Midterm Project Report: Comparative Analysis of Classification Algorithms

Course Name: AI and Machine Learning
Course Code: SDM274

Submitted By:

- **Name:** `Han Lubing`
- **Student ID:** `122102162`
- **Email:** `12210262@mail.sustech.edu.cn`

Date of Submission: `Nov.16, 2025`

Abstract

This report presents a comparative analysis of four classification algorithms—Multiclass Logistic Regression, Multi-Layer Perceptron (MLP), K-Nearest Neighbors (KNN), and Decision Tree—implemented from scratch using NumPy on the MNIST dataset. The results show that non-linear models significantly outperformed their counterparts, with KNN achieving the highest accuracy of 97.14%(at k=3), followed by the MLP at 96.21%. A key finding is the stark trade-off between accuracy and computational efficiency. Despite its top accuracy, KNN's prediction time was prohibitively long (over 10,000 seconds), making it impractical for large-scale applications due to the curse of dimensionality. Conversely, the MLP, despite a lengthy training phase, demonstrated fast inference, positioning it as the most balanced and practical model for real-world deployment. This study highlights the importance of algorithm selection based on the balance of predictive power, computational cost, and suitability for the data's inherent structure, providing a deep, practical understanding of fundamental classifier mechanics.

Keywords: Machine Learning, Classification Algorithms, MNIST Dataset, NumPy, Comparative Analysis

1. Introduction

1.1 Background and Motivation

Classification is one of the most fundamental tasks in machine learning, and understanding the behavior of different classification algorithms is essential for both theoretical learning and practical application. While modern machine-learning development often relies on high-level frameworks, such as scikit-learn or TensorFlow, these tools abstract away many implementation details that are crucial for understanding how models learn from data.

To strengthen this foundational understanding, this project focuses on implementing several classical classification algorithms entirely from scratch using NumPy, enabling a deeper examination of their internal mechanics and computational characteristics.

1.2 Project Objectives

This project aims to implement four classic supervised classification algorithms from first principles using NumPy — Multiclass Logistic Regression, Multi-Layer Perceptron (MLP), K-Nearest Neighbors (KNN), and Decision Tree — and to perform a systematic comparative evaluation on the MNIST dataset. The specific objectives are:

- To validate and compare the classification performance of each algorithm on a multiclass handwritten digit recognition task.
- To measure and contrast training and inference time for each model, assessing computational efficiency in addition to accuracy.
- To analyze per-class performance using precision, recall, F1-score and confusion matrices, identifying which digits are most frequently confused and why.
- To summarize the strengths and limitations of each approach and propose practical improvements (e.g., dimensionality reduction, data augmentation, model ensembles).

1.3 Dataset Description

The MNIST handwritten digits dataset serves as the benchmark for this study. It consists of 70,000 grayscale images representing digits from 0 to 9, each stored as a 28×28 pixel grid flattened into a 784-dimensional vector. MNIST is widely used for evaluating classification algorithms due to its manageable complexity, balanced class distribution, and clear multiclass structure. An 80/20 stratified split ensures that class proportions remain consistent between training and testing sets, preserving data representativeness for evaluation.

2. Methodology

2.1 Multiclass Logistic Regression

Multiclass logistic regression is a linear classifier that maps input features to class probabilities using the softmax function. It serves as a strong baseline model for multiclass problems due to its simplicity and interpretability.

2.1.1 Model Formulation

For an input sample $x \in \mathbb{R}^d$, the model computes class scores:

$$z = x^\top W + b,$$

where $W \in \mathbb{R}^{d \times C}$and $b \in \mathbb{R}^C$.

The softmax function converts these scores into probabilities:

$$\hat{y}_k = \frac{e^{z_k}}{\sum_{j=1}^{C} e^{z_j}}.$$

This corresponds to the following implementation:

```python
def forward(self, X):
    self.X = X
    z = X @ self.W + self.b
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    output = exp_z / np.sum(exp_z, axis=1, keepdims=True)
    return output
```

To train the model, class labels are encoded into one-hot vectors:

```python
one_hot[np.arange(n_samples), y] = 1
```

2.1.2 Loss Function

The model is optimized using softmax cross-entropy:

$$\mathcal{L}_{CE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{C} y_{ik}\log(\hat{y}_{ik}),$$

with an added L2 regularization term:

$$\mathcal{L} = \mathcal{L}_{CE} + \frac{\lambda}{2}\|W\|_2^2.$$

Implemented as:

```python
def entropy_loss(self, y_pred, y_true):
    esp = 1e-15
    y_pred = np.clip(y_pred, esp, 1-esp)
    origin_loss = -np.mean(np.sum(y_true * np.log(y_pred), axis = 1))
    l2_loss = 0.5 * self.reg_lambda * np.sum(self.W**2)
    return origin_loss + l2_loss
```

2.1.3 Parameter Optimization

Gradients for weights and biases are derived from the cross-entropy loss:

$$\nabla_W = \frac{1}{N}X^\top(\hat{Y} - Y) + \lambda W, \quad \nabla_b = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i).$$

Code implementation:

```python
def backward(self, y_true, y_pred):
    grad = y_pred - y_true
    grad_W = self.X.T @ grad / self.X.shape[0] + self.reg_lambda * self.W
    grad_b = np.sum(grad, axis=0) / self.X.shape[0]
    self.W -= self.learning_rate * grad_W
    self.b -= self.learning_rate * grad_b
```

The training loop uses mini-batch gradient descent:

```
for j in range(0, X.shape[0], batch_size):
    X_batch = X[indices[j:j+batch_size]]
    y_batch = y[indices[j:j+batch_size]]
    y_pred = self.forward(X_batch)
    self.backward(y_batch, y_pred)
```

2.1.4 Prediction

Final class assignments are made by selecting the maximum-probability class:

$$\hat{y} = \arg\max_k \hat{y}_k.$$

```
def predict(self, X):
    probability = self.forward(X)
    return np.argmax(probability, axis = 1)
```

2.2 Multilayer Perceptron (MLP)

2.2.1 Layer Structure and Forward Pass

Each layer performs a linear transformation

$$z = XW + b$$

followed by a nonlinearity. Hidden layers use ReLU:

$$\mathrm{ReLU}(z) = \max(0, z)$$

and the final layer uses Softmax to produce class probabilities:

$$\mathrm{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}.$$

The layer stores its input and pre-activation values to support gradient computation.

```
def forward(self, X):
    self.input = X
    self.activation_input = X @ self.W + self.b
    if self.activation == 'relu':
        self.output = np.maximum(0, self.activation_input)
    elif self.activation == 'softmax':
        exp_values = np.exp(self.activation_input -
np.max(self.activation_input, axis=1, keepdims=True))
        self.output = exp_values / np.sum(exp_values, axis=1, keepdims=True)

    return self.output
```

