Cs412 Machine Learning Homework 1 Report

MNIST Digit Classification using k-NN and Decision Tree

Jupyter notebook link

Merve Bilgi 29117

1. Introduction

For this homework, I implemented and evaluated two classifiers on the MNIST dataset:

- k-Nearest Neighbors (k-NN)

- Decision Tree Classifier

I did this to compare their performance in handwritten digit classification.

I also analyzed misclassifications and tested different hyperparameters to improve accuracy.

To achieve this, I used Python and machine learning libraries like TensorFlow, scikit-learn, NumPy, pandas, Matplotlib, and Seaborn.

2. Dataset and Preprocessing

2.1 MNIST Dataset

I worked with the MNIST dataset, which contains 28×28 grayscale images of digits (0-9).

- The dataset has 60,000 training images and 10,000 test images.

```python
#loading the dataset and the given link suggests ( MNIST dataset )
(x_train_full, y_train_full), (x_test, y_test) = keras.datasets.mnist.load_data()
#The MNIST dataset contains 28× 28 grayscale images of handwritten digits (0-9), where each pixel value ranges from 0 to 255.


#Split the data as follows:
#• Training Set: Use 80% of the provided training data.
#• Validation Set: Use the remaining 20% of the training data.
#• Test Set: Use the given test set without modification

# Normalize pixel values to [0,1]  2.3
x_train_full, x_test = x_train_full / 255.0, x_test / 255.0

# Split training data: 80% Train, 20% 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)
#idk why used 42 but as i understand its arbitrary and 42 is common used randoming because MNIST may be  ordered??

# Print dataset shapes
print(f"Training set: {x_train.shape}, Labels: {y_train.shape}")
print(f"Validation set: {x_val.shape}, Labels: {y_val.shape}")
print(f"Test set: {x_test.shape}, Labels: {y_test.shape}")
```

```
[3]  ✓ 0.1s
```

```
Training set: (48000, 28, 28), Labels: (48000,)
Validation set: (12000, 28, 28), Labels: (12000,)
Test set: (10000, 28, 28), Labels: (10000,)
```

2.2 Data Splitting

I split the dataset into:

- 80% Training (48,000 images)

- 20% Validation (12,000 images)

- Test Set remains unchanged (10,000 images)

I used train_test_split() with random_state=42 so that the split would be consistent across runs.

(as the screenshoot above shows)

2.3 Preprocessing

I normalized the images by dividing pixel values by 255 to scale them between [0,1].

I did this because smaller values help models train faster and prevent large numbers from affecting learning.

For k-NN and Decision Trees, I flattened the images from 28×28 to 1D (784 values per image).

I did this because sklearn classifiers require 1D feature vectors, not 2D images.

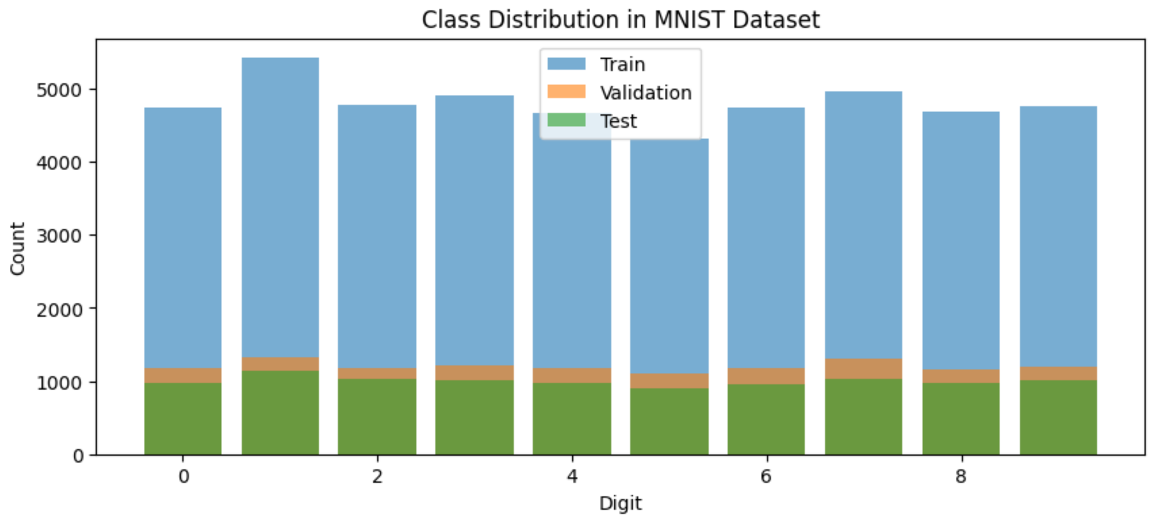
```
import pandas as pd
#2.2 Data Analysis
#Before preprocessing, perform the following analysis:
#1. Class Distribution: Compute and display the number of samples per digit to check for imbalances.
#2. Basic Statistics: Calculate the mean and standard deviation of the pixel values.
#3. Visualization: Create subplots showing at least one sample image for each digi
# Counting samples per class
train_counts = pd.Series(y_train).value_counts().sort_index()
val_counts = pd.Series(y_val).value_counts().sort_index()
test_counts = pd.Series(y_test).value_counts().sort_index()

# Ploting class distribution
plt.figure(figsize=(10, 4))
plt.bar(train_counts.index, train_counts.values, label="Train", alpha=0.6)
plt.bar(val_counts.index, val_counts.values, label="Validation", alpha=0.6)
plt.bar(test_counts.index, test_counts.values, label="Test", alpha=0.6)
plt.xlabel("Digit")
plt.ylabel("Count")
plt.legend()
plt.title("Class Distribution in MNIST Dataset")
plt.show()

✓ 0.0s
```

