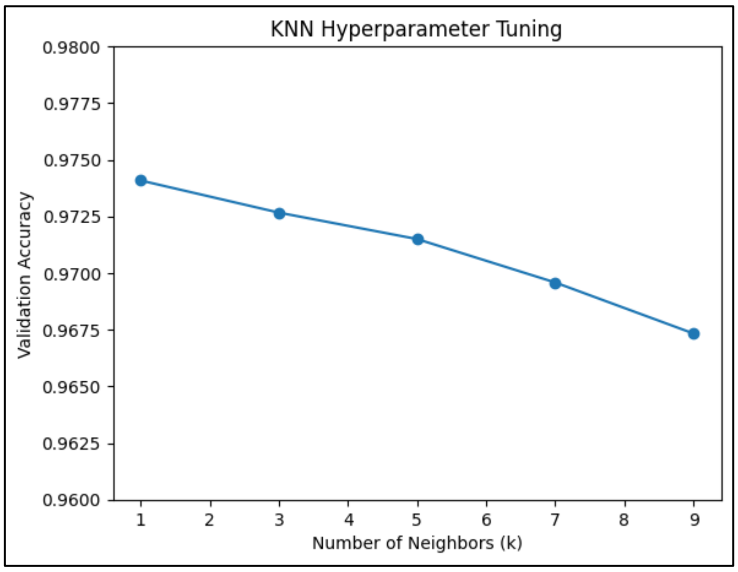
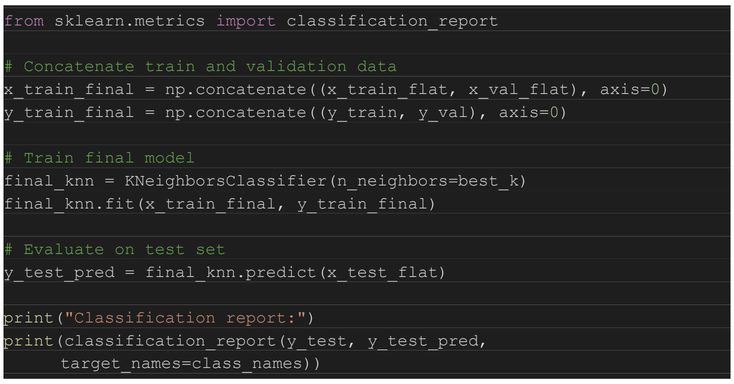


Figure 14 - Accuracy of Different k Values

The best value for k is found to be 1 with a validation accuracy of 97.41%. This can be due to the dataset being well separated or dataset not being noisy. However, a low k value will result in a higher sensitivity to noise and overfitting.

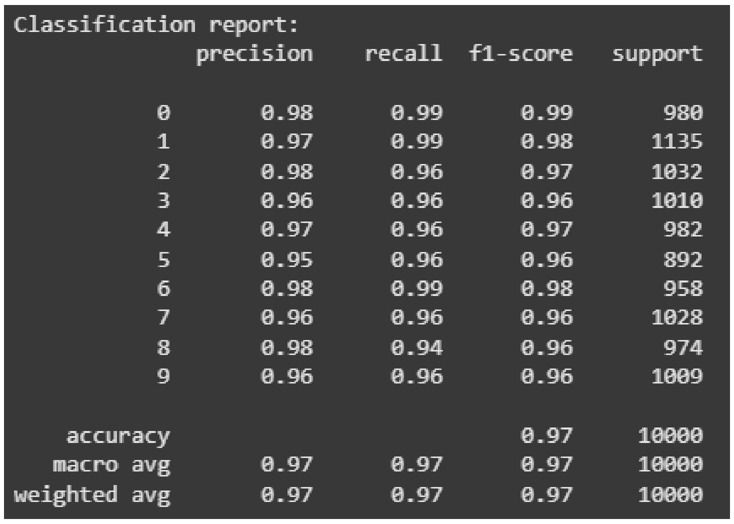
3.2. Final Model Training and Evaluation

For the final model, the training and validation sets are combined to retrain the k-NN classifier. Then, the final model is evaluated on the test set by reporting accuracy, precision, recall and F1-score.



```
from sklearn.metrics import classification_report

# Concatenate train and validation data
x_train_final = np.concatenate((x_train_flat, x_val_flat), axis=0)
y_train_final = np.concatenate((y_train, y_val), axis=0)

# Train final model
final_knn = KNeighborsClassifier(n_neighbors=best_k)
final_knn.fit(x_train_final, y_train_final)

# Evaluate on test set
y_test_pred = final_knn.predict(x_test_flat)

print("Classification report:")
print(classification_report(y_test, y_test_pred,
      target_names=class_names))
```