2.2.2Backpropagation and Parameter Gradients

Using the chain rule, the gradient w.r.t. weights and biases is

$$\frac{\partial L}{\partial W} = X^\top \delta, \qquad \frac{\partial L}{\partial b} = \sum \delta,$$

where δ is the backpropagated error.
For ReLU,

$$\delta = \delta_{\text{next}} \cdot \mathbb{1}(z > 0),$$

and for Softmax with cross-entropy,

$$\delta = y_{\text{pred}} - y_{\text{true}}.$$

The code is as follows:

```
def backward(self, grad_output):
    if self.activation == 'relu':
        grad_activation = grad_output * (self.activation_input > 0)
    elif self.activation == 'softmax':
        grad_activation = grad_output

    grad_input = grad_activation @ self.W.T / self.input.shape[0]
    grad_W = self.input.T @ grad_activation / self.input.shape[0]
    grad_b = np.sum(grad_activation, axis=0) / self.input.shape[0]

    return grad_input, grad_W, grad_b
```

2.2.3 MLP Architecture and Training

The MLP stacks several ReLU layers followed by a Softmax output. A forward pass simply pushes inputs through each layer in order, enabling complex nonlinear mappings from input to output. The backward pass iterates over layers in reverse order, passing gradients backward and updating each layer's weights. The use of Softmax with cross-entropy allows the gradient at the output layer to be computed as `y_pred - y_true`, simplifying implementation.

```
def forward(self, X):
    output = X
    for i in range(len(self.layers)):
        output = self.layers[i].forward(output)
    return output

def backward(self, y_true, y_pred):
    grad_output = y_pred - y_true
    for i in range(len(self.layers)-1, -1, -1):
        grad_input , grad_W, grad_b = self.layers[i].backward(grad_output)
        #update
        self.layers[i].W -= self.learning_rate * grad_W
        self.layers[i].b -= self.learning_rate * grad_b
        grad_output = grad_input
```

The loss is cross-entropy:

$$L = -\frac{1}{N} \sum_i y_i \log(\hat{y}_i).$$

And training uses mini-batch gradient descent following this pipeline: forward pass → compute loss → backward pass → parameter update, just the same as the logistic method.

2.3 K-Nearest Neighbors (KNN)

2.3.1 Algorithm Principle

KNN is a non-parametric method that classifies a sample based on the majority label among its k nearest neighbors. The distance metric used is Euclidean distance:

$$d(\mathbf{x}, \mathbf{x}_i) = \sqrt{\sum_{j=1}^{n} (x_j - x_{i,j})^2}.$$

Given a query point, the k closest training points are identified, and the predicted label is the mode of these neighbors.

```python
def euclidean_distance(self, x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))

def find_neighbors_brute(self, x):
    distances = [self.euclidean_distance(x, x_train) for x_train in self.X_train]
    k_indices = np.argsort(distances)[:self.k]
    return self.y_train[k_indices]
```

2.3.2 KD-Tree Optimization

To accelerate neighbor search in high-dimensional space, a KD-Tree is built recursively. Each node splits the data along the axis with the current depth modulo the feature dimension. During search, branches that cannot contain closer neighbors than the current k candidates are pruned. This reduces the average query complexity from O(N) to O(log N) in low to moderate dimensions.

```python
def _build_tree(self, X, y, depth):
    if len(X) == 0:
        return None

    # 选择划分轴
    axis = depth % self.k

    # 按照当前轴的值排序
    sorted_idx = np.argsort(X[:, axis])
    X = X[sorted_idx]
    y = y[sorted_idx]

    # 选择中位数作为分割点
    median_idx = len(X) // 2

    # 创建节点
    node = KDNode(
        point=X[median_idx],
        label=y[median_idx],
        axis=axis
    )

    # 递归构建左右子树
    node.left = self._build_tree(X[:median_idx], y[:median_idx], depth + 1)
```

```
    node.right = self._build_tree(X[median_idx + 1:], y[median_idx + 1:], depth + 1)

    return node
```

During search, we use the max heap to sustain the nearest k neighbors. Branches that cannot contain closer neighbors than the current k candidates are pruned. This reduces the average query complexity from O(N) to O(log N) in low to moderate dimensions.

```
def find_k_nearest(self, point, k):
    # 使用最大堆来维护k个最近邻
    max_heap = []

    def search(node, depth=0):
        if node is None:
            return

        distance = self._distance(point, node.point)

        # 如果堆中元素少于k个，直接加入
        if len(max_heap) < k:
            heapq.heappush(max_heap, (-distance, node.label))
        # 否则，如果当前点比堆顶更近，更新堆
        elif -distance > max_heap[0][0]:
            heapq.heapreplace(max_heap, (-distance, node.label))

        axis = depth % self.k
        diff = point[axis] - node.point[axis]

        # 递归搜索更可能包含近邻的子树
        if diff <= 0:
            search(node.left, depth + 1)
            # 如果到超平面的距离小于当前最大距离，搜索另一子树
            if node.right and (len(max_heap) < k or abs(diff) < -max_heap[0][0]):
                search(node.right, depth + 1)
        else:
            search(node.right, depth + 1)
            if node.left and (len(max_heap) < k or abs(diff) < -max_heap[0][0]):
                search(node.left, depth + 1)

    search(self.root)
    # 返回标签列表
    return [label for _, label in sorted(max_heap, reverse=True)]
```

2.3.3 Prediction

For a new input, the algorithm either uses the KD-Tree or brute-force search to find k nearest neighbors. The predicted class is:

$$\hat{y} = \arg\max_{c} \sum_{i \in N_k} \mathbf{1}(y_i = c),$$

where N_k is the set of k nearest neighbors.

```python
def predict_sample(self, x=None, neighbors=None):
    if neighbors is None:
        neighbors = self.find_neighbors(x)
    vote = np.bincount(neighbors).argmax()
    return vote
```

2.4 Decision Tree

2.4.1 Algorithm Principle

Decision Tree is a tree-structured classifier that recursively splits the dataset based on feature thresholds to maximize information gain. For a node containing dataset D, the entropy is calculated as:

$$H(D) = -\sum_{i=1}^{k} p_i \log_2 p_i,$$

where p_i is the proportion of samples in class i. The information gain from splitting by feature f at threshold t is:

$$IG(D, f, t) = H(D) - \frac{|D_{\text{left}}|}{|D|} H(D_{\text{left}}) - \frac{|D_{\text{right}}|}{|D|} H(D_{\text{right}}),$$

where D_{left} and D_{right} are the subsets after splitting.

```python
def _information_gain(self, X, y, feature, threshold):
    # 计算信息增益
    parent_entropy = self._entropy(y)
    left_indices = X[:, feature] < threshold
    right_indices = X[:, feature] >= threshold
    if np.sum(left_indices) == 0 or np.sum(right_indices) == 0:
        return 0
    left_entropy = self._entropy(y[left_indices])
    right_entropy = self._entropy(y[right_indices])
    child_entropy = (np.sum(left_indices) * left_entropy +
np.sum(right_indices) * right_entropy) / len(y)
    return parent_entropy - child_entropy

def _entropy(self, y):
    if len(y) == 0:
        return 0.0
    num = np.bincount(y)
    probs = num / num.sum()
    return -np.sum(p * np.log2(p) for p in probs if p > 0)
```

The tree grows recursively until either all samples belong to one class or the maximum depth is reached. The leaf node predicts the most common class in its samples.

