Handwritten digit recognition using Machine Learning Classification

Espen Standal

Table of Contents

1. Introduction *1*

2. Background *1*

3. Problem statement *1*

4. Project objectives *2*

5. Data warehouse details *2*

6. Interesting insights *2*

7. Data pre-processing *2*

8. Measuring performance *2*

9. Algorithms applied *3*

9.1 Algorithm 1 3

9.2 Algorithm 2 3

9.3 Algorithm 3 3

9.4 Comparison 3

10. Discussion *3*

11. Conclusion *3*

References *4*

1. Introduction

Can a computer understand your handwriting? Humans effortlessly recognise numbers scribbled on sticky notes or scrawled on a whiteboard, but teaching a machine to do the same is far from trivial. The human visual system unconsciously filters noise, fills in gaps, and corrects for distortions. Computers, by contrast, must be given an explicit strategy-or learn one-to interpret the same patterns. This report delves into the classic problem of handwritten digit recognition, demonstrating how modern algorithms can learn to distinguish digits based solely on pixel intensity values.

The challenge is framed around the MNIST dataset, a benchmark collection of 70 000 28×28-pixel grayscale images of digits 0-9. These images provide a standard testbed for evaluating computer-vision models and encapsulate much of the variability found in real-world handwriting: different stroke widths, sizes, angles, and writing instruments. Each pixel intensity ranges from 0 (black) to 255 (white), resulting in 784 raw numerical features per image. Despite its apparent simplicity, MNIST remains a compelling dataset because it captures many of the subtleties of image classification while remaining computationally tractable.

This study explores three representative machine-learning paradigms-Logistic Regression, Random Forest, and a Multilayer Perceptron (MLPClassifier)-to compare how linear models, ensemble methods, and neural networks learn from the same visual data. By analysing model accuracy, per-class precision and recall, confusion matrices, and misclassification visuals, we aim to uncover deeper insights into why certain algorithms excel and where they struggle. In doing so, readers will not only see “which model wins,” but will also understand the design trade-offs behind model selection, interpretability, and computational cost.

Beyond academic interest, digit recognition underpins many real-world systems: mobile cheque deposit, digital form processing, automatic number-plate recognition, and more. A solid understanding of how different models approach this seemingly simple task prepares practitioners for larger, messier problems in computer vision, such as full-page optical character recognition (OCR), autonomous-vehicle sign reading, or medical-image analysis.

Key contributions of this report:

- A reproducible workflow for MNIST classification using only mainstream Python libraries (NumPy, pandas, scikit-learn, Matplotlib).

- A side-by-side empirical comparison of three algorithm families, highlighting strengths, weaknesses, and resource requirements.
- Diagnostic visualisations-class distribution, misclassification grids, and confusion matrices-that reveal practical insights beyond headline accuracy figures.

By the end of this report, readers should have a clear picture of how even modest machine-learning models can achieve high accuracy on image tasks, the situations in which each algorithm shines, and how the lessons learned here scale to more complex vision challenges.

2. Background

Computer Vision and Handwritten Digit Recognition

Computer vision is the field of artificial intelligence that focuses on teaching machines to interpret and understand the visual world. It allows computers to process images, detect objects, track motion, and, ultimately, make decisions based on what they "see." For humans, vision is second nature-we can instantly recognize faces, distinguish numbers, or read a handwritten note. But for machines, interpreting images involves complex computations across thousands of tiny data points known as pixels.

One of the earliest and most practical problems tackled in computer vision was the recognition of handwritten digits. In the pre-digital era, institutions like banks and post offices faced the monumental task of manually processing forms, checks, and envelopes-much of which involved reading handwritten numbers. Automating this task meant designing systems that could deal with highly variable input: different writing styles, speeds, slants, and levels of legibility. As such, handwritten digit recognition became a benchmark problem-a controlled, well-defined challenge that could be used to test the capabilities of visual recognition systems.

The Role of Machine Learning in Visual Recognition

Traditional approaches to digit recognition relied on rule-based systems-hardcoded instructions to detect loops, lines, and angles. However, these systems quickly broke down when faced with even slight inconsistencies in handwriting. This led to a shift toward machine learning, where models are trained on large datasets and learn to identify patterns statistically, rather than

following explicit instructions. The key advantage of machine learning is its ability to generalize from examples-a critical skill when working with human-written content, which is inherently messy and unpredictable.

Handwritten digit recognition sits at the intersection of pattern recognition and visual learning. It is complex enough to require robust algorithms, yet simple enough to allow researchers to test, iterate, and improve their models quickly. That's why it remains a standard dataset in tutorials, research papers, and academic courses.

