Smart Digit Recognizer – Detailed Report

# Overview

This project performs handwritten digit recognition using the MNIST training set (60,000 images). The goal was to preprocess the data, train multiple machine learning models, evaluate them, and visualize predictions. The best-performing model in the experiments was the MLP (Multi-Layer Perceptron).

Dataset source (train only): <https://www.kaggle.com/datasets/rakuraku678/mnist-60000-hand-written-number-images>

# Dataset & Initial Inspection

The dataset was loaded into a pandas DataFrame. The observed shape was (60000, 785):

• Column 0: target label (digit 0–9)  
• Columns 1–784: flattened pixel values for a 28×28 grayscale image (784 features)

## Code: Load & split target/features

```python  
# Column 0 is the target labels (digits)  
y = df.iloc[:, 0]  
  
# Columns 1 to the end are the features (784 pixels)  
X = df.iloc[:, 1:]  
  
# Check the shape of X and y  
print("Shape of X:", X.shape) # Should be (60000, 784)  
print("Shape of y:", y.shape) # Should be (60000,)  
print("First few labels:", y.head())  
```

Observed output:

Shape of X: (60000, 784)  
Shape of y: (60000,)  
First few labels:  
0 5  
1 0  
2 4  
3 1  
4 9  
Name: 0, dtype: int64

## Code: Rename columns

```python  
# Rename columns for clarity  
df.columns = ["target"] + [f"pixel\_{i}" for i in range(df.shape[1] - 1)]  
df.head()  
```

# Data Quality Checks

Checked for missing values and duplicates. No missing values and no duplicates were found in the training set as executed in the notebook.

## Code: Missing values check

```python  
# Filter to show only columns that contain at least one NaN  
null\_counts = df.isnull().sum()  
null\_counts\_with\_values = null\_counts[null\_counts > 0]  
  
# Print only columns with missing values  
print(null\_counts\_with\_values)  
```

Observed output:  
Series([], dtype: int64)

# Class Balance (Distribution of target labels)

Checked counts and normalized proportions for each digit to ensure the dataset is balanced. The notebook output:

Counts per class (value\_counts):

1 6742  
7 6265  
3 6131  
2 5958  
9 5949  
0 5923  
6 5918  
8 5851  
4 5842  
5 5421  
Name: count, dtype: int64

Proportions per class (value\_counts(normalize=True)):  
1 0.112367  
7 0.104417  
3 0.102183  
2 0.099300  
9 0.099150  
0 0.098717  
6 0.098633  
8 0.097517  
4 0.097367  
5 0.090350  
Name: proportion, dtype: float64  
  
Conclusion: The dataset is approximately balanced across classes.

# Outliers Detection and Capping (IQR method)

To reduce the effect of extreme pixel values, the notebook used an IQR-based outlier detection and capped values outside the fences to the fence values (vectorized).

Code: Detect outliers and cap to fences

```python  
# Select only numeric columns  
numeric\_cols = df.select\_dtypes(include=['number']).columns  
  
# Check outliers using IQR  
for col in numeric\_cols:  
 Q1 = df[col].quantile(0.25)  
 Q3 = df[col].quantile(0.75)  
 IQR = Q3 - Q1  
   
 lower\_bound = Q1 - 1.5 \* IQR  
 upper\_bound = Q3 + 1.5 \* IQR  
   
 outliers = df[(df[col] < lower\_bound) | (df[col] > upper\_bound)]  
   
 if not outliers.empty:  
 print(f"Column '{col}' has {len(outliers)} outliers.")  
 else:  
 print(f"Column '{col}' has no outliers.")  
  
# Replace outliers with fences (fast vectorized version)  
for col in numeric\_cols:  
 Q1 = df[col].quantile(0.25)  
 Q3 = df[col].quantile(0.75)  
 IQR = Q3 - Q1  
 lower\_Fence = Q1 - 1.5 \* IQR  
 upper\_Fence = Q3 + 1.5 \* IQR  
   
 # Replace directly without extracting values  
 df[col] = df[col].mask(df[col] < lower\_Fence, lower\_Fence)  
 df[col] = df[col].mask(df[col] > upper\_Fence, upper\_Fence)  
```

After capping, the notebook rechecked columns to confirm outliers were handled.