2.4.2 Tree Construction and Prediction

- **Best Split Selection:** For each feature, all unique values are considered as candidate thresholds. The split maximizing information gain is chosen.

```python
def _best_split(self, X, y):
    # 遍历所有特征和阈值，寻找最佳分割
```

```
    best_feature = None
    best_threshold = None
    best_gain = 0
    for feature in range(X.shape[1]):
        thresholds = np.unique(X[:, feature])
        for threshold in thresholds:
            gain = self._information_gain(X, y, feature, threshold)
            if gain > best_gain:
                best_gain = gain
                best_feature = feature
                best_threshold = threshold
    return best_feature, best_threshold
```

- **Recursion:** The dataset is split into left and right subsets, and the process repeats for each child node.

```
def _build_tree(self, X, y, depth=0):
    # 如果所有样本属于同一类别，或达到最大深度，创建叶节点
    if len(np.unique(y)) == 1 or (self.max_depth is not None and depth == self.max_depth):
        return Node(value=self._most_common_label(y))

    # 寻找最佳分割
    feature, threshold = self._best_split(X, y)
    if feature is None:
        return Node(value=self._most_common_label(y))

    # 创建左右子树
    left_indices = X[:, feature] < threshold
    right_indices = X[:, feature] >= threshold
    left = self._build_tree(X[left_indices], y[left_indices], depth + 1)
    right = self._build_tree(X[right_indices], y[right_indices], depth + 1)
    return Node(feature=feature, threshold=threshold, left=left, right=right)
```

- **Prediction:** A new sample is passed from the root to leaf nodes according to the feature thresholds. The predicted class is the label of the reached leaf node.

```
def _predict_sample(self, x, node):
    if node.value is not None:
        return node.value
    if x[node.feature] < node.threshold:
        return self._predict_sample(x, node.left)
    else:
        return self._predict_sample(x, node.right)
```

In conclusion, we implemented four classification algorithms on the MNIST dataset. **Logistic Regression** is a linear model with softmax and L2 regularization. **MLP** adds hidden layers and nonlinear activations to capture complex patterns. **KNN** classifies based on nearest neighbors, with KD-Tree for efficiency. **Decision Tree** splits features recursively to maximize information gain, offering interpretability but prone to overfitting. These methods cover linear, nonlinear, and non-parametric approaches, providing a basis for comparative analysis.

3. Experimental Setup

3.1 Data Preprocessing

The MNIST dataset was loaded using `fetch_openml`, resulting in 70,000 grayscale images (28×28 pixels) with 10 classes. Each image was flattened to a 784-dimensional vector. Preprocessing included:

1.**Normalization**: Each feature (pixel) was mean-normalized:

$$X_{\text{norm}} = \frac{X - \mu}{\max(X) - \min(X) + \epsilon},$$

where μ is the mean of each feature and $\epsilon = 1 \times 10^{-8}$ prevents division by zero.

2.**Train-Test Split**: The dataset was shuffled and split into 80% training (56,000 samples) and 20% testing (14,000 samples), maintaining class distribution.

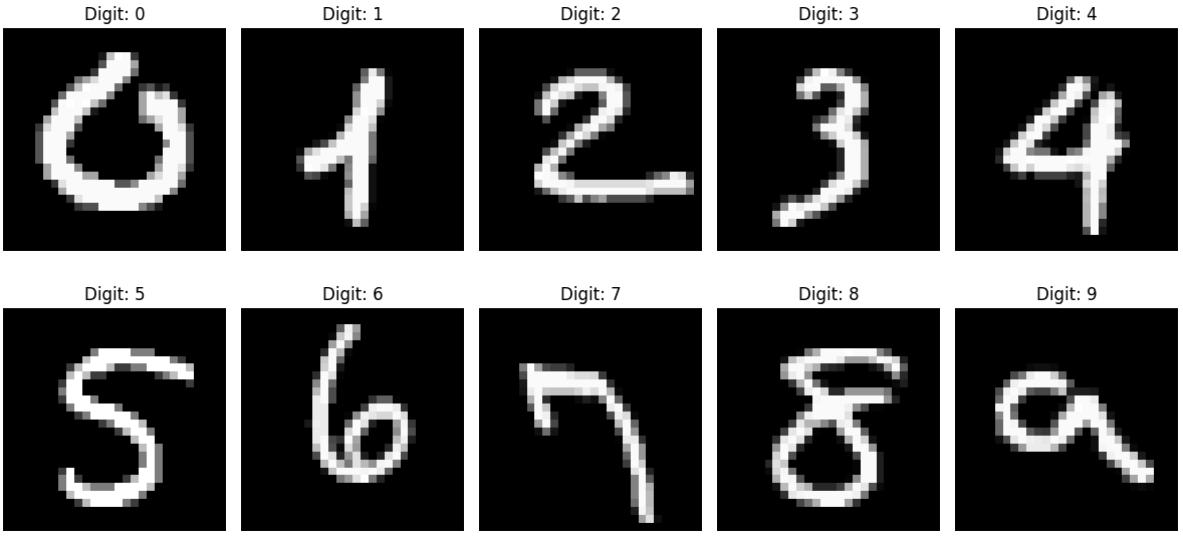
```
def preprocess(self):
    if self.X is None or self.y is None:
        raise ValueError("Data not loaded. Call load_data() first.")

    # normalization
    X_normalized = (self.X - np.mean(self.X, axis=0)) / (self.X.max(axis=0) - self.X.min(axis=0) + 1e-8)

    # data split
    np.random.seed(42)
    indices = np.arange(self.X.shape[0])
    np.random.shuffle(indices)
    split_idx = int(self.X.shape[0] * 0.8)
    train_idx, test_idx = indices[:split_idx], indices[split_idx:]
    self.X_train, self.X_test = X_normalized[train_idx], X_normalized[test_idx]
    self.y_train, self.y_test = self.y[train_idx], self.y[test_idx]
    print("Data preprocessing completed!")

    return self.X_train, self.X_test, self.y_train, self.y_test
```

3.**Visualization**: Sample images from each class were plotted to confirm data integrity and class representation.



3.2 Evaluation Methodology

To assess model performance on the MNIST dataset, we employ multiple metrics capturing both overall accuracy and class-specific performance.

1.**Accuracy** measures the fraction of correctly predicted samples:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Samples}} = \frac{\sum_{i=1}^{N} \mathbf{1}(y_i = \hat{y}_i)}{N}.$$

2.**Precision, Recall, and F1-Score** are computed for each class (i):

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i}, \quad \text{Recall}_i = \frac{TP_i}{TP_i + FN_i}, \quad F1_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i},$$

where (TP_i), (FP_i), (FN_i) are true positives, false positives, and false negatives for class (i).

