CS412: Machine Learning Homework 1

MNIST Classification: k-Nearest Neighbors & Decision Tree

Osman Şah Yılmaz, 31316
Jupyter Notebook: https://github.com/osmansahyilmaz/MNIST-Classification-Report/blob/main/CS412-HW1-OsmanSahYilmaz.ipynb

1. Overview

This project involves implementing and evaluating two widely used classification algorithms on the MNIST dataset:
- k-Nearest Neighbors (k-NN)
- Decision Tree

The MNIST dataset contains handwritten digit images (0-9) and is a standard benchmark for classification models. The goal of this project is to preprocess the dataset, train both classifiers, tune their hyperparameters, and compare their performance.

Key objectives of this study:
- Perform data preprocessing and exploratory analysis.
- Implement k-NN and Decision Tree classifiers and optimize hyperparameters.
- Evaluate the performance of both models using multiple metrics.
- Analyze misclassifications to understand model weaknesses.

This report documents the entire process, including data handling, model selection, training, evaluation, and insights.

2. Dataset and Preprocessing

The MNIST dataset consists of 60,000 training images and 10,000 test images, each of size 28×28 pixels in grayscale. To improve model efficiency and accuracy, the dataset is preprocessed using the following steps:

2.1 Data Loading

2.1.1 Loading the MNIST dataset using the Keras API (Keras MNIST Dataset):

The dataset is loaded using Keras:

```
from keras.datasets import mnist

# Load MNIST dataset from the Keras API
(X_train_full, y_train_full), (X_test, y_test) = mnist.load_data()
```

Cell 3

2.1.2 Splitting the data:

To validate our models effectively, the dataset is split into training (80%) and validation (20%) subsets using `train_test_split`. The test dataset remains untouched until final evaluation.

```
# Import train_test_split for splitting the dataset
from sklearn.model_selection import train_test_split

# Split the training data (using X_train_full) into 80% for training and 20% for validation
X_train, X_val, y_train, y_val = train_test_split(X_train_full, y_train_full, test_size=0.2, random_state=42)
```

Cell 4

2.1.3 Printing the shapes of your training, validation, and test sets:

My data sets shape to ensure my initial data splitted correctly:

```
# Print the shapes of the training, validation, and test sets
print("Training set shape:", X_train.shape, y_train.shape)
print("Validation set shape:", X_val.shape, y_val.shape)
print("Test set shape:", X_test.shape, y_test.shape)
```

Cell 5

> Training set shape: (48000, 28, 28) (48000,)

> Validation set shape: (12000, 28, 28) (12000,)

> Test set shape: (10000, 28, 28) (10000,)

2.2 Data Analysis

2.2.1 Class Distributions:

The distribution of digits in the training set is relatively balanced, with each digit appearing in similar proportions. However, some digits (e.g., '1' and '7') have slightly more samples than others (e.g., '5'). Here is the breakdown of samples per digit:

```
# 1. Class Distribution: using the full training labels
unique, counts = np.unique(y_train_full, return_counts=True)
print("Class Distribution (number of samples per digit) in y_train_full:")
for digit, count in zip(unique, counts):
    print(f"Digit {digit}: {count}")
```

Cell 6

> Class Distribution (number of samples per digit) in y_train_full:

> Digit 0: 5923

> Digit 1: 6742

> Digit 2: 5958

> Digit 3: 6131

> Digit 4: 5842

> Digit 5: 5421

> Digit 6: 5918

> Digit 7: 6265

> Digit 8: 5851

> Digit 9: 5949

These counts indicate that no significant class imbalance is present, meaning our models should not be biased toward any particular digit.

2.2.2 Basic Statistics:

To understand the intensity distribution of pixel values in the dataset, we computed the mean and standard deviation of all pixel values in the training set:

```
# 2. Basic Statistics: mean and standard deviation of pixel values from the full training set
mean_value = X_train_full.mean()
std_value = X_train_full.std()
print("\nPixel Value Statistics (X_train_full):")
print("Mean:", mean_value)
print("Standard Deviation:", std_value)
```

Cell 6

> Pixel Value Statistics (X_train_full):