Why This Problem Still Matters

Although digit recognition might seem like a solved problem, it serves as a gateway to far more complex applications. The same techniques used to classify digits can be extended to recognizing letters, words, faces, traffic signs, and objects in photos or videos. Today's handwriting recognition systems power technologies like automatic form scanning, license plate reading, and mobile document capture.

Moreover, studying digit classification offers insights into how models learn, what they struggle with, and how we can improve their performance. It allows us to test various algorithms-from simple linear models to sophisticated neural networks-on the same task and explore their strengths and limitations.

In this project, the focus isn't just on achieving high classification accuracy. The broader goal is to understand how different models interpret visual information, how they differ in their approach to learning, and what this tells us about the design of intelligent systems.

3. Problem statement

The task of enabling machines to read human handwriting accurately highlights the broader difficulty of teaching computers to understand visual patterns. While humans can effortlessly recognize handwritten digits-even when they're written hastily, at odd angles, or in unfamiliar styles-machines require detailed instructions or learned representations to achieve even modest levels of success.

Handwritten digit recognition, in particular, presents a unique challenge. Though it may seem straightforward at first glance, the reality is far more

complex. Variations in how people write-differences in stroke width, curvature, spacing, orientation, and pressure-can drastically alter the appearance of a digit. A "2" written by one person may closely resemble a "3" written by another, and even the same person might write a digit differently across multiple instances. These inconsistencies make it difficult for a model to rely on strict patterns or templates; instead, it must learn the essence of each digit class and recognize it in many different forms.

This problem is further complicated by the way visual information is represented numerically. In the MNIST dataset, each digit is stored as a 784-dimensional vector of pixel values, meaning that models must navigate a high-dimensional space in which the visual relationships between digits are not always intuitively obvious. The challenge is not simply matching shapes, but learning invariant features-those aspects of a digit that remain consistent despite distortions, noise, or writing styles.

Solving this problem is more than an academic exercise. It reflects a broader need in artificial intelligence: the need to build systems that can understand, adapt, and generalize from ambiguous, imperfect data. Handwritten digit recognition serves as a foundational problem that prepares the ground for more complex tasks in optical character recognition (OCR), document analysis, signature verification, and even scene understanding in autonomous systems.

In this project, the problem is approached through a comparative study of three machine learning algorithms-Logistic Regression, Random Forest, and Multilayer Perceptron-each tasked with learning the structure of handwritten digits and making accurate predictions on unseen data. The objective is not only to build a working classifier, but to gain insight into how different learning algorithms approach visual data, what patterns they capture, and where they fall short.

4. Project objectives

Main Objective:
Develop a machine learning system capable of accurately classifying handwritten digits from image data.

Sub-Objectives:

-Preprocess the dataset by normalizing pixel values to improve model performance.

-Implement and train a Logistic Regression model.

-Implement and train a Random Forest classifier.

-Implement and train a Multilayer Perceptron (MLPClassifier).

-Evaluate each model using accuracy, precision, recall, F1-score, and confusion matrix.

-Compare the performance of all three models to determine which best handles digit recognition.

-Visualize misclassified examples to better understand each model's weaknesses.

5. Data warehouse details

The dataset used in this study was sourced from Kaggle, a popular platform for data science competitions and datasets. Specifically, the data was obtained from the publicly available “MNIST in CSV” dataset provided by user oddrationale. This version of the MNIST dataset is formatted for easy use in traditional machine learning workflows, where each image is represented as a single row of pixel values.

No merging or integration of additional sources was necessary for this project, as both training and test data were provided in a clean, pre-split format. This made it easy to ingest the data directly into the machine learning pipeline without extensive preprocessing or cleaning.

6. Interesting insights

An observation made from exploratory data analysis:

The label counts (0-9) are fairly evenly distributed - between ~5400 to ~6700 samples per class

This is excellent - no class imbalance, so models won't be biased

7. Data pre-processing

Normalization

```
18    # Normalize pixel values to [0, 1]
19    X_train /= 255.0
20    X_test /= 255.0
```

Figure 1.

All pixel values in the MNIST dataset originally ranged from 0 to 255, with each value representing the intensity of a grayscale pixel (0 for black, 255 for white). In this raw form, the dataset contains high-magnitude numerical values that can introduce issues for many machine learning algorithms.

To address this, the data was normalized to a range of [0, 1] by dividing every pixel value by 255 as shown in figure 1. This simple transformation plays a crucial role in ensuring effective model training and improving performance.

Normalization serves several important purposes:

- **Numerical stability:** Algorithms like logistic regression and neural networks use gradient-based optimization. Large feature values can lead to unstable or slow convergence during training. Normalization brings all features to a similar scale, helping the optimizer perform more efficiently.