3.**Macro F1-Score**: the average F1 across all classes:

$$\text{Macro-F1} = \frac{1}{k} \sum_{i=1}^{k} F1_i.$$

4.**Weighted F1-Score**: F1 weighted by the number of samples in each class:

$$\text{Weighted-F1} = \sum_{i=1}^{k} \frac{n_i}{N} \cdot F1_i,$$

where n_i is the number of true samples of class i.

5.**Confusion Matrix**: a $k \times k$matrix C where C_{ij} counts samples of true class i predicted as class j. It provides insight into per-class errors and common misclassifications.

The `Metrics` class handles all evaluation calculations, including accuracy, precision/recall/F1, macro and weighted F1, and confusion matrix visualization. It allows systematic comparison of all four models—Logistic Regression, MLP, KNN, and Decision Tree—under identical preprocessing and testing conditions.

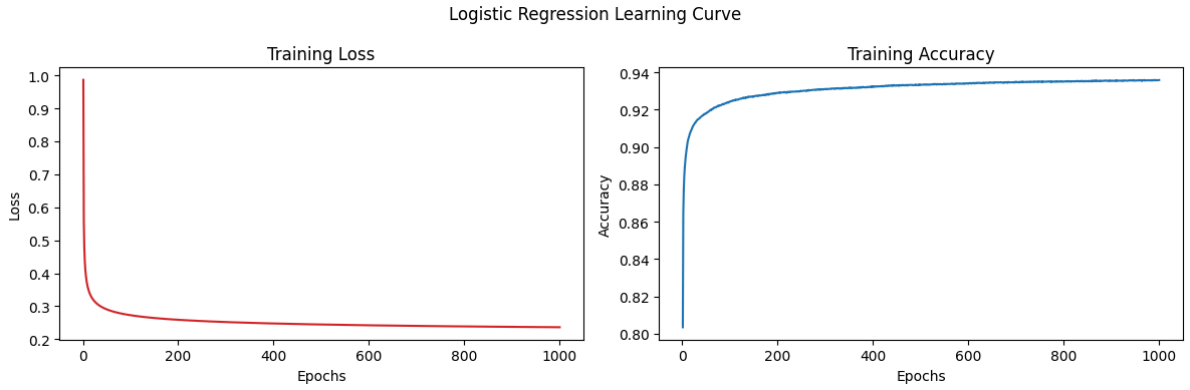
This setup ensures fair comparison among Logistic Regression, MLP, KNN, and Decision Tree classifiers on the multiclass MNIST dataset.

4. Results

This section presents the empirical results obtained from evaluating the four implemented classification algorithms—Multiclass Logistic Regression, Multi-Layer Perceptron (MLP), K-Nearest Neighbors (KNN), and Decision Tree—on the 20% test split (14,000 samples) of the MNIST dataset. The performance of each model was assessed using a comprehensive suite of metrics, including overall accuracy, per-class metrics (precision, recall, and F1-score), macro and weighted-average F1-scores, and the computational time required for training and prediction.

4.1 Multiclass Logistic Regression Results

The Logistic Regression model, optimized using gradient descent with L2 regularization (`λ=0.00001`), achieved an **overall accuracy of 92.26%** on the test set. The learning curve showed stable convergence with learning rate 0.01, reaching a **final training accuracy of 93.59%** after 1000 epochs.



The detailed performance metrics are summarized below:

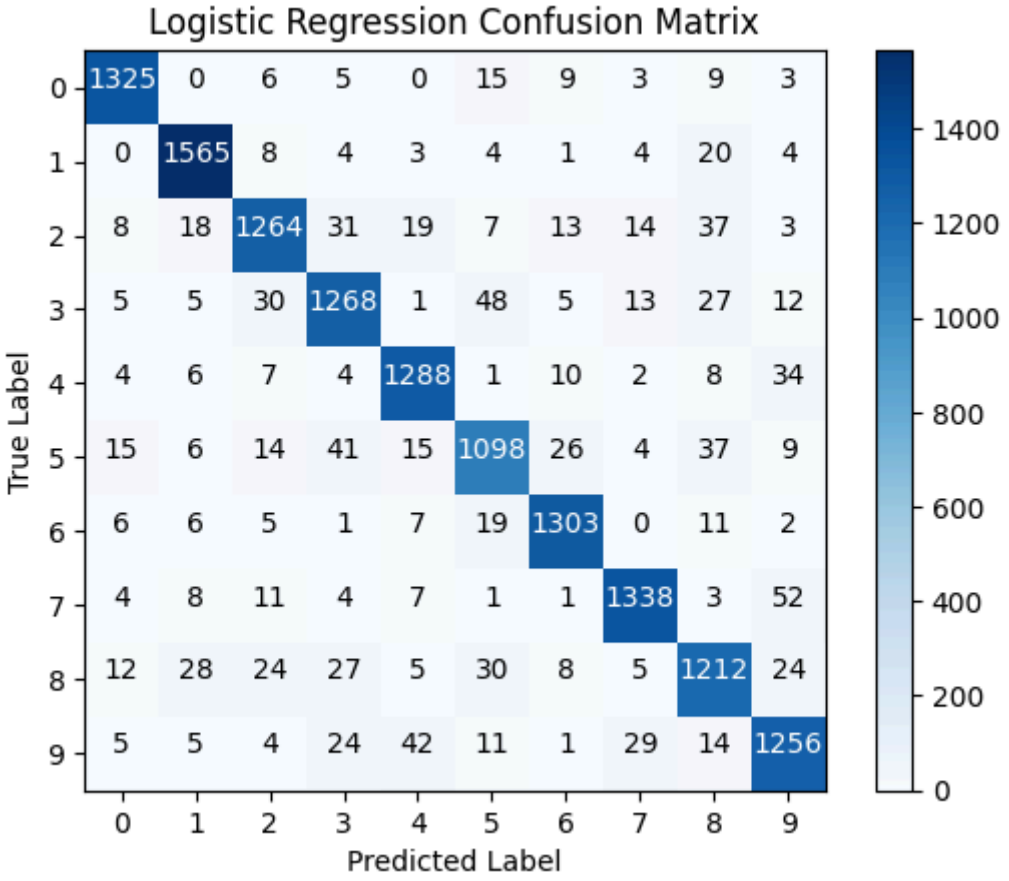
Metric	Value
Overall Accuracy	**92.26%**
Macro F1-Score	92.16%
Weighted F1-Score	92.25%
Training Time	1505.27 seconds
Prediction Time	0.021 seconds

The per-class performance breakdown is provided in the table below:

Digit	Precision	Recall	F1-Score
0	95.74%	96.36%	96.05%
1	95.02%	97.02%	96.01%
2	92.06%	89.39%	90.71%
3	89.99%	89.67%	89.83%
4	92.86%	94.43%	93.64%
5	88.98%	86.80%	87.88%