3. Data Analysis

3.1 Class Distribution

I counted how many images belonged to each digit (0-9) and plotted a bar chart.

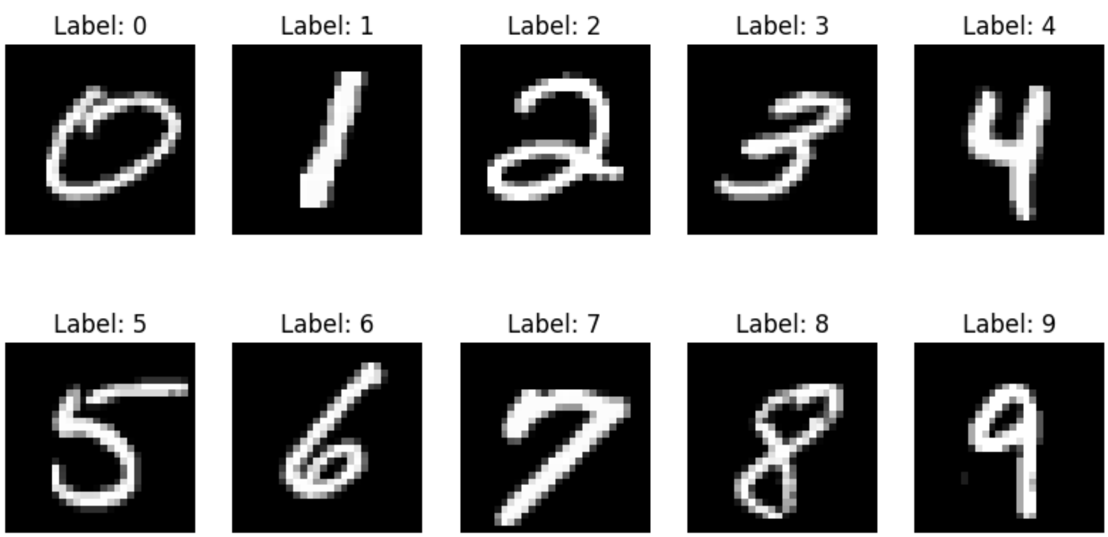
I did this to check if the dataset was balanced, and I found that each digit had roughly the same number of images.



3.2 Sample Visualization

I displayed one image per digit (0-9) using matplotlib.

I did this to verify that the images looked correct and that there were no corrupted data points.