- **Fair feature comparison:** Without normalization, models might mistakenly assign more importance to features with larger numerical ranges (e.g., brighter pixels) even if they're not more informative. Scaling ensures all pixels contribute equally at the start of training.

- **Improved model generalization:** Normalized inputs lead to smoother learning curves and better generalization to unseen data, especially for models that rely on distance calculations or matrix operations, such as

MLPs.

While some models-like decision trees and random forests are less sensitive to feature scale, normalization is still a good practice when working with image data. It ensures consistent preprocessing across all models, facilitates comparisons, and simplifies pipeline development.

In summary, normalization helps transform raw pixel data into a format that is more amenable to learning, allowing the algorithms to focus on discovering patterns in the shapes of digits, rather than being skewed by the raw intensity of individual pixels.

8. Measuring performance

To assess the effectiveness of the machine learning models developed for handwritten digit recognition, a combination of classification evaluation metrics was used. These metrics were selected to provide both a broad overview of model performance and specific insight into how well each model handles different digit classes.

Accuracy

Accuracy was used as the primary metric for initial evaluation. It measures the percentage of correctly classified digits across the entire test set. While accuracy is easy to understand and useful for quick comparisons, it can be misleading if the dataset is imbalanced. Fortunately, the MNIST dataset is balanced across digit classes, making accuracy a reliable indicator in this case.

Precision, Recall, and F1-Score

To gain deeper insight into model performance across individual digit classes, three additional metrics were used:

- Precision measures how many of the predicted digits were actually correct. It is useful in situations where false positives are costly.

- Recall measures how many actual digits were correctly predicted. It is useful when false negatives are more problematic.

- F1-Score is the harmonic mean of precision and recall. It balances both concerns and is especially helpful when comparing models with slightly different strengths.

Confusion Matrix

The confusion matrix provides a visual summary of correct and incorrect predictions for each class. It helps identify specific digits that models struggle to classify (e.g., confusing “5” with “3” or “8”), making it a valuable diagnostic tool alongside the other metrics.

9. Algorithms applied

This project applies three different machine learning algorithms-Logistic Regression, Random Forest, and Multilayer Perceptron (MLPClassifier)-to the MNIST handwritten digit dataset. These algorithms were selected because they each represent a different learning paradigm: linear modeling, ensemble tree-based learning, and neural network-based learning, respectively. By applying these distinct methods to the same dataset, the project enables a meaningful comparison in terms of accuracy, generalization ability, interpretability, and training efficiency.

9.1 Logistic Regression

Logistic Regression is a linear classification algorithm that models the probability of a sample belonging to a particular class using the sigmoid (logistic) function. Despite its simplicity, it is remarkably powerful in linearly separable problems and provides fast, reliable predictions with minimal computation.

Logistic Regression was chosen as a baseline model. It provides a reference point against which the performance of more complex algorithms can be

measured. While it does not capture nonlinear relationships, it helps illustrate the value added by more flexible models. Additionally, Logistic Regression is highly interpretable, making it a great educational tool for visualizing decision boundaries and understanding feature influence.

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

# Load data
train_df = pd.read_csv("mnist_train.csv")
test_df = pd.read_csv("mnist_test.csv")

# Split into features and labels
X_train = train_df.drop(columns=["label"]).values.astype(np.float32)
y_train = train_df["label"].values

X_test = test_df.drop(columns=["label"]).values.astype(np.float32)
y_test = test_df["label"].values

# Normalize pixel values to [0, 1]
X_train /= 255.0
X_test /= 255.0

# Create and train Logistic Regression model
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)
```

Figure 2.

Figure 2 shows code examples of how the model was created and trained.

```python
# Predict on test set
y_pred = model.predict(X_test)

# Evaluate performance
print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Figure 3.

Figure 3 shows code examples of how the model was evaluated.

```
Test Accuracy: 0.9266

Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.98      0.96       980
           1       0.96      0.98      0.97      1135
           2       0.93      0.90      0.91      1032
           3       0.91      0.91      0.91      1010
           4       0.94      0.94      0.94       982
           5       0.90      0.87      0.88       892
           6       0.94      0.95      0.95       958
           7       0.93      0.92      0.93      1028
           8       0.88      0.88      0.88       974
           9       0.91      0.92      0.91      1009

    accuracy                           0.93     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       0.93      0.93      0.93     10000
```

Figure 4.

```
Confusion Matrix:
[[ 958    0    1    3    1    9    4    3    1    0]
 [   0 1111    4    2    0    2    3    2   11    0]
 [   6    9  930   16   10    3   12    9   33    4]
 [   4    1   17  923    1   25    2   10   19    8]
 [   1    3    8    3  922    0    5    4    6   30]
 [   9    2    3   35    8  779   15    6   31    4]
 [   8    3    7    2    7   16  912    2    1    0]
 [   1    7   24    6    6    1    0  949    2   32]
 [  10   11    6   20    8   28   14    9  856   12]
 [   9    7    1    9   21    7    0   21    8  926]]
```

Figure 5.