# Normalization (Min–Max Scaling)

The pixel values were scaled from [0, 255] to [0, 1] using MinMaxScaler. In the notebook the scaler was fit on the full dataset before splitting (note: best practice is to fit on the training set only).

Code: Min–Max normalization

```python  
from sklearn.preprocessing import MinMaxScaler  
  
# normalize numerical columns  
numeric\_cols = X.select\_dtypes('number').columns  
scaler = MinMaxScaler()  
scaler.fit(X[numeric\_cols])  
X[numeric\_cols] = scaler.transform(X[numeric\_cols])  
```

# Train / Test Split

The dataset was split into train and test sets with test\_size = 0.2 (80% train, 20% test) and a fixed random\_state for reproducibility. Stratify by y was used in the notebook to keep class proportions consistent.

Code: split

```python  
from sklearn.model\_selection import train\_test\_split  
  
x\_train, x\_test, y\_train, y\_test = train\_test\_split(  
 X, y, test\_size=0.2, random\_state=42, stratify=y  
)  
```

# Models Trained

Five models were trained and evaluated using the same train/test split. The models and their notebook hyperparameters were:

* • Random Forest: RandomForestClassifier(n\_estimators=100, max\_depth=5, max\_features='sqrt', random\_state=42)
* • Decision Tree: DecisionTreeClassifier(max\_depth=8, random\_state=42)
* • XGBoost: XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss', random\_state=42)
* • Perceptron (single-layer via MLPClassifier): MLPClassifier(hidden\_layer\_sizes=(), activation='logistic', max\_iter=1000, random\_state=42)
* • MLP (two hidden layers): MLPClassifier(hidden\_layer\_sizes=(100, 50), activation='logistic', max\_iter=1000, random\_state=42)

## Code: Example model training snippets

```python  
# Random Forest  
from sklearn.ensemble import RandomForestClassifier  
rf = RandomForestClassifier(n\_estimators=100, max\_depth=5, max\_features='sqrt', random\_state=42)  
rf.fit(x\_train, y\_train)  
  
# Decision Tree  
from sklearn.tree import DecisionTreeClassifier  
dt = DecisionTreeClassifier(max\_depth=8, random\_state=42)  
dt.fit(x\_train, y\_train)  
  
# XGBoost  
import xgboost as xgb  
xgb\_model = xgb.XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss', random\_state=42)  
xgb\_model.fit(x\_train, y\_train)  
  
# Perceptron (single-layer MLP)  
from sklearn.neural\_network import MLPClassifier  
perceptron = MLPClassifier(hidden\_layer\_sizes=(), activation='logistic', max\_iter=1000, random\_state=42)  
perceptron.fit(x\_train, y\_train)  
  
# MLP (two hidden layers)  
mlp = MLPClassifier(hidden\_layer\_sizes=(100, 50), activation='logistic', max\_iter=1000, random\_state=42)  
mlp.fit(x\_train, y\_train)  
```

# Evaluation & Metrics

The notebook computed training and testing accuracies for each model, and also calculated precision, recall, and F1-score (weighted) on the test set. The results observed in the notebook are listed below:

Code: evaluation example and assembling results DataFrame

```python  
from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score  
  
# Example: Random Forest  
y\_train\_pred = rf.predict(x\_train)  
y\_test\_pred = rf.predict(x\_test)  
rf\_train\_accuracy = accuracy\_score(y\_train, y\_train\_pred)  
rf\_test\_accuracy = accuracy\_score(y\_test, y\_test\_pred)  
  
# Then similar for other models and assemble results into a DataFrame  
```