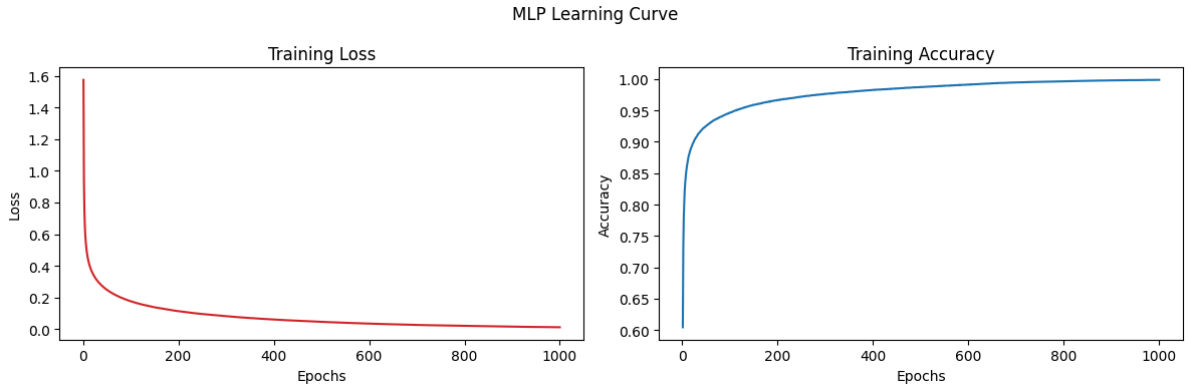
Digit	Precision	Recall	F1-Score
6	94.63%	95.81%	95.21%
7	94.76%	93.63%	94.19%
8	87.95%	88.15%	88.05%
9	89.78%	90.29%	90.04%

The confusion matrix indicates minor confusion between digits such as 4 and 9, as well as 5 and 3. Overall, the model demonstrated robust and balanced performance across most classes.



4.2 Multi-Layer Perceptron (MLP) Results

The MLP, configured with an architecture of two hidden layers (128 and 64 neurons) and the ReLU activation function, delivered the highest accuracy among all models. It achieved an **overall accuracy of 96.21%** on the test data. The learning curve displayed rapid convergence, with a **final training accuracy of 99.88%**, suggesting a very strong fit to the training data.



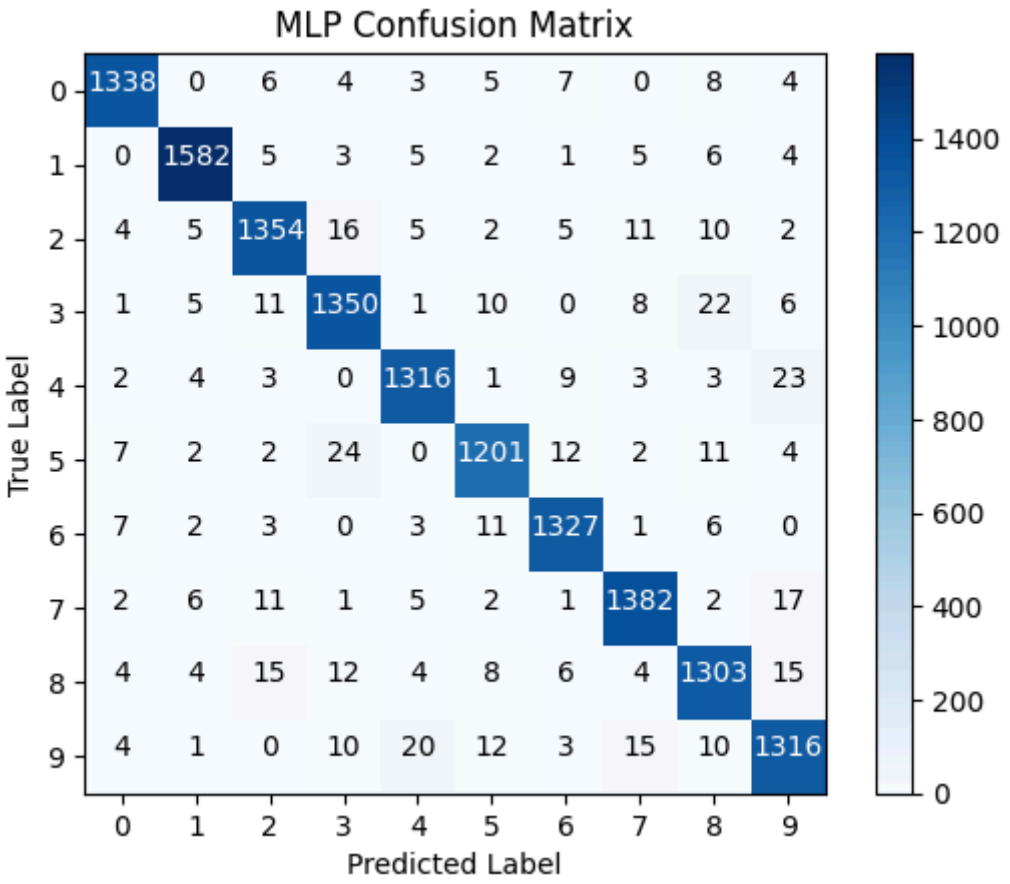
The detailed performance metrics are summarized below:

Metric	Value
Overall Accuracy	**96.21%**
Macro F1-Score	96.17%
Weighted F1-Score	96.21%
Training Time	4721.84 seconds
Prediction Time	0.156 seconds

The per-class performance breakdown is provided in the table below:

Digit	Precision	Recall	F1-Score
0	97.74%	97.31%	97.52%
1	98.20%	98.08%	98.14%
2	96.03%	95.76%	95.89%
3	95.07%	95.47%	95.27%
4	96.62%	96.48%	96.55%
5	95.77%	94.94%	95.36%
6	96.79%	97.57%	97.18%
7	96.58%	96.71%	96.64%
8	94.35%	94.76%	94.56%
9	94.61%	94.61%	94.61%

The MLP's confusion matrix shows a distinct reduction in classification errors compared to Logistic Regression, demonstrating its superior ability to learn complex feature representations.



4.3 K-Nearest Neighbors (KNN) Results

The KNN classifier was evaluated with a `k` value of 20. This model is notable for its near-instantaneous training time, as it primarily involves storing the dataset. However, its prediction time was significantly the highest of all models tested. Its performance was highly competitive, achieving an **overall accuracy of 96.19%**.