Figure 15 - Final k-NN and Evaluation Code

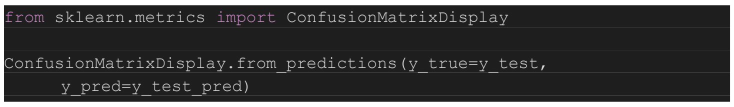


```
Classification report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       980
           1       0.97      0.99      0.98      1135
           2       0.98      0.96      0.97      1032
           3       0.96      0.96      0.96      1010
           4       0.97      0.96      0.97       982
           5       0.95      0.96      0.96       892
           6       0.98      0.99      0.98       958
           7       0.96      0.96      0.96      1028
           8       0.98      0.94      0.96       974
           9       0.96      0.96      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

Figure 16 - Classification Report

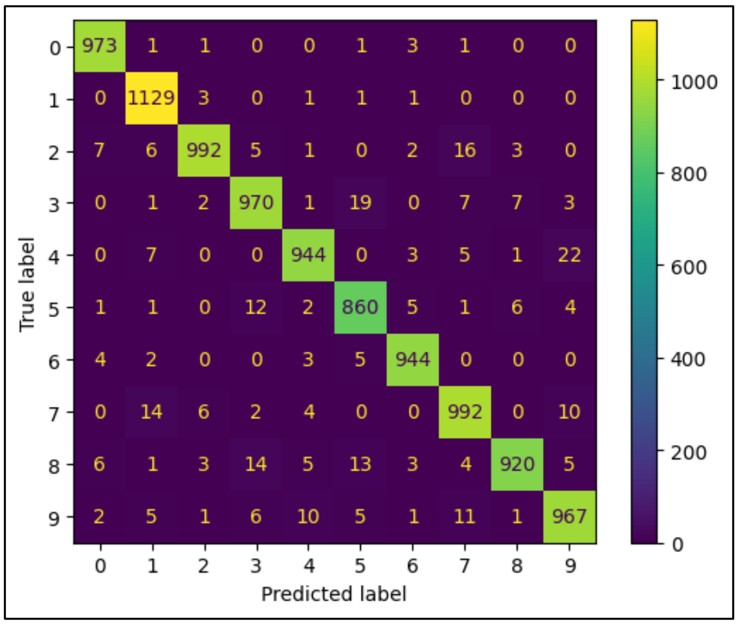
The accuracy score from the test set is 97% which matches the validation accuracy of 97.41%. Each of the digits have remarkably high precision and recall values which means fewer false positives and false negatives, respectively.

Then, the confusion matrix is created to determine which digits are being confused with each other.



