AI and Machine Learning Project1 Report 纪浩正 12311405

Student: 纪浩正, jihz2023@mail.sustech.edu.cn

1 Dataset

The MNIST dataset, comprising grayscale images of handwritten digits, is utilized in this study. To enhance processing efficiency, the dataset underwent a custom preprocessing pipeline to convert its dense pixel data into a sparse representation.

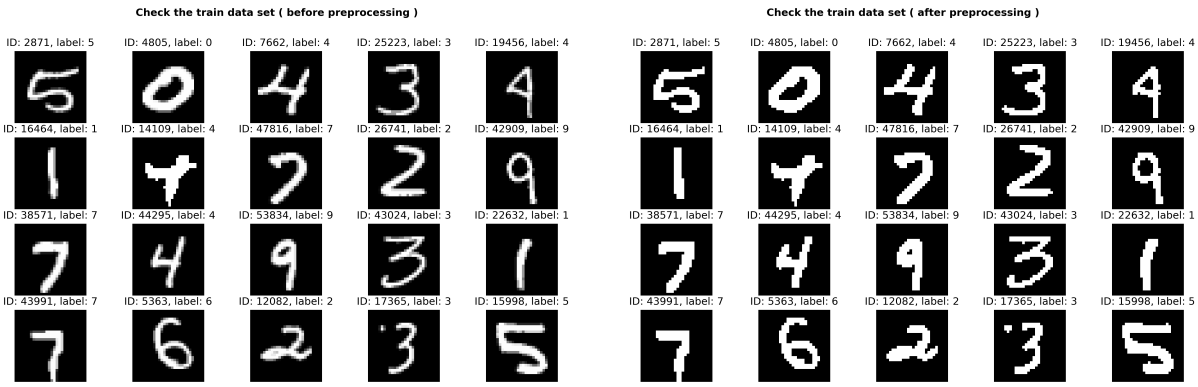
1.1 Preprocessing for Sparse Representation

Each MNIST image is a 28x28 pixel grid. The preprocessing involves two main steps:

1. **Binarization:** For every row of pixels in an image, the average pixel intensity is computed. Pixels with intensities below this average are set to 0, while those above are set to 1. This converts each row into a binary sequence, emphasizing the presence or absence of significant features.

2. **Sparse Interval Encoding:** After binarization, continuous segments of '1's (non-zero intervals) within each row are identified. Each interval is defined by its start and end column indices. To create a fixed-size, sparse representation, each row's information is then compressed into four numerical values: the start and end indices of the first non-zero interval, followed by those of the second non-zero interval. If a row has fewer than two non-zero intervals, the remaining slots are filled with default values (e.g., zeros). This method effectively transforms the dense pixel data into a sparse feature vector for each row.

Figure 1(a) displays a sample MNIST image before preprocessing, while Figure 1(b) shows the same image after binarization, highlighting the transformation to a 0-1 representation.



((a)) A sample MNIST image before preprocessing. **((b)) After binarization.**

Figure 1 Comparison of a sample MNIST image before and after the binarization preprocessing step.

An analysis of the full dataset revealed that this two-interval encoding is highly effective. The distribution of non-zero intervals per row is as follows:

- 0 intervals: 29.50% of rows
- 1 interval: 51.60% of rows
- 2 intervals: 18.16% of rows (cumulative coverage: 99.27% of rows)
- 3 or more intervals: Less than 1% of rows