The detailed performance metrics for `k=20` are:

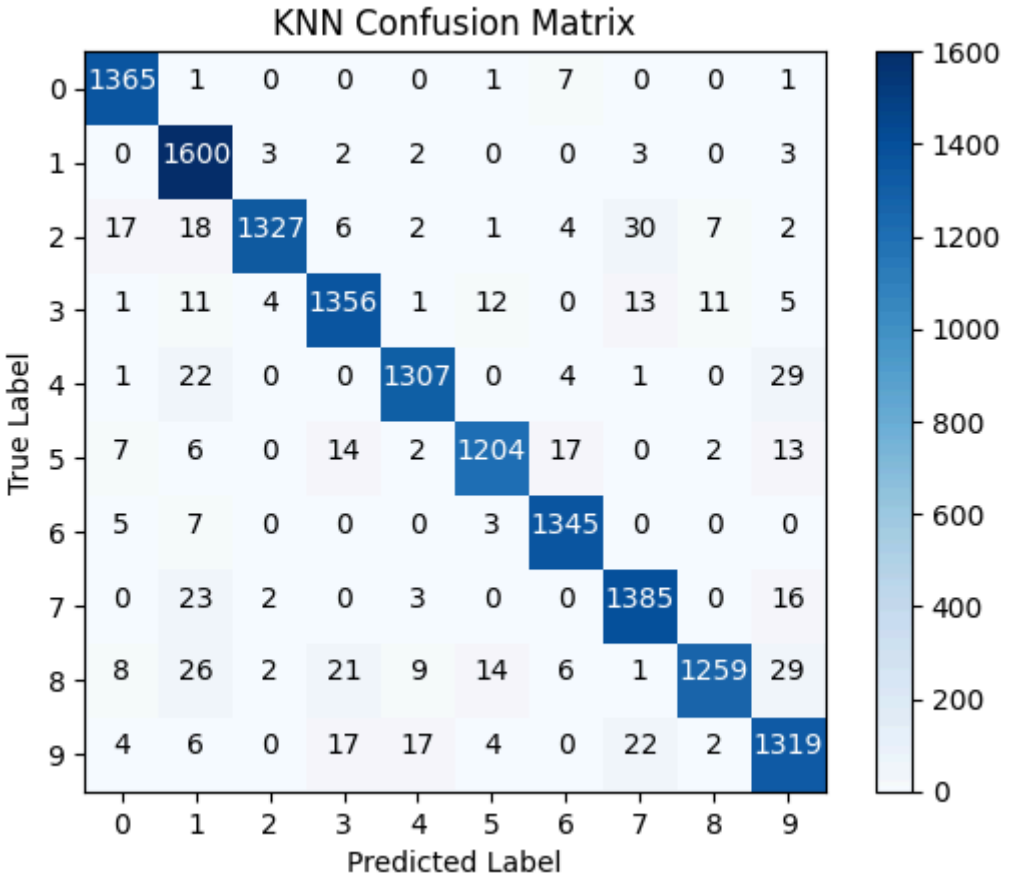
Metric	Value
Overall Accuracy	**96.19%**
Macro F1-Score	96.20%
Weighted F1-Score	96.19%
Training Time	4.56 seconds
Prediction Time	13133.19 seconds

The per-class performance for `k=20` is detailed in the table below:

Digit	Precision	Recall	F1-Score
0	96.95%	99.27%	98.10%
1	93.02%	99.19%	96.01%
2	99.18%	93.85%	96.44%

Digit	Precision	Recall	F1-Score
3	95.76%	95.90%	95.83%
4	97.32%	95.82%	96.56%
5	97.18%	95.18%	96.17%
6	97.25%	98.90%	98.07%
7	95.19%	96.92%	96.05%
8	98.28%	91.56%	94.80%
9	93.08%	94.82%	93.95%

The confusion matrix confirms this strong performance with high values along the diagonal for all digits. The most frequent errors involve misclassifying visually similar digits, such as '2' being confused with '7', and '8' with '1'.



4.3.1 Analysis of the k Hyperparameter

An additional evaluation was performed by varying the value of `k` to observe its impact on accuracy. The results indicate that lower `k` values yielded higher accuracy, peaking at **97.14% with k=3**.

k Value	Accuracy
1	97.07%
3	97.14%

k Value	Accuracy
5	97.00%
7	96.85%
10	96.62%
20	96.19%

4.3.2 Search Algorithm Analysis

The performance of the neighbor search algorithm with kd tree was compared against a brute-force search with `k=20`. This resulted in an identical accuracy (96.19%) but with a prediction time of **10016.25 seconds**. It shows the use of kd tree actually increases the time to find neighbors.

4.4 Decision Tree Results

The Decision Tree was trained with a `max_depth` of 10. This model was the fastest to predict after training but achieved the lowest accuracy of the four algorithms, with an **overall accuracy of 86.85%**.

The detailed performance metrics for `max_depth=10` are:

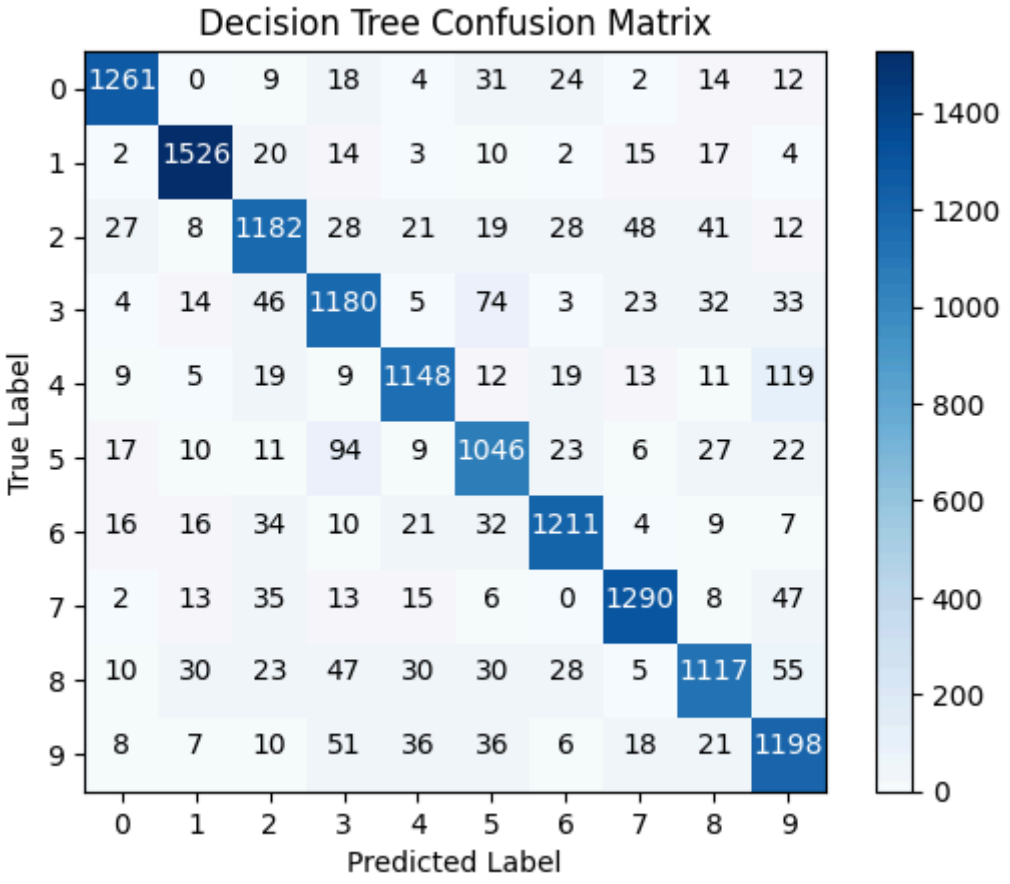
Metric	Value
Overall Accuracy	**86.85%**
Macro F1-Score	86.72%
Weighted F1-Score	86.87%
Training Time	4416.92 seconds
Prediction Time	0.082 seconds

The per-class performance breakdown is provided in the table below:

Digit	Precision	Recall	F1-Score
0	92.99%	91.71%	92.35%
1	93.68%	94.61%	94.14%
2	85.10%	83.59%	84.34%
3	80.60%	83.45%	82.00%
4	88.85%	84.16%	86.45%
5	80.71%	82.69%	81.69%
6	90.10%	89.04%	89.57%
7	90.59%	90.27%	90.43%
8	86.12%	81.24%	83.61%

Digit	Precision	Recall	F1-Score
9	79.39%	86.13%	82.62%

The confusion matrix shows notable confusion between multiple classes, such as 4 with 9 and 5 with 3, highlighting the model's difficulty in separating certain digit features.