## Observed Results (from the notebook)

The printed summary table in the notebook showed the following results:  
  
Random Forest: Train Accuracy = 0.859417, Test Accuracy = 0.853167  
Decision Tree: Train Accuracy = 0.829187, Test Accuracy = 0.806333  
XGBoost: Train Accuracy = 1.000000, Test Accuracy = 0.975750  
Perceptron: Train Accuracy = 0.912875, Test Accuracy = 0.904750  
MLP (multi-layer): Train Accuracy = 1.000000, Test Accuracy = 0.975000  
  
Precision, Recall and F1 (weighted) were also computed and included in the results table; the top-performing models (XGBoost and MLP) showed precision/recall/F1 ~0.975.

## Results Table (reproduced)

| Model | Train Accuracy | Test Accuracy | Precision | Recall | F1 Score |  
|---|---:|---:|---:|---:|---:|  
| Random Forest | 0.8594 | 0.8532 | 0.8568 | 0.8532 | 0.8514 |  
| Decision Tree | 0.8292 | 0.8063 | 0.8111 | 0.8063 | 0.8071 |  
| XGBoost | 1.0000 | 0.9758 | 0.9758 | 0.9758 | 0.9757 |  
| Perceptron | 0.9129 | 0.9048 | 0.9045 | 0.9048 | 0.9046 |  
| MLP | 1.0000 | 0.9750 | 0.9750 | 0.9750 | 0.9750 |

# Prediction Visualization (Test Images)

The notebook included a plotting snippet to display multiple test images with true labels and model predictions. Initially the code used fixed indices (0..7) which always showed the same images on each run. The notebook was updated to pick random indices so each run shows different test examples. Also, the KeyError observed earlier was due to using pandas indexing on DataFrame/Series instead of numpy arrays; the fix was to convert to numpy before indexing.

Code: plotting random test images with predictions

```python  
import matplotlib.pyplot as plt  
import numpy as np  
  
# ensure numpy arrays instead of pandas DataFrames  
x\_test\_np = np.array(x\_test)  
y\_test\_np = np.array(y\_test)  
  
num\_images = 8 # number of images to display  
random\_indices = np.random.choice(len(x\_test\_np), num\_images, replace=False)  
  
plt.figure(figsize=(12,6))  
for i, idx in enumerate(random\_indices):  
 image = x\_test\_np[idx].reshape(28,28)  
 true\_label = y\_test\_np[idx]  
 pred\_label = mlp.predict(x\_test\_np[idx].reshape(1, -1))[0]  
  
 plt.subplot(2, num\_images//2, i+1)  
 plt.imshow(image, cmap='gray')  
 plt.title(f"True: {true\_label}, Pred: {pred\_label}")  
 plt.axis('off')  
  
plt.tight\_layout()  
plt.show()  
```

Code: explanation for KeyError and fix

```python  
# KeyError usually occurred because x\_test was a pandas DataFrame or y\_test was a Series.  
# Accessing x\_test[i] on a DataFrame looks for a column named 'i' rather than the i-th row.  
# Convert to numpy arrays first to index by integer position:  
  
x\_test\_np = np.array(x\_test)  
y\_test\_np = np.array(y\_test)  
  
# Now indexing x\_test\_np[i] returns the i-th row (flattened 784-vector) as expected.  
```

# Conclusion

This notebook applied multiple classical machine learning models to the MNIST training set, including Decision Tree, Random Forest, XGBoost, Perceptron, and a multi-layer MLP. Results showed that XGBoost and the multi-layer MLP achieved the highest test accuracy (~97.5%). The MLP model was selected as the best-performing model in this experiment.  
  
Recommendations:  
- For stricter reproducibility, fit scalers on the training set only and then transform the test set.  
- Use cross-validation and hyperparameter tuning for more robust model selection.  
- Plot confusion matrices to inspect per-class errors and common digit confusions.