Performance Summary

The model achieved an accuracy of approximately 92% as demonstrated in figure 4. It performed reliably for well-separated digits like 0, 1, and 4 but showed weaknesses when differentiating between visually similar digits such as 3, 5, and 8 as shown by the confusion matrix in figure 5. These limitations are expected for a linear model dealing with highly variable, nonlinear image data.

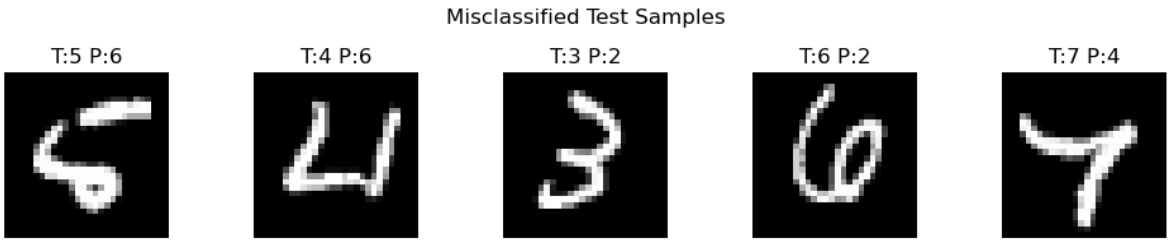


Figure 6.

Figure 6 shows the examples of 5 misclassified digits where “T” represents the true digit and “P” the predicted digit.

9.2 Random Forest Classifier

Random Forest Classifier is an ensemble learning algorithm that builds and combines multiple decision trees to make more accurate and stable predictions. Each tree is trained on a different subset of the data using a process known as bagging (bootstrap aggregating). The final prediction is typically based on a majority vote across all trees. Random Forests are known for their robustness to overfitting and their ability to capture nonlinear decision boundaries.

Random Forest was selected to explore a non-parametric, nonlinear approach to digit classification. It’s particularly well-suited for image data because of its capacity to handle a large number of input features (in this case, 784 pixel values per image). It also offers useful model diagnostics, such as feature importance scores, which can help identify which pixels contribute most to the classification task.

```python
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

# Load the MNIST data
train_df = pd.read_csv("mnist_train.csv")
test_df = pd.read_csv("mnist_test.csv")

# Prepare features and labels
X_train = train_df.drop(columns=["label"]).values.astype(np.float32)
y_train = train_df["label"].values

X_test = test_df.drop(columns=["label"]).values.astype(np.float32)
y_test = test_df["label"].values

# Normalize pixel values to range [0, 1]
X_train /= 255.0
X_test /= 255.0

# Initialize and train Random Forest model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

Figure 7.

Figure 7 shows code examples of how the model was created and trained.

```python
# Predict on test set
y_pred = model.predict(X_test)

# Evaluate model
print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Figure 8.

Figure 8 shows code examples of how the model was evaluated.

```
Test Accuracy: 0.9704

Classification Report:
              precision    recall  f1-score   support

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