4.4.1 Analysis of the `max_depth` Hyperparameter

The impact of maximum depth on model performance was evaluated. Accuracy improved with increasing depth, peaking at **87.96% with a `max_depth` of 15**. Increasing the depth beyond this point did not yield further improvement.

max_depth	Accuracy
5	68.54%
10	86.85%
15	87.96%
20	87.85%
25	87.85%

4.5 Comparative Summary of Results

For direct comparison, the following table summarizes the key performance metrics and computational times for each model evaluated with its initial hyperparameters.

Algorithm	Overall Accuracy	Macro F1-Score	Weighted F1-Score	Training Time (s)	Prediction Time (s)
Logistic Regression	92.26%	92.16%	92.25%	1505.27	0.021
Multi-Layer Perceptron (MLP)	96.21%	96.17%	96.21%	4721.84	0.156
KNN (`k=20`)	96.19%	96.20%	96.19%	4.56	13133.19
Decision Tree (`max_depth=10`)	86.85%	86.72%	86.87%	4416.92	0.082

These quantitative results form the basis for the comparative analysis and discussion presented in the next section of the report.

Of course. Here is the "Discussion" section, written in English, based on your experimental results and the project requirements.

5. Discussion

This section provides a comparative analysis of the results presented in the previous section. The strengths and weaknesses of each algorithm are evaluated, the trade-offs between accuracy and computational efficiency are discussed, common classification errors are analyzed, and the impact of hyperparameters on model performance is examined.

5.1 Overall Performance and Best-Performing Algorithm

When comparing overall accuracy, the **Multi-Layer Perceptron (MLP)** and **K-Nearest Neighbors (KNN)** emerged as the top-performing algorithms. The MLP achieved an accuracy of **96.21%**, while the KNN with `k=20` was very close at **96.19%**. Crucially, upon hyperparameter tuning, the KNN's performance improved further, reaching a peak accuracy of **97.14% at k=3**, making it the most accurate model in this comparison.

The success of these two models can be attributed to their ability to capture the non-linear relationships inherent in the pixel data:

- **MLP**: Its hierarchical layer architecture allows it to learn complex and abstract features from the raw pixel data, ranging from simple edges to complete digit shapes. This makes it intrinsically well-suited for image recognition tasks.
- **KNN**: Its high performance suggests that in the 784-dimensional feature space, digits of the same class tend to cluster closely. Variations in handwriting are subtle enough that the nearest neighbors are a highly reliable indicator of the correct class.

In contrast, **Logistic Regression**, despite being a linear model, provided a strong baseline performance at **92.26%** accuracy. However, its inability to model complex patterns limited its potential. The **Decision Tree** was the lowest performer with an accuracy of **86.85%**, indicating that its method of creating axis-aligned splits based on individual pixel features is not effective for capturing the spatial structure of digits.

5.2 Comparative Analysis of Computational Efficiency

An analysis of training and prediction times reveals a fundamental trade-off between model complexity, accuracy, and efficiency.

Algorithm	Training Time (s)	Prediction Time (s)	Trade-off Analysis
Logistic Regression	1505.27	**0.021**	Good balance, fast at prediction.
MLP	4721.84	0.156	**Slow to train**, but fast at prediction.
KNN (`k=20`)	**4.56**	13133.19	**Fast to train**, but extremely slow at prediction.
Decision Tree (`d=10`)	4416.92	0.082	Slow to train, fast at prediction.

- **Training Time**: KNN is by far the fastest to "train," as its training phase only consists of storing the dataset. The MLP and Decision Tree were the slowest due to the intensive nature of their fitting algorithms (backpropagation and recursive tree construction, respectively).
- **Prediction Time**: Here, the roles are dramatically reversed. **KNN is computationally infeasible for large-scale prediction**, taking over three hours to classify the test set. This is because it must compute the distance from each test point to all 56,000 training samples. In contrast, the other three models are extremely fast at prediction, as they only require matrix operations (Logistic Regression and MLP) or traversing a tree structure (Decision Tree).

This analysis highlights that while KNN can be highly accurate, its inference cost makes it impractical for real-time applications. The **MLP, though expensive to train, offers an excellent balance** of high accuracy and fast prediction, making it an ideal candidate for a deployed system.

5.3 Analysis of Class-Specific Errors and Dataset Challenges

An examination of the confusion matrices for each model reveals consistent error patterns stemming from the visual similarity between certain digits:

- **Most Confused Digits**: The digit pairs that generated the most errors across all models were **(4, 9)**, **(3, 5)**, and **(2, 7)**. For instance, the Decision Tree misclassified 119 instances of the digit 9 as 4.
- **Cause of Confusion**: This difficulty arises because these pairs share structural features. A '4' with a closed top closely resembles a '9'. A '3' and a '5' share similar curves in their central structure. Handwriting variations, such as slant or stroke thickness, accentuate these similarities.

- **Model Performance on Difficult Classes**: The MLP and KNN demonstrated a greater ability to distinguish these difficult digits compared to Logistic Regression and, especially, the Decision Tree. This confirms that non-linear models are better equipped to learn the subtle nuances that differentiate these classes. The KNN's confusion matrix (Appendix C) confirms its strong performance with high values along the diagonal, but its most frequent errors still involve these visually similar digits, such as '2' being confused with '7' and '8' with '1'.

Of course. I have revised section 5.4 to focus on the inherent strengths and weaknesses of each algorithm and created a new, more detailed section 5.5 to analyze the effects of hyperparameters and other interesting phenomena you observed during your experiments.