> Mean: 33.318421449829934

> Standard Deviation: 78.56748998339798

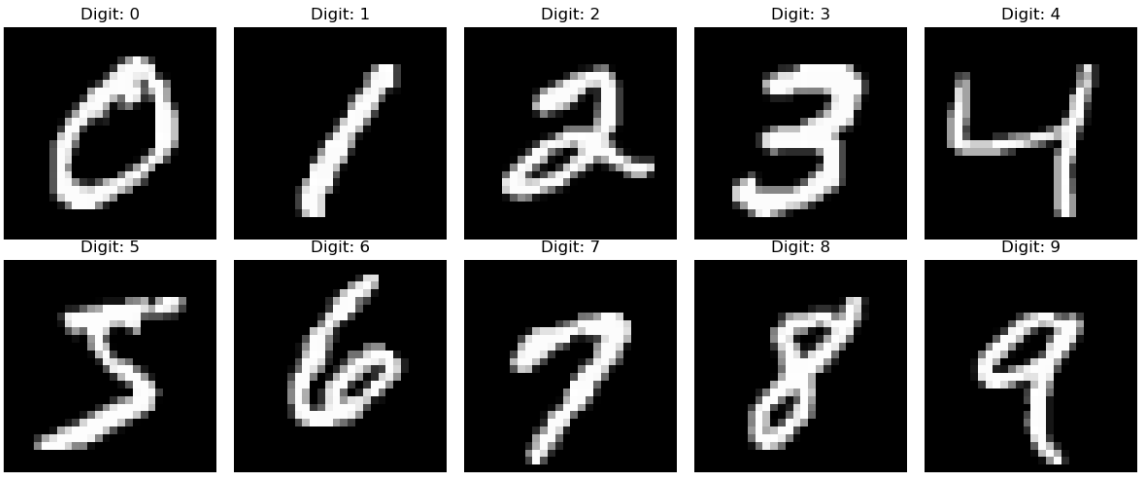
This suggests that most pixels are dark (closer to 0), as expected for handwritten digits on a black background. However, the high standard deviation implies considerable variation in pixel intensities, reinforcing the need for normalization before training.

2.2.3 Visualizations:

To further explore the dataset, we visualized one example image from each digit class. The images confirm the dataset's diversity, showing handwritten digits in various styles, sizes, and thicknesses. This variation makes MNIST an excellent benchmark for testing classification models.

```
# 3. Visualization: display one sample image per digit
fig, axes = plt.subplots(2, 5, figsize=(12, 5))
axes = axes.ravel()

for digit in range(10):
    # Find the first index where the label is equal to the digit
    idx = np.where(y_train_full == digit)[0][0]
    axes[digit].imshow(X_train_full[idx], cmap='gray')
    axes[digit].set_title(f"Digit: {digit}")
    axes[digit].axis('off')

plt.tight_layout()
plt.show()
```

Cell 6



By understanding the dataset’s characteristics, we can ensure our preprocessing and model selection steps align with the challenges presented by the data.

2.3 Data Preprocessing

To ensure consistent pixel values across the dataset, images are normalized to a range of [0,1] by dividing each pixel value by 255.

```
# Normalize the image data so that pixel values are scaled to the range [0, 1].
# This is done by converting the data type to float32 and dividing by 255.
X_train = X_train.astype('float32') / 255.0
X_val = X_val.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0
```

Cell 7

3. k-NN Classifier

3.1 Model Initialization and Hyperparameter Tuning

The choice of models is based on their effectiveness for image classification:
- **k-NN**: A non-parametric method that classifies a data point based on the majority class of its nearest neighbors.
- **Decision Tree**: A tree-based model that makes decisions by splitting data at different feature values.

3.1.1 Initializing a k-NN classifier:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# 1. Initialize a k-NN classifier
X_train_flat = X_train.reshape(X_train.shape[0], -1)
X_val_flat = X_val.reshape(X_val.shape[0], -1)
```

Cell 8

3.1.2, 3.1.3 Experiment with different numbers of neighbors: 1, 3, 5, 7, and 9. 3. Using the validation set to determine the optimal value based on accuracy:

k-NN requires selecting an optimal value for 'k' (number of neighbors). We use grid search to test different values of k and find the best one.