```
Confusion Matrix:
[[ 971    0    0    0    0    2    3    1    3    0]
 [   0 1127    2    2    0    1    2    0    1    0]
 [   6    0 1002    5    3    0    3    8    5    0]
 [   1    0    9  972    0    9    0    9    8    2]
 [   1    0    0    0  955    0    5    1    4   16]
 [   5    1    1    9    2  860    5    2    5    2]
 [   7    3    0    0    3    3  937    0    5    0]
 [   1    4   20    2    0    0    0  989    2   10]
 [   4    0    6    7    5    5    5    4  930    8]
 [   7    6    2   12   12    1    0    4    4  961]]
```

Figure 10.

Performance Summary

Random Forest achieved an accuracy of approximately 97%, outperforming Logistic Regression in nearly every digit class. It handled class imbalance and digit variation better, due to its ability to model complex decision boundaries. However, it required significantly more memory and training time, especially as the number of trees increased.

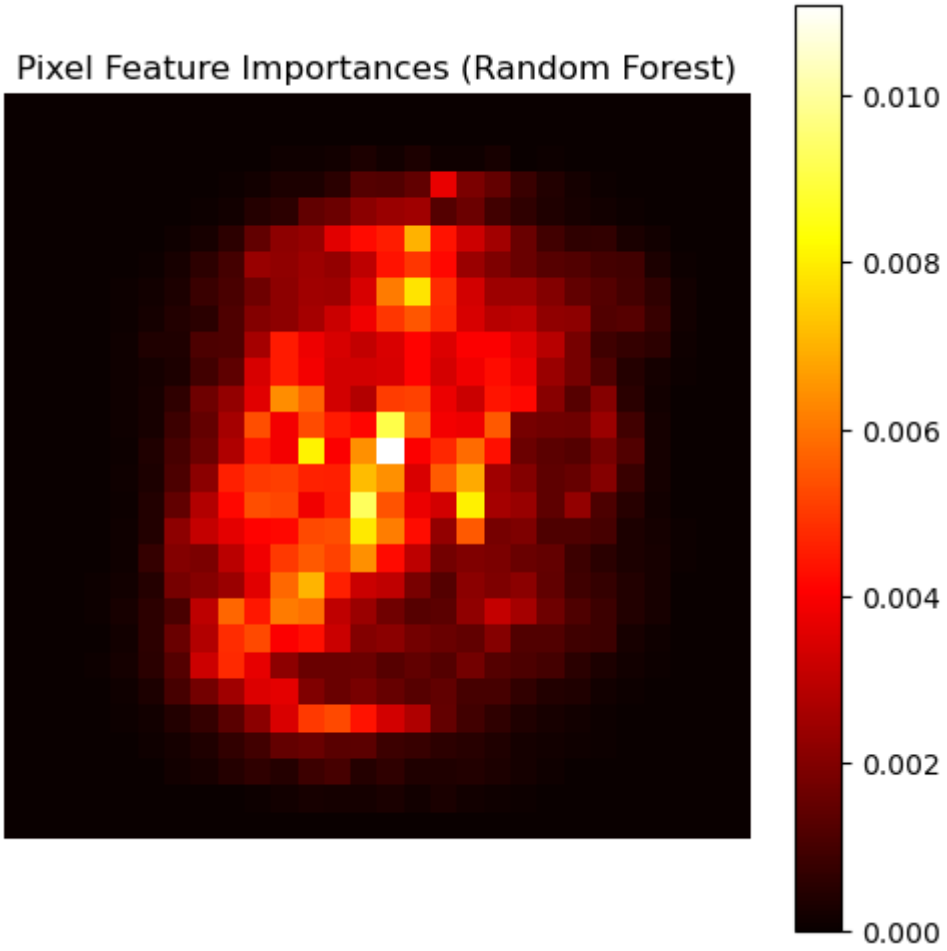


Figure 11.

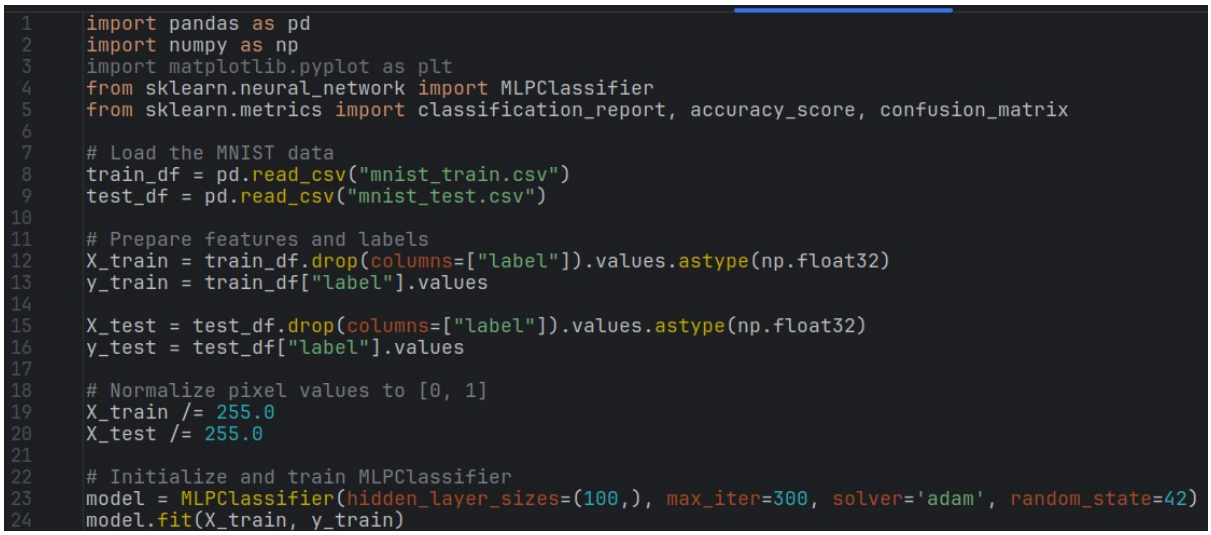