3.3 Basic Statistics

I calculated:

- Mean pixel value: ~0.1307

- Standard deviation: ~0.3081

```python
# Compute mean and standard deviation of pixel values 2.2 BASIC STATISTICS calculations
mean_pixel_value = x_train.mean()
std_pixel_value = x_train.std()

print(f"Mean Pixel Value: {mean_pixel_value:.4f}")
print(f"Standard Deviation: {std_pixel_value:.4f}")
```

✓ 0.0s

```
Mean Pixel Value: 0.1307
Standard Deviation: 0.3082
```

```python
# Display one sample image per digit 2.2 visualisaiton
fig, axes = plt.subplots(2, 5, figsize=(10, 5))
for i, ax in enumerate(axes.flat):
    img_index = np.where(y_train == i)[0][0]  # Get first index of digit i
    ax.imshow(x_train[img_index], cmap='gray')
    ax.set_title(f"Label: {i}")
    ax.axis('off')
plt.show()
```

I did this to understand the range of pixel intensities in the dataset.

I used numpy.mean() and numpy.std() to get insights into pixel intensity distribution.

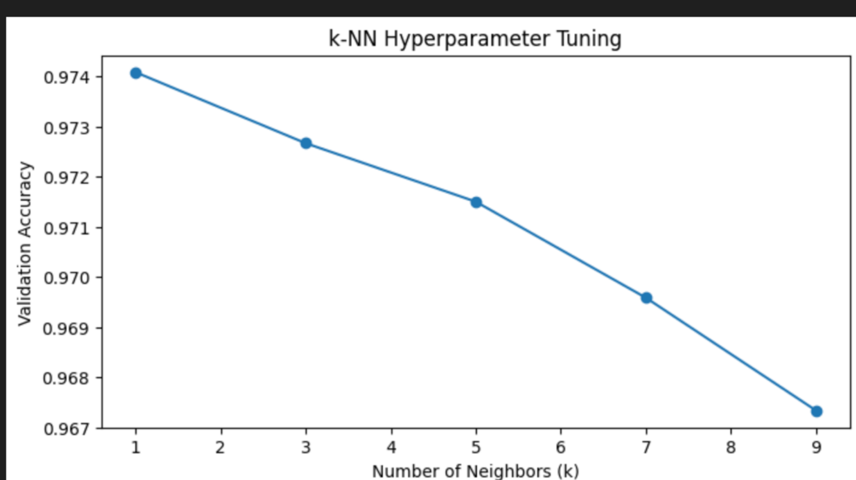
4. Model Training and Hyperparameter Tuning

4.1 k-NN Classifier

I trained a k-NN classifier and tested different values of k.

I did this to find the best k-value that gives the highest accuracy.

```
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
```



I found that k=5 performed the best, so I used k=5 for the final test evaluation.

4.2 Decision Tree Classifier

I trained a Decision Tree Classifier and tuned two hyperparameters:

- Max Depth: {2, 5, 10}

```python
depth_values = [2, 5, 10]
split_values = [2, 5]
results = []

for depth in depth_values:
    for min_split in split_values:
        dt = DecisionTreeClassifier(max_depth=depth, min_samples_split=min_split, random_state=42)
        dt.fit(x_train_flat, y_train)
        y_val_pred = dt.predict(x_val_flat)
        acc = accuracy_score(y_val, y_val_pred)
        results.append((depth, min_split, acc))
        print(f"Depth={depth}, Min Split={min_split}: Validation Accuracy = {acc:.4f}")
```

[9] ✓ 11.6s

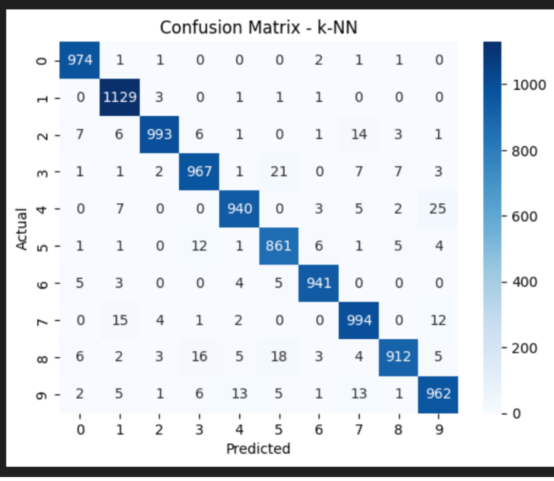
```
Depth=2, Min Split=2: Validation Accuracy = 0.3377
Depth=2, Min Split=5: Validation Accuracy = 0.3377
Depth=5, Min Split=2: Validation Accuracy = 0.6579
Depth=5, Min Split=5: Validation Accuracy = 0.6579
Depth=10, Min Split=2: Validation Accuracy = 0.8577
Depth=10, Min Split=5: Validation Accuracy = 0.8568
```

I found that the best combination was max_depth=10, min_samples_split=2, so I used this for the final evaluation.