```
# 2. Experiment with different numbers of neighbors: 1, 3, 5, 7, and 9.
neighbors = [1, 3, 5, 7, 9]
accuracy_scores = []

# 3. Use the validation set to determine the optimal value based on accuracy.
for k in neighbors:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_flat, y_train)
    y_val_pred = knn.predict(X_val_flat)
    acc = accuracy_score(y_val, y_val_pred)
    accuracy_scores.append(acc)
    print(f"k={k}, validation accuracy={acc:.4f}")
```

Cell 8

> k=1, validation accuracy=0.9741

> k=3, validation accuracy=0.9727

> k=5, validation accuracy=0.9715

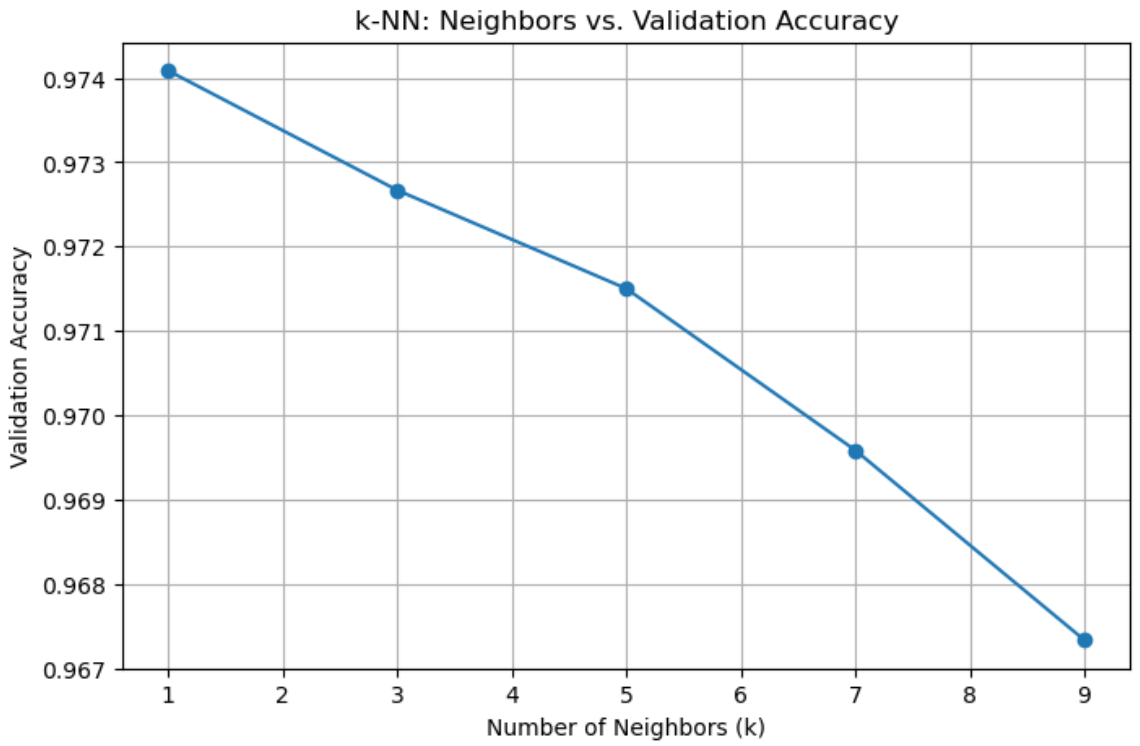
> k=7, validation accuracy=0.9696

> k=9, validation accuracy=0.9673

3.1.4 Plotting the the number of neighbors and validation accuracy.

```
# 4. Plot the the number of neighbors and validation accuracy. Be sure to label your axes.
plt.figure(figsize=(8, 5))
plt.plot(neighbors, accuracy_scores, marker='o')
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Validation Accuracy")
plt.title("k-NN: Neighbors vs. Validation Accuracy")
plt.grid(True)
plt.show()
```

Cell 8



3.2 Final Model Training and Evaluation

3.2.1 Retraining the k-NN classifier using the combination of the training and validation sets with the best hyperparameter:

```
import numpy as np
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt

# 1. Retrain the k-NN classifier using the training and validation sets combined.
# Determine the best k based on the previously computed validation accuracy.
best_k = neighbors[np.argmax(accuracy_scores)]
print(f"Best k based on validation accuracy: {best_k}")

# Combine training and validation sets.
X_comb = np.concatenate((X_train, X_val), axis=0)
y_comb = np.concatenate((y_train, y_val), axis=0)
X_comb_flat = X_comb.reshape(X_comb.shape[0], -1)