Figure 11 shows which regions of the image the model relies on most when making predictions. It was made using Matplotlib.

9.3 MLPClassifier

The MLPClassifier is a feedforward neural network composed of an input layer, one or more hidden layers, and an output layer. It uses backpropagation and gradient descent to update its internal weights based on error signals during training. Unlike logistic regression, MLPs can learn nonlinear relationships, making them ideal for complex pattern recognition problems like image classification.

The MLPClassifier was included to represent a neural network-based approach that doesn't require specialized deep learning frameworks. It provides a middle

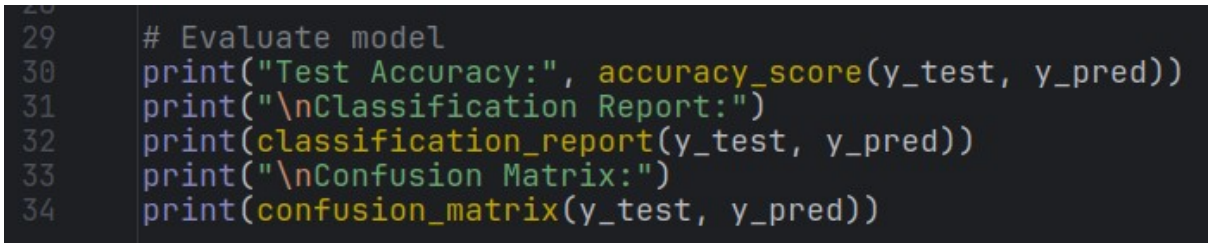
ground between traditional machine learning and deep learning, offering better flexibility and accuracy than linear models, but with less overhead than convolutional neural networks (CNNs). MLPs are particularly effective when the input data is already flattened and normalized, as is the case with the MNIST CSV format.