```
from sklearn.metrics import ConfusionMatrixDisplay

ConfusionMatrixDisplay.from_predictions(y_true=y_test,
      y_pred=y_test_pred)
```

Figure 17 - Confusion Matrix Code



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	973	1	1	0	0	1	3	1	0	0
1	0	1129	3	0	1	1	1	0	0	0
2	7	6	992	5	1	0	2	16	3	0
3	0	1	2	970	1	19	0	7	7	3
4	0	7	0	0	944	0	3	5	1	22
5	1	1	0	12	2	860	5	1	6	4
6	4	2	0	0	3	5	944	0	0	0
7	0	14	6	2	4	0	0	992	0	10
8	6	1	3	14	5	13	3	4	920	5
9	2	5	1	6	10	5	1	11	1	967

Figure 18 - Confusion Matrix

From the confusion matrix, the misclassifications can be determined.

- 8 is misclassified as 3 and 5. This can be because of similar round shapes.
- 2 and 7 are confused with each other. This is likely because of their similar shapes.
- 3 and 5 are confused with each other. This can be because of similar round shapes.
- 4 and 9 are confused with each other. This is likely because of their similar shapes.

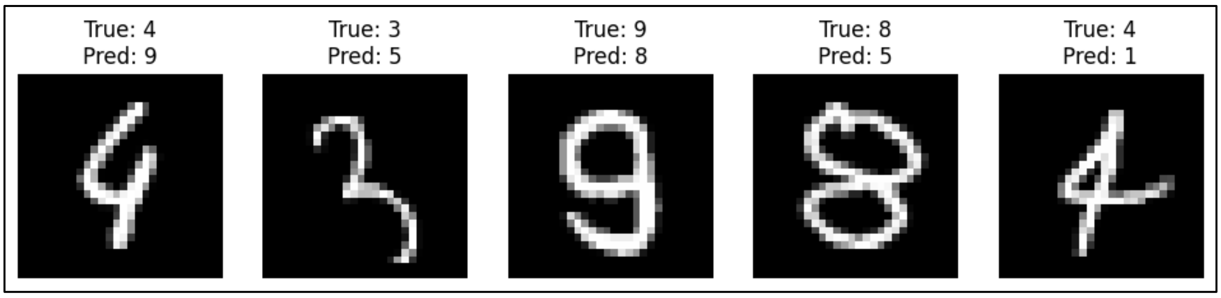


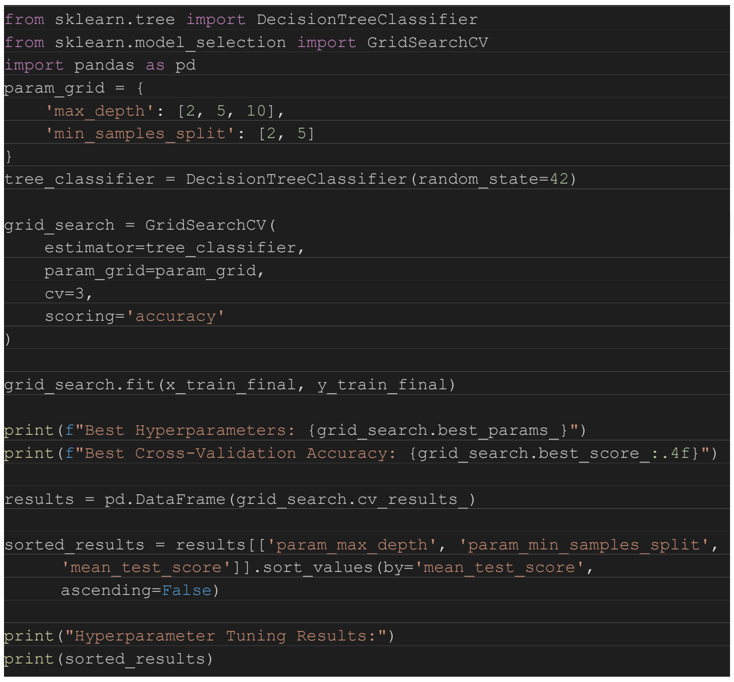
Figure 19 - Some Misclassifications

4. Decision Tree Classifier

4.1. Model Training and Hyperparameter Tuning

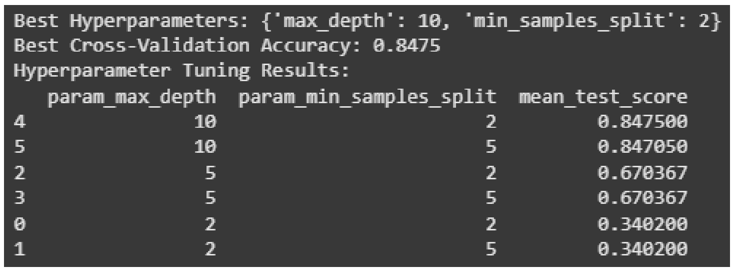
Next, a Decision Tree classifier is trained on the MNIST dataset. The hyperparameters are tuned in the following range:

- Max. depth: [2, 5, 10]
- Min. samples split: [2, 5]



```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV
import pandas as pd
param_grid = {
    'max_depth': [2, 5, 10],
    'min_samples_split': [2, 5]
}
tree_classifier = DecisionTreeClassifier(random_state=42)

grid_search = GridSearchCV(
    estimator=tree_classifier,
    param_grid=param_grid,
    cv=3,
    scoring='accuracy'
)

grid_search.fit(x_train_final, y_train_final)

print(f"Best Hyperparameters: {grid_search.best_params_}")
print(f"Best Cross-Validation Accuracy: {grid_search.best_score_:.4f}")

results = pd.DataFrame(grid_search.cv_results_)

sorted_results = results[['param_max_depth', 'param_min_samples_split',
     'mean_test_score']].sort_values(by='mean_test_score',
     ascending=False)