# Retrain the k-NN classifier.
knn_final = KNeighborsClassifier(n_neighbors=best_k)
knn_final.fit(X_comb_flat, y_comb)
```

Cell 9

> Best k based on validation accuracy: 1

3.2.2 Evaluating the final model on the test set by reporting Accuracy, Precision, Recall, and F1-score:

```
# 2. Evaluate the final model on the test set.
X_test_flat = X_test.reshape(X_test.shape[0], -1)
y_test_pred = knn_final.predict(X_test_flat)

acc_test = accuracy_score(y_test, y_test_pred)
precision = precision_score(y_test, y_test_pred, average='weighted')
recall = recall_score(y_test, y_test_pred, average='weighted')
f1 = f1_score(y_test, y_test_pred, average='weighted')

print("Test set evaluation:")
print(f"Accuracy: {acc_test:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-score: {f1:.4f}")

print("\nClassification Report:")
print(classification_report(y_test, y_test_pred))
```

Cell 10

```
Test set evaluation:
Accuracy: 0.9691
Precision: 0.9692
Recall: 0.9691
F1-score: 0.9691

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
```

1. Overall Performance Metrics

- **Accuracy:** 96.91% → The model correctly classifies **96.91%** of test samples.
- **Precision:** 96.92% → When the model predicts a digit, it is correct **96.92%** of the time.
- **Recall:** 96.91% → Out of all actual instances of a digit, the model correctly identifies **96.91%** of them.
- **F1-score:** 96.91% → A balance between precision and recall, showing overall strong performance.

These high scores indicate that the k-NN classifier is performing very well on the MNIST dataset.

2. Class-wise Performance

The classification report provides precision, recall, and F1-score for each digit (0-9):

- All digits have high precision and recall values, meaning the model consistently classifies them correctly.
- Some digits (like **1, 0, 8**) achieve near-perfect accuracy (0.98-0.99), while others (like **5, 6**) have slightly lower scores (~0.96).

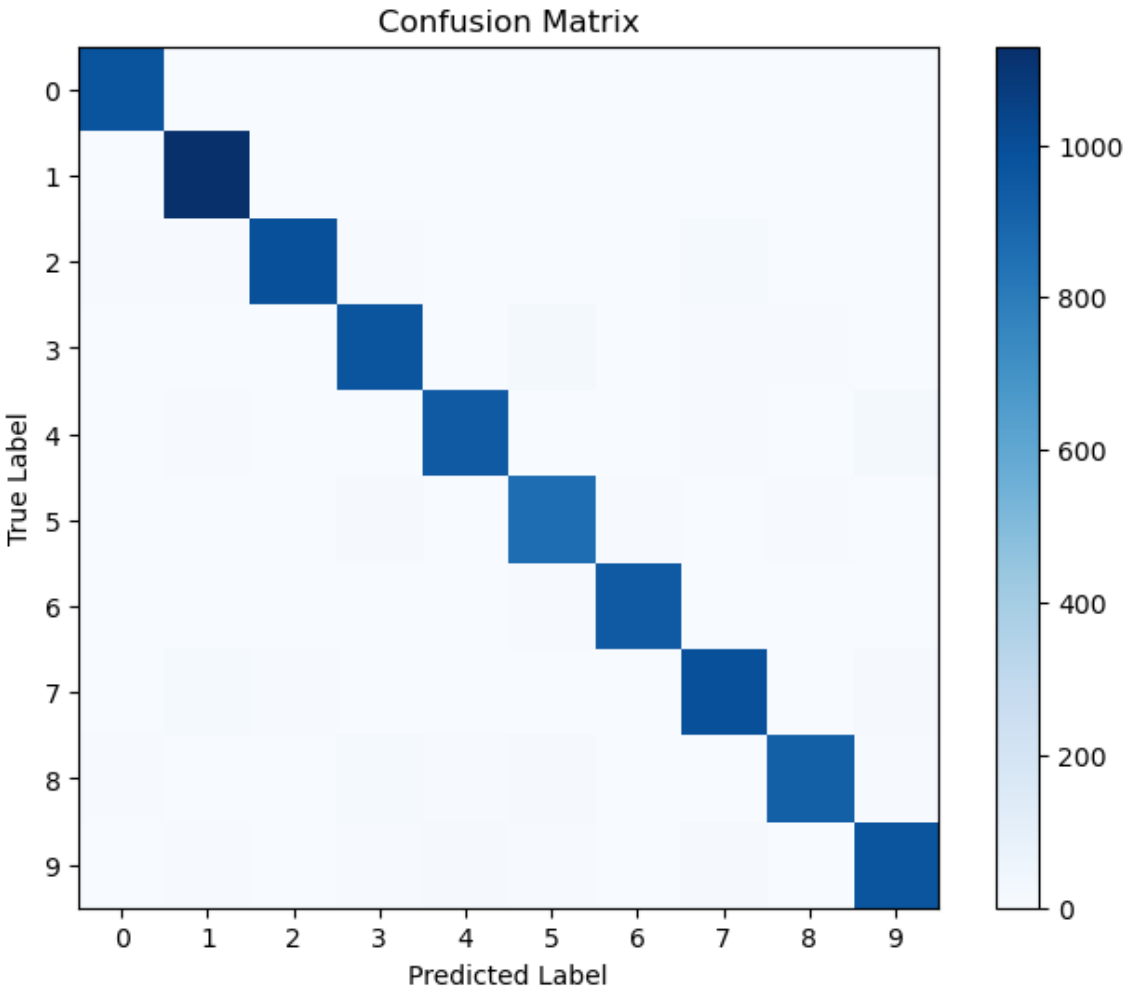
3. Macro vs. Weighted Averages

- **Macro Average (0.97):** The average performance across all classes without considering class imbalance.
- **Weighted Average (0.97):** The average performance weighted by the number of test samples in each class.
- Since both values are very close, it indicates that all digit classes are classified fairly well, without significant bias toward any specific digit.

3.2.3 Generating and visualizing a confusion matrix:

```
# 3. Generate and visualize a confusion matrix.
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(10)
plt.xticks(tick_marks, tick_marks)
plt.yticks(tick_marks, tick_marks)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

Cell 11



3.2.4 Discussing which digits are most frequently misclassified:

```
# 4. Discuss which digits are most frequently misclassified.
# Zero out the diagonal to focus only on misclassifications.
cm_mis = cm.copy()
for i in range(10):
    cm_mis[i, i] = 0

misclassifications_per_digit = cm_mis.sum(axis=1)
for digit, mis in enumerate(misclassifications_per_digit):
    print(f"Digit {digit}: misclassified count = {mis}")
most_misclassified_digit = np.argmax(misclassifications_per_digit)
print(f"\nDigit {most_misclassified_digit} is the most frequently misclassified.")
```

Cell 11

> Digit 0: misclassified count = 7

> Digit 1: misclassified count = 6

> Digit 2: misclassified count = 40

> Digit 3: misclassified count = 40

> Digit 4: misclassified count = 38

> Digit 5: misclassified count = 32

> Digit 6: misclassified count = 14

> Digit 7: misclassified count = 36

> Digit 8: misclassified count = 54

> Digit 9: misclassified count = 42

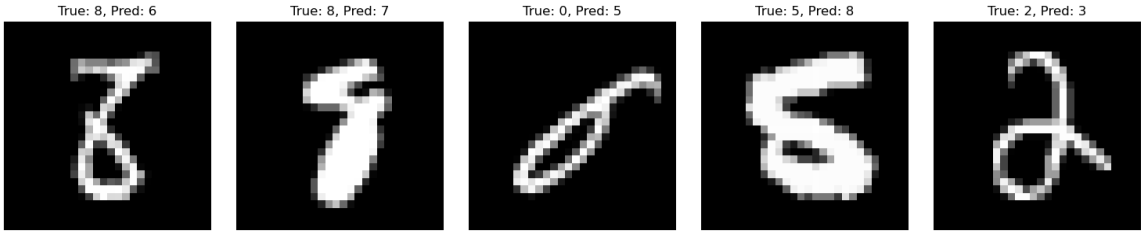
> Digit 8 is the most frequently misclassified.