5.4 Strengths and Weaknesses of Algorithms

The experimental results highlight the distinct characteristics of each algorithm when applied to the MNIST image classification task:

- **Logistic Regression**:
    - **Strengths**: Its primary advantages are simplicity and computational efficiency, especially during prediction (0.021s). It serves as an excellent, easy-to-implement baseline.
    - **Weaknesses**: As a linear model, it is fundamentally limited in its ability to capture the complex, non-linear patterns present in image data. This inherent limitation is the main reason for its lower accuracy compared to the MLP and KNN.
- **Multi-Layer Perceptron (MLP)**:
    - **Strengths**: The MLP's key strength is its capacity to learn hierarchical, non-linear feature representations automatically. This allowed it to achieve a very high accuracy (96.21%) and demonstrate strong generalization. Once trained, its prediction speed is also very fast.
    - **Weaknesses**: It was the most computationally expensive model to train (4721.84s). The significant gap between its training accuracy (99.88%) and test accuracy (96.21%) suggests a tendency to overfit, indicating that further regularization could be beneficial. It also functions as a "black box," making its decision process difficult to interpret.
- **K-Nearest Neighbors (KNN)**:
    - **Strengths**: KNN is conceptually simple and non-parametric, making no assumptions about the underlying data distribution. Its ability to achieve the highest accuracy (97.14% at k=3) demonstrates its effectiveness when data points of the same class are closely clustered in the feature space.
    - **Weaknesses**: The model's most significant drawback is its prohibitive prediction time (over 3 hours for `k=20`), which scales with the size of the training set. It also requires a large amount of memory to store the entire training dataset, making it impractical for resource-constrained or real-time applications.
- **Decision Tree**:
    - **Strengths**: It offers fast prediction times and is highly interpretable, as its decision-making process can be visualized as a series of rules.
    - **Weaknesses**: The Decision Tree was the lowest-performing model. Its reliance on axis-aligned splits on individual features (pixels) is poorly suited for capturing the spatial relationships in image data. This structural limitation prevents it from effectively modeling the visual patterns of digits.

5.5 Hyperparameter Tuning and Model Behavior Analysis

The experiments provided valuable insights into how hyperparameters and preprocessing choices influence model behavior and performance.

5.5.1. Analysis of `k` in KNN

The accuracy of the KNN classifier was highly sensitive to the choice of `k`. Performance peaked at `k=3` **(97.14%)** and gradually decreased as `k` increased. This behavior can be explained as follows:

- **For small `k` (e.g., 1-3)**, the model is highly flexible and adapts to the local structure of the data. This works well for the MNIST dataset, where local clusters are dense and informative. A value of `k=3` provides a good balance, offering more robustness to noise than `k=1` without sacrificing local sensitivity.
- **For large `k` (e.g., 20)**, the decision boundary becomes overly smoothed. The model considers neighbors from a much larger region of the feature space, which can include points from different classes. This introduces a bias, as the classification is influenced by a less specific neighborhood, leading to a decrease in accuracy.

5.5.2. Analysis of `max_depth` in Decision Trees

The performance of the Decision Tree was critically dependent on its `max_depth`. The accuracy showed a steep improvement from **68.54% at `depth=5` to 87.96% at `depth=15`**, after which it plateaued.

- **A shallow depth (e.g., 5)** results in **underfitting**. The tree is too simple to learn the complex rules needed to distinguish between the ten digits.
- **Increasing the depth to 15** allows the model to capture more intricate patterns, finding a better balance between bias and variance.
- **Beyond a depth of 15**, the model begins to **overfit**. It starts learning noise and specific artifacts from the training data that do not generalize to the test set. The plateau in performance indicates that the new splits are no longer capturing meaningful, generalizable patterns, highlighting the inherent limitations of the model's axis-aligned splitting mechanism for this task.

5.5.3. Analysis of KNN Prediction Time and Search Algorithms

An interesting observation was that the custom KD-tree implementation for neighbor search did not reduce prediction time; in fact, it was slightly slower than a brute-force search. This is a classic example of the **"curse of dimensionality."**
KD-trees are efficient in low-dimensional spaces because they can effectively prune large portions of the search space. However, in high-dimensional spaces like MNIST's 784 dimensions, the concept of distance becomes less intuitive, and most points are far from each other. Consequently, a query point is often close to the boundary of many partitions in the KD-tree. To find the true nearest neighbors, the search algorithm must explore a large number of branches, making the overhead of traversing the tree structure greater than the benefit of pruning. In such cases, a brute-force search can be more efficient, which aligns with the experimental results.

5.5.4. Impact of Data Normalization on KNN Performance

Another notable finding was that **mean normalization resulted in significantly better KNN performance than standard normalization** (Z-score scaling). This phenomenon is directly related to how the Euclidean distance metric interacts with feature variance.

- **Mean normalization** centers the data but preserves the original variance of each feature (pixel). Since most pixels in the MNIST dataset share a similar range of values, this method maintains the natural structure of the data.
- **Standard normalization**, on the other hand, scales each feature to have a unit variance. In the MNIST dataset, pixels in the corners have very low variance (they are almost always black), while pixels in the center have high variance. By scaling to unit variance, standard normalization **amplifies the importance of these low-variance, often noisy or uninformative, corner pixels**. This distortion of the feature space misleads the Euclidean distance metric, causing it to give undue weight to irrelevant features and ultimately degrading the classifier's accuracy.

6. Conclusion

This project undertook a comprehensive comparative analysis of four fundamental classification algorithms—Multiclass Logistic Regression, Multi-Layer Perceptron (MLP), K-Nearest Neighbors (KNN), and Decision Tree—implemented from scratch using NumPy and evaluated on the MNIST handwritten digits dataset. The primary objective was to assess not only the predictive accuracy of each model but also their computational efficiency and inherent strengths and weaknesses in a practical, multi-class image classification scenario.

The experimental results revealed that the non-linear models, MLP and KNN, significantly outperformed their linear and tree-based counterparts. The K-Nearest Neighbors classifier, with `k=3`, achieved the highest overall accuracy at **97.14%**, followed closely by the Multi-Layer Perceptron at **96.21%**. In contrast, Logistic Regression provided a respectable baseline accuracy of 92.26%, while the Decision Tree was the least effective model, achieving only 87.96% even with an optimized depth. This outcome underscores the importance of non-linear models for capturing the complex patterns inherent in image data.

A critical finding of this analysis is the stark trade-off between accuracy and computational efficiency, particularly highlighted by the KNN algorithm. Despite its superior accuracy, KNN's prediction time was prohibitively long (over 10,000 seconds), rendering it impractical for real-world, large-scale applications. This inefficiency, attributed to the curse of dimensionality in the 784-feature space, persisted even when attempting to optimize with a KD-tree structure. Conversely, the MLP, while being the most computationally expensive to train, demonstrated excellent prediction speed (0.156 seconds). This positions the MLP as the most practical choice for a deployed system where training is a one-time cost, but fast inference is critical.

The project provided valuable lessons on the suitability of different algorithms for specific data types. The Decision Tree's poor performance confirmed its limitations with spatially structured data, while the success of the MLP and KNN validated their strength in pattern recognition. Furthermore, the analysis of hyperparameters—such as `k` for KNN and `max_depth` for the Decision Tree—demonstrated their profound impact on the bias-variance trade-off and overall model performance. An interesting and practical insight was the superior performance of KNN with mean normalization over standard normalization. This highlighted how preprocessing

choices can dramatically affect distance-based metrics in high-dimensional spaces by either preserving or distorting the feature landscape.

In conclusion, through the hands-on implementation and rigorous evaluation of these four classifiers, this project successfully achieved its objective of providing a deep, practical understanding of their operational characteristics, performance trade-offs, and optimal use cases in the context of multi-class classification.