```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

# Load the MNIST data
train_df = pd.read_csv("mnist_train.csv")
test_df = pd.read_csv("mnist_test.csv")

# Prepare features and labels
X_train = train_df.drop(columns=["label"]).values.astype(np.float32)
y_train = train_df["label"].values

X_test = test_df.drop(columns=["label"]).values.astype(np.float32)
y_test = test_df["label"].values

# Normalize pixel values to [0, 1]
X_train /= 255.0
X_test /= 255.0

# Initialize and train MLPClassifier
model = MLPClassifier(hidden_layer_sizes=(100,), max_iter=300, solver='adam', random_state=42)
model.fit(X_train, y_train)
```

Figure 12.

Figure 12 shows code examples of how the model was created and trained.



```python
# Evaluate model
print("Test Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Figure 13.

Figure 13 shows code examples of how the model was evaluated.

```
Test Accuracy: 0.9786

Classification Report:
              precision    recall  f1-score   support

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

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

Figure 14.

```
Confusion Matrix:
[[ 970    0    1    0    1    0    2    1    4    1]
 [   0 1124    4    0    0    0    2    1    4    0]
 [   4    1 1006    3    1    0    4    5    7    1]
 [   0    0    3  990    0    5    0    3    3    6]
 [   1    1    3    2  957    0    7    3    2    6]
 [   1    1    0    9    2  870    5    0    2    2]
 [   5    2    3    1    5    6  935    0    1    0]
 [   2    3   10    1    2    0    0 1003    3    4]
 [   5    1    3    5    5    2    2    3  945    3]
 [   2    2    0    6    5    4    0    2    2  986]]
```

Figure 15.

Performance Summary

The MLPClassifier achieved an accuracy of around 98%, the highest among the three models tested. It performed especially well with ambiguous digits by learning deep, nonlinear representations from the data. However, this came at the cost of longer training time, greater sensitivity to hyperparameters, and increased computational complexity compared to the other models.

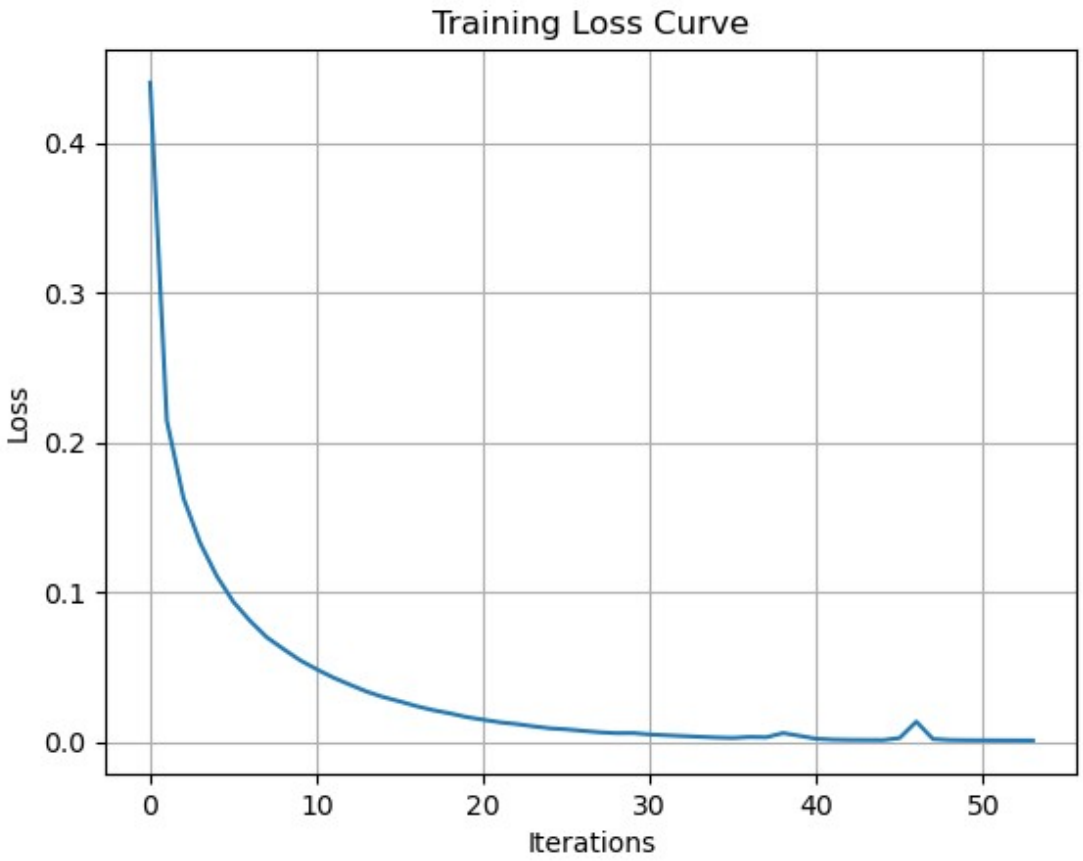


Figure 16.

Figure 16 shows how well the model converged — if the curve flattens, it likely finished learning properly.

9.4 Comparison

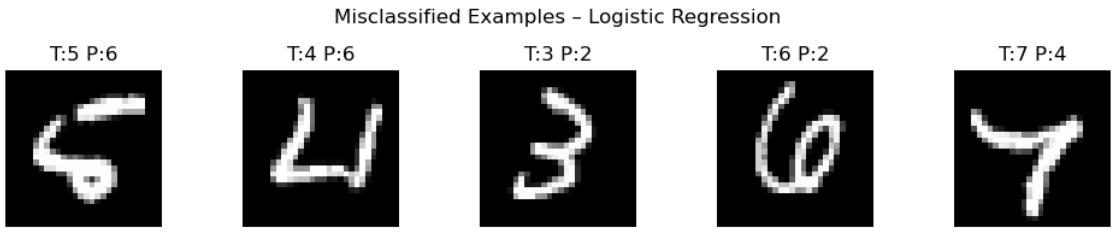


Figure 17.

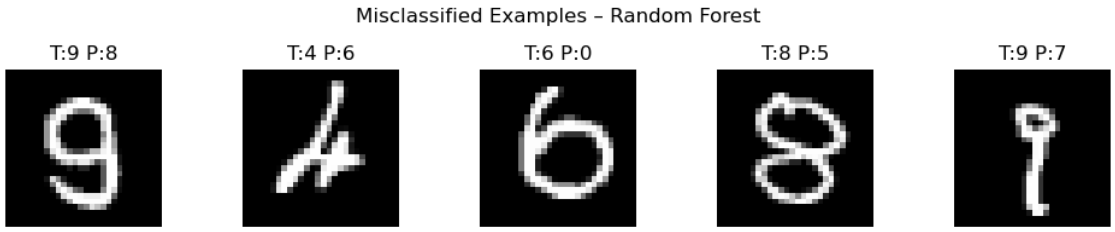


Figure 18.

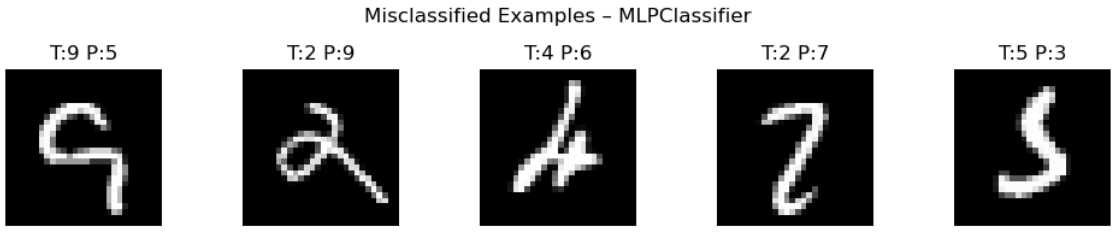


Figure 19.

The comparison of the three models reveals clear differences in performance, complexity, and learning behavior. Logistic Regression, while fast and highly interpretable, struggled with visually similar digits due to its linear nature, achieving an accuracy of around 92%. Random Forest improved on this by capturing non-linear relationships through ensemble decision trees, yielding higher accuracy (~97%) and better generalization across most digit classes. However, the most effective model was the Multilayer Perceptron (MLPClassifier), which demonstrated superior accuracy (~98%) and robustness in handling complex digit variations. Although it required longer training time, the MLP's ability to learn intricate pixel patterns made it the top performer overall. This comparison highlights the trade-offs between simplicity, interpretability, and performance in machine learning models.

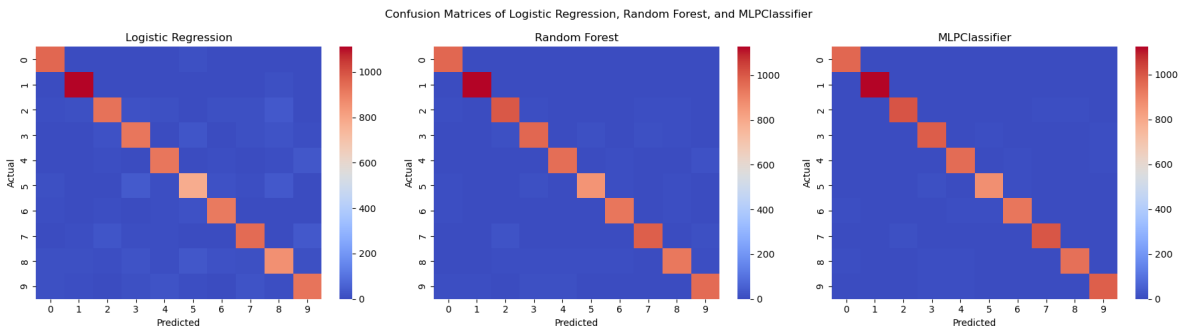


Figure 17.

10. Discussion