print("Hyperparameter Tuning Results:")
print(sorted_results)
```

Figure 20 - Decision Tree Code



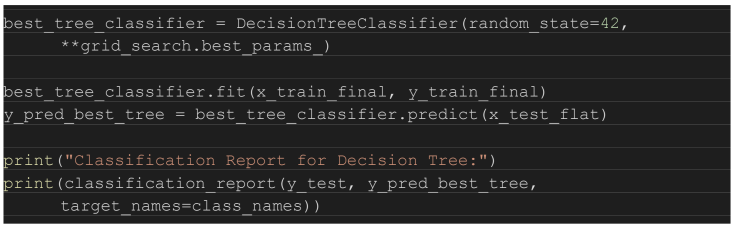
```
Best Hyperparameters: {'max_depth': 10, 'min_samples_split': 2}
Best Cross-Validation Accuracy: 0.8475
Hyperparameter Tuning Results:
   param_max_depth  param_min_samples_split  mean_test_score
4               10                        2         0.847500
5               10                        5         0.847050
2                5                        2         0.670367
3                5                        5         0.670367
0                2                        2         0.340200
1                2                        5         0.340200
```

Figure 21 - Decision Tree Hyperparameter Results

The results show that with a 'max_depth' of 10 and 'min_samples_split' of 2 results with the highest accuracy with 84.75%. The accuracy is proportional with the maximum depth of the tree. This makes sense as a deeper tree allows for a more detailed capture of the data. The minimum samples split is inversely proportional with accuracy. This is because a higher number of results in less overfitting.

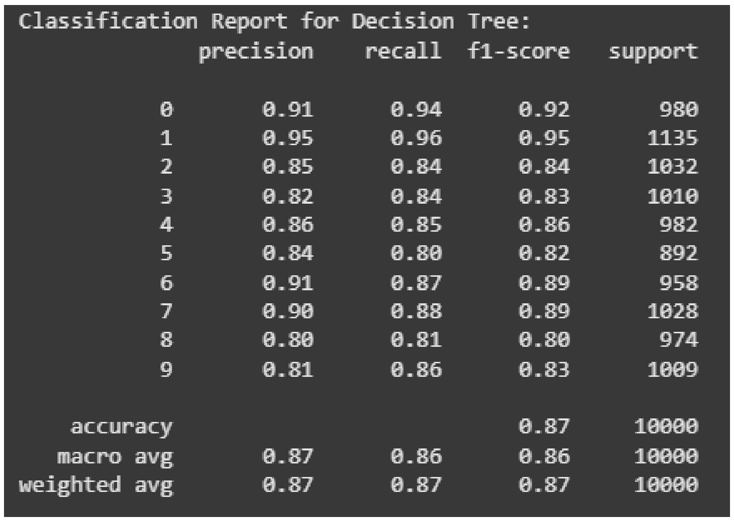
4.2. Evaluation

The Decision Tree classifier is evaluated in the same way as the k-NN classifier. A classification report is created to determine accuracy, precision, recall and F1-score.



```
best_tree_classifier = DecisionTreeClassifier(random_state=42,
      **grid_search.best_params_)

best_tree_classifier.fit(x_train_final, y_train_final)
y_pred_best_tree = best_tree_classifier.predict(x_test_flat)

print("Classification Report for Decision Tree:")
print(classification_report(y_test, y_pred_best_tree,
      target_names=class_names))
```

Figure 22 - Evaluation Code



```
Classification Report for Decision Tree:
              precision    recall  f1-score   support

           0       0.91      0.94      0.92       980
           1       0.95      0.96      0.95      1135
           2       0.85      0.84      0.84      1032
           3       0.82      0.84      0.83      1010
           4       0.86      0.85      0.86       982
           5       0.84      0.80      0.82       892
           6       0.91      0.87      0.89       958
           7       0.90      0.88      0.89      1028
           8       0.80      0.81      0.80       974
           9       0.81      0.86      0.83      1009

    accuracy                           0.87     10000
   macro avg       0.87      0.86      0.86     10000
weighted avg       0.87      0.87      0.87     10000
```

Figure 23 - Classification Report

The accuracy score from the test set is 87%. It performs best with digits 1 (precision: 0.95, recall: 0.96) and 0 (precision: 0.91, recall: 0.94). However, the model struggles with digits 5 (precision: 0.84, recall: 0.80) and 8 (precision: 0.80, recall: 0.81).

5. Conclusion

In summary, both the k Nearest Neighbors (k-NN) and Decision Tree models did well on the MNIST dataset with k-NN having an accuracy rate of 97% compared to the 87% of the Decision Tree. The k-NN classifier had high accuracy but can face overfitting issues with lower k values while the Decision Tree model encountered challenges with specific digits, especially those that had resemblances in their shapes.