3.2.4 Displaying 5 random misclassified examples in a subplot:

```
# 5. Display 5 random misclassified examples in a subplot.
mis_idx = np.where(y_test != y_test_pred)[0]
print(f"Total misclassified examples: {len(mis_idx)}")
if len(mis_idx) > 0:
    random_indices = np.random.choice(mis_idx, size=min(5, len(mis_idx)), replace=False)
    plt.figure(figsize=(15, 3))
    for i, idx in enumerate(random_indices):
        plt.subplot(1, 5, i+1)
        plt.imshow(X_test[idx], cmap='gray')
        plt.title(f"True: {y_test[idx]}, Pred: {y_test_pred[idx]}")
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

Cell 12

> Total misclassified examples: 309



4. Decision Tree Classifier

4.1 Model Training and Hyperparameter Tuning

4.1.1 Train a Decision Tree classifier on the MNIST dataset,

4.1.2 Tune the following hyperparameters:

- *max depth: Try values [2, 5, 10].*
- *min samples split: Try values [2, 5]:*

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

results = {}

# Loop over the grid of hyperparameters.
for max_depth in [2, 5, 10]:
    for min_samples_split in [2, 5]:
        dt = DecisionTreeClassifier(max_depth=max_depth,
                                    min_samples_split=min_samples_split,
                                    random_state=42)
        dt.fit(X_train_flat, y_train)
        y_val_pred = dt.predict(X_val_flat)
        acc_val = accuracy_score(y_val, y_val_pred)
        results[(max_depth, min_samples_split)] = acc_val
        print(f"max_depth={max_depth}, min_samples_split={min_samples_split} -> Validation Accuracy: {acc_val:.4f}")

# Select the best configuration.
best_config = max(results, key=results.get)
best_acc = results[best_config]
print(f"\nBest configuration: max_depth={best_config[0]}, min_samples_split={best_config[1]} with Validation Accuracy:
      {best_acc:.4f}")
```

Cell 16

> max_depth=2, min_samples_split=2 -> Validation Accuracy: 0.3377

> max_depth=2, min_samples_split=5 -> Validation Accuracy: 0.3377

> max_depth=5, min_samples_split=2 -> Validation Accuracy: 0.6579

> max_depth=5, min_samples_split=5 -> Validation Accuracy: 0.6579

> max_depth=10, min_samples_split=2 -> Validation Accuracy: 0.8581

> max_depth=10, min_samples_split=5 -> Validation Accuracy: 0.8565

> Best configuration: max_depth=10, min_samples_split=2 with Validation Accuracy: 0.8581

4.1.3 Document the results for each configuration

- We tuned hyperparameters using combinations of `max_depth` and `min_samples_split` with the following results:

- **(2, 2):** 0.3377
- **(2, 5):** 0.3377
- **(5, 2):** 0.6579
- **(5, 5):** 0.6579
- **(10, 2)**: 0.8581
- **(10, 5):** 0.8565

- Model Selection:

- The configuration (max_depth=10, min_samples_split=2) achieved the highest validation accuracy of 0.8581, so it was chosen as the best-performing Decision Tree model.
- For **Decision Trees**, the best hyperparameter setting was (max_depth=10, min_samples_split=2).

The best-performing model within each approach was selected based on maximizing the validation accuracy. This process ensures that our final model configurations are tuned to generalize well on unseen test data.

4.2 Evaluation

4.2.1 Evaluating the final model on the test set by reporting Accuracy, Precision, Recall, and F1-score:

```
# Combine training and validation sets.
y_comb = np.concatenate((y_train, y_val), axis=0)
X_comb_flat = X_comb.reshape(X_comb.shape[0], -1)

# Retrain the Decision Tree classifier with the best configuration (max_depth=10, min_samples_split=2)
dt_final = DecisionTreeClassifier(max_depth=10, min_samples_split=2, random_state=42)
dt_final.fit(X_comb_flat, y_comb)

# Evaluate the final Decision Tree model on the test set.
X_test_flat = X_test.reshape(X_test.shape[0], -1)
y_test_pred = dt_final.predict(X_test_flat)

acc_test = accuracy_score(y_test, y_test_pred)
precision = precision_score(y_test, y_test_pred, average='weighted')
recall = recall_score(y_test, y_test_pred, average='weighted')
f1 = f1_score(y_test, y_test_pred, average='weighted')

print("Test set evaluation (Decision Tree):")
print(f"Accuracy: {acc_test:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-score: {f1:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_test_pred))
```