The results of this project demonstrate that different machine learning models bring different strengths to the task of handwritten digit recognition. While all three models-Logistic Regression, Random Forest, and MLPClassifier-achieved relatively high accuracy, their performance varied based on complexity, learning capacity, and computational cost.

From a technical perspective, the Multilayer Perceptron (MLPClassifier) showed the highest accuracy, indicating that neural networks are well-suited for capturing the subtle variations in handwritten digits. Random Forest also performed well, offering a good balance between accuracy and interpretability. Logistic Regression, while less accurate, still provided respectable results and served as a strong linear baseline. These findings reinforce a key insight: for visual tasks involving complex patterns, models capable of learning non-linear relationships tend to outperform simpler algorithms.

Beyond accuracy, these models have practical utility in real-world applications. A well-trained digit recognition model can be deployed in automated form processing systems, banking applications for check digitization, postal sorting, and even mobile handwriting input tools. Because MNIST is considered a foundational benchmark, the success of these models can also be seen as a stepping stone to more advanced tasks like optical character recognition (OCR) or handwritten text transcription.

Ultimately, this project illustrates that even relatively simple machine learning models can perform impressively when applied to well-preprocessed visual data. With minimal engineering and no deep learning libraries, it is possible to build accurate and functional recognition systems for real-world use.

11. Conclusion

The primary objective of this project was to develop a machine learning system capable of accurately classifying handwritten digits using the MNIST dataset. Through the implementation and evaluation of Logistic Regression, Random Forest, and MLPClassifier, this objective was successfully achieved. Each model was trained, tested, and analyzed in terms of both performance metrics and

practical applicability. The results showed clear improvements from linear to non-linear models, with the Multilayer Perceptron delivering the best overall performance.

Reflecting on the journey, this project provided a hands-on opportunity to explore how different algorithms interpret and learn from visual data. One important lesson was the significance of data preprocessing-even simple steps like normalization had a measurable impact on performance. Another key takeaway was the value of model comparison, which not only highlighted which model performed best but also why certain models are more appropriate for particular types of data. Working through the full machine learning pipeline-from data to evaluation-offered a clearer understanding of how models behave beyond their accuracy scores.

This experience has deepened my appreciation for the balance between simplicity and performance in machine learning. It also reinforced the importance of interpreting results contextually rather than relying on a single metric. Overall, the study has met its goals and provided a strong foundation for tackling more complex machine learning problems in the future.

References

“MNIST in CSV” dataset - https://www.kaggle.com/datasets/oddrationale/mnist-in-csv/data