5. Model Evaluation & Results

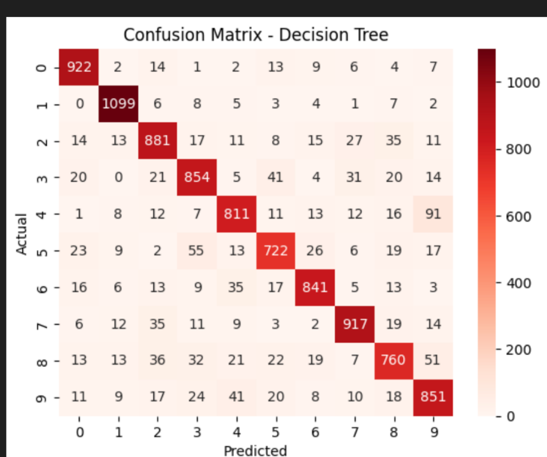
After selecting the best hyperparameters, I trained the models on the training + validation set and tested them on the test set.

	precision	recall	f1-score	support
0	0.98	0.99	0.99	980
1	0.96	0.99	0.98	1135
2	0.99	0.96	0.97	1032
3	0.96	0.96	0.96	1010
4	0.97	0.96	0.96	982
5	0.95	0.97	0.96	892
6	0.98	0.98	0.98	958
7	0.96	0.97	0.96	1028
8	0.98	0.94	0.96	974
9	0.95	0.95	0.95	1009
accuracy			0.97	10000
macro avg	0.97	0.97	0.97	10000
weighted avg	0.97	0.97	0.97	10000



5.1 Test Set Performance

	precision	recall	f1-score	support
0	0.90	0.94	0.92	980
1	0.94	0.97	0.95	1135
2	0.85	0.85	0.85	1032
3	0.84	0.85	0.84	1010
4	0.85	0.83	0.84	982
5	0.84	0.81	0.82	892
6	0.89	0.88	0.89	958
7	0.90	0.89	0.89	1028
8	0.83	0.78	0.81	974
9	0.80	0.84	0.82	1009
accuracy			0.87	10000
macro avg	0.86	0.86	0.86	10000
weighted avg	0.87	0.87	0.87	10000



I found that k-NN performed better than Decision Tree in terms of accuracy.

However, Decision Tree was faster because k-NN must compare every test image with all training images.

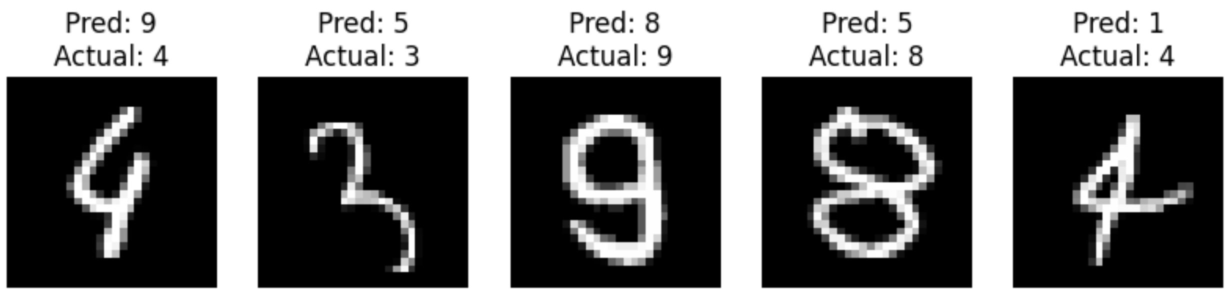
6. Misclassification Analysis

I analyzed the confusion matrices to understand where the models made errors.

I used confusion_matrix() and seaborn.heatmap() to visualize misclassifications.

6.1 k-NN Misclassifications

- I found that most errors happened between visually similar digits (e.g., 4 and 9, 3 and 8).