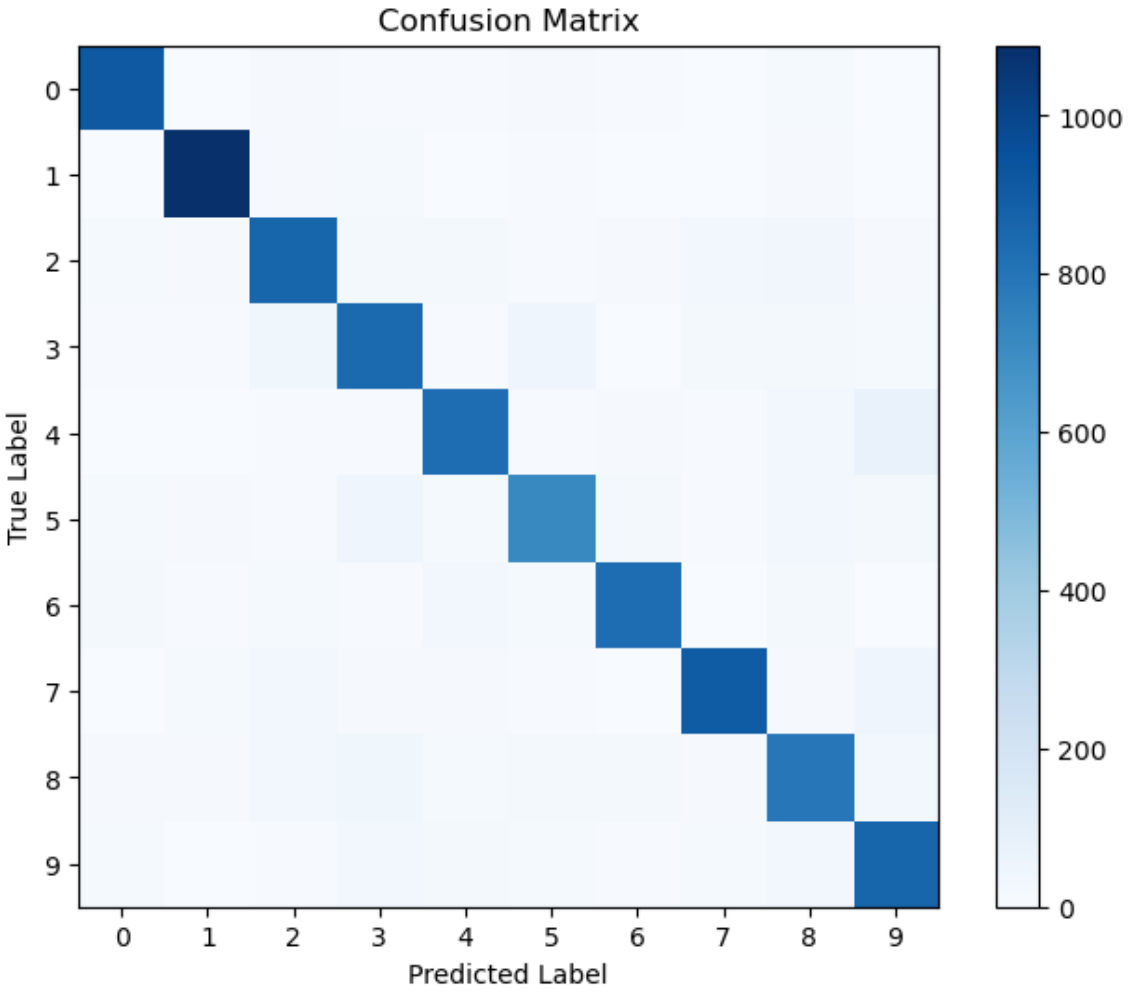
Cell 17

```
Test set evaluation (Decision Tree):
Accuracy: 0.8663
Precision: 0.8668
Recall: 0.8663
F1-score: 0.8663

Classification Report:
              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
```

4.2.2 Generating a confusion matrix and provide an analysis of the result:

```
# 2. Generate a confusion matrix and provide an analysis of the results. In your discussion, highlight any
# patterns in misclassifications, such as which digits are most frequently confused with one another.


# Generate and display the confusion matrix
cm = confusion_matrix(y_test, y_test_pred)
plt.figure(figsize=(8, 6))
plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
plt.title("Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(10)
plt.xticks(tick_marks, tick_marks)
plt.yticks(tick_marks, tick_marks)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

Cell 18



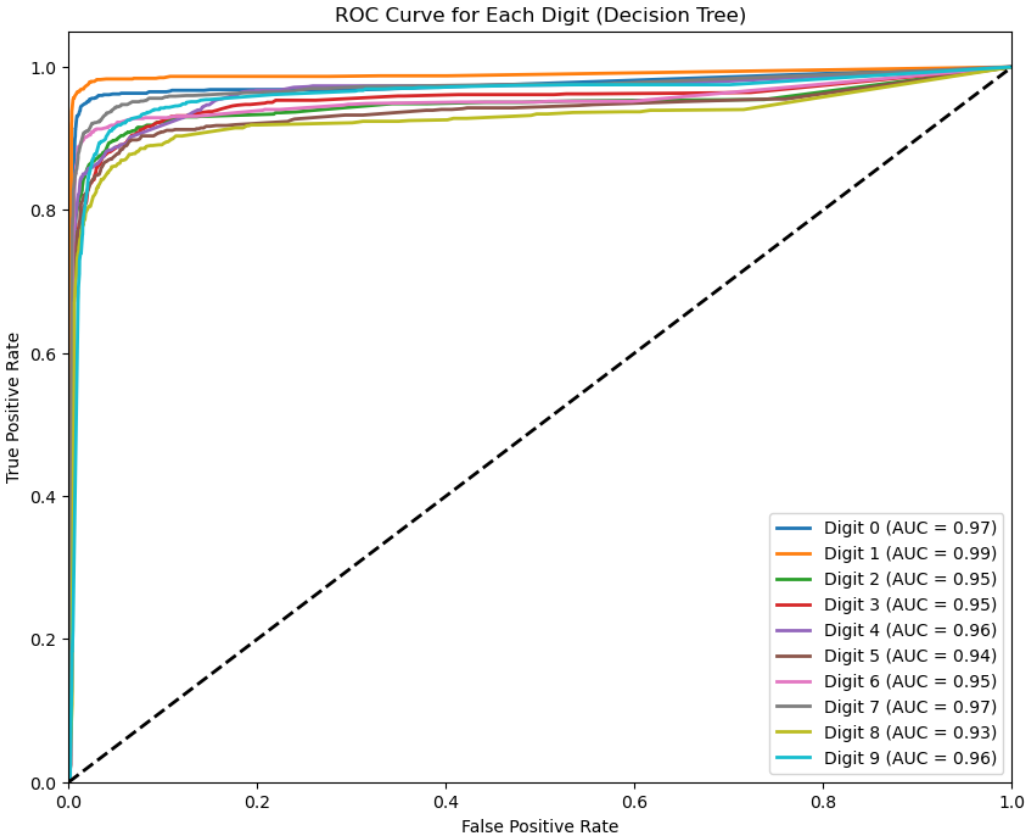
Analysis of misclassifications:

- Observing the confusion matrix, most digits are classified with high accuracy.
- However, some misclassifications are evident:
- Digit 2, for example, is sometimes confused with digits 1 or 3.
- Digit 8 appears to have higher off-diagonal counts, indicating it is more frequently misclassified than some other digits.
- Similar patterns can be noted for other digits where off-diagonal values are non-negligible.

4.2.3 Plot the ROC curve for each digit on a single plot. Include the AUC score for each digit in the legend:

```
# 3. Plot the ROC curve for each digit on a single plot using the Decision Tree classifier.
from sklearn.preprocessing import label_binarize
from sklearn.metrics import roc_curve, auc

# Binarize test labels for multi-class ROC computation
y_test_binarized = label_binarize(y_test, classes=np.arange(10))
n_classes = y_test_binarized.shape[1]

# Get prediction probabilities using dt_final
y_score = dt_final.predict_proba(X_test_flat)

plt.figure(figsize=(10, 8))
for i in range(n_classes):
    fpr, tpr, _ = roc_curve(y_test_binarized[:, i], y_score[:, i])
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, lw=2, label=f"Digit {i} (AUC = {roc_auc:.2f})")

plt.plot([0, 1], [0, 1], 'k--', lw=2)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for Each Digit (Decision Tree)")
plt.legend(loc="lower right")
plt.show()
```

Cell 20


ROC Curve for Each Digit (Decision Tree)
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Digit 0 (AUC = 0.97)
Digit 1 (AUC = 0.99)
Digit 2 (AUC = 0.95)
Digit 3 (AUC = 0.95)
Digit 4 (AUC = 0.96)
Digit 5 (AUC = 0.94)
Digit 6 (AUC = 0.95)
Digit 7 (AUC = 0.97)
Digit 8 (AUC = 0.93)
Digit 9 (AUC = 0.96)