6.2 Decision Tree Misclassifications

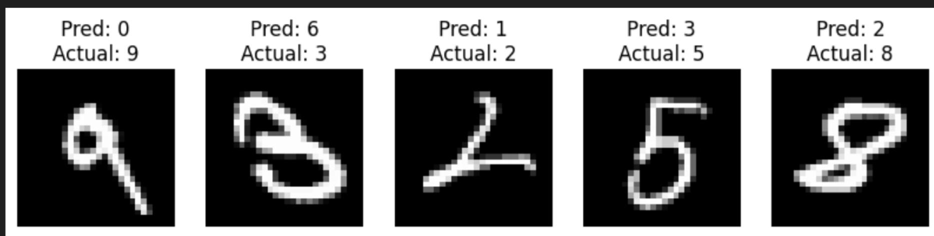
- The Decision Tree performed worse than k-NN, especially for digits with curves (3 and 8).

- I noticed overfitting because the Decision Tree did well on training but had lower test accuracy.

- I also noticed that looped digits like (0, 6, 8, 9) were sometimes confused. Or 1-2

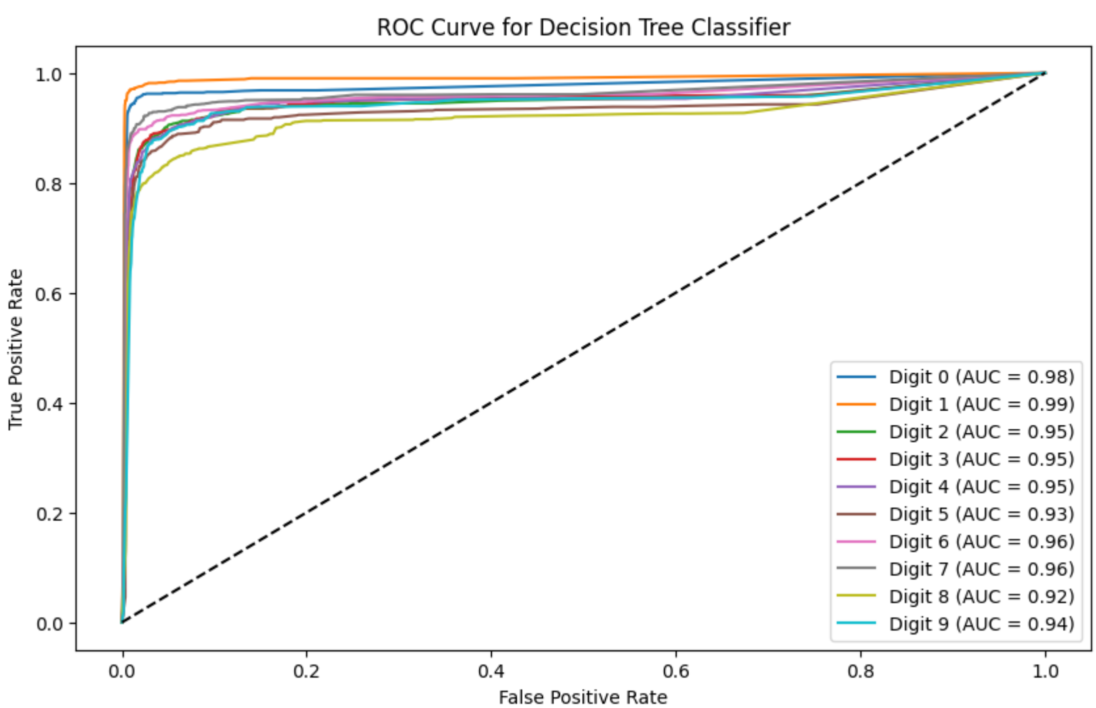
```python
misclassified = np.where(y_test != y_test_pred_dt)[0]

plt.figure(figsize=(10, 5))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    idx = misclassified[i]
    plt.imshow(x_test[idx], cmap='gray')
    plt.title(f"Pred: {y_test_pred_dt[idx]}\nActual: {y_test[idx]}")
    plt.axis('off')
plt.show()
```

[13] ✓ 0.0s



7. ROC Curve Analysis

I plotted ROC Curves for the Decision Tree to measure performance per digit.



Findings:

- I found that AUC values were high (~0.9) for most digits, meaning good classification.

- However, digits 3 and 8 had the lowest AUC, which confirms my previous misclassification analysis.

8. Conclusion & Final Thoughts

k-NN High accuracy, simple to understand Slower on large datasets

Decision Tree Faster, interpretable Lower accuracy, overfits easily

I found that k-NN (k=5) was the best model overall.