This demonstrates that using four numbers to represent two intervals covers almost all rows, providing a highly efficient sparse representation. Figure 2 visualizes this distribution.

```
PS C:\Users\20369\Documents\SUSTech\05Programming\AI and Mechine Learn
正在加载 MNIST 训练数据...
训练图像形状: (60000, 28, 28), 训练标签形状: (60000,)
正在加载 MNIST 测试数据...
测试图像形状: (10000, 28, 28), 测试标签形状: (10000,)
图像已二值化 (所有非零像素转换为1)。
训练集和测试集已合并。总图像数: 70000, 图像形状: (28, 28)
总标签数: 70000

--- 正在分析 完整数据集 (训练+测试) (70000 张图像, 28x28 像素) ---
  已处理 7000/70000 张图像...
  已处理 14000/70000 张图像...
  已处理 21000/70000 张图像...
  已处理 28000/70000 张图像...
  已处理 35000/70000 张图像...
  已处理 42000/70000 张图像...
  已处理 49000/70000 张图像...
  已处理 56000/70000 张图像...
  已处理 63000/70000 张图像...
  已处理 70000/70000 张图像...

--- 完整数据集分析结果 ---
完整数据集中单行最大连续非零区间数量: 5
完整数据集非零区间数量分布 (区间数: 出现总行数):
  0 个区间: 578213 行 (29.50%), 累计占比: 29.50%
  1 个区间: 1011413 行 (51.60%), 累计占比: 81.10%
  2 个区间: 355977 行 (18.16%), 累计占比: 99.27%
  3 个区间: 14307 行 (0.73%), 累计占比: 100.00%
  4 个区间: 89 行 (0.00%), 累计占比: 100.00%
  5 个区间: 1 行 (0.00%), 累计占比: 100.00%
要覆盖 100% 的 完整数据集 行, 最大区间数应设置为: 3
因此, 为 完整数据集, 建议 N_gap 设置为: 3

最终选择的 N_gap 值 (基于 99.9% 覆盖率): 3
(注意: 如果选择 N_gap = 5 将确保不丢失任何区间信息)
```

Figure 2 Distribution of non-zero intervals per row across the MNIST dataset, validating the two-interval sparse encoding strategy.

The specific rules for storing this sparse matrix representation, where each row is converted into

a feature vector of four values, are illustrated in Figure 3. This compact format significantly reduces memory footprint and potentially speeds up distance calculations.

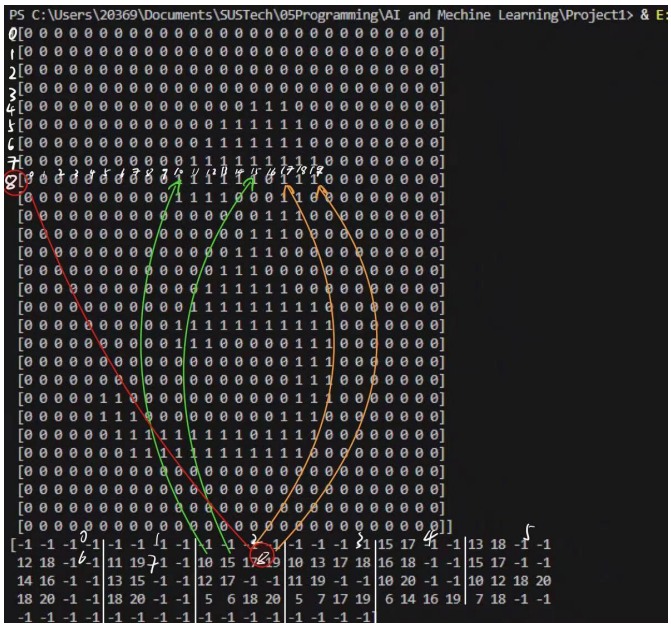


Figure 3 Illustration of the sparse matrix storage rules. Each image row is encoded into four values representing the start and end indices of its first two non-zero intervals.

1.2 Dataset Split and Class Distribution

The full MNIST dataset was partitioned into training and test sets using an 80%/20% split, ensuring a representative distribution of each digit class in both partitions. The breakdown of samples for each digit label is provided below:

The distribution of these digit labels across the training and test sets is visually represented in Figure 4.

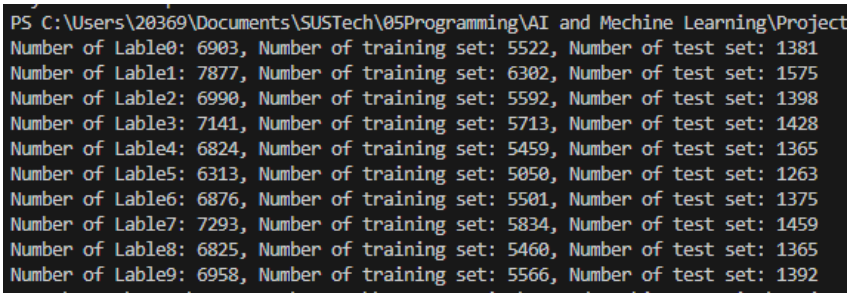


Figure 4 Distribution of samples per digit label, showing the counts in the training and test sets.

Table 1 Distribution of MNIST samples across digit labels for training and test sets.

Label	Total Samples	Training Samples	Test Samples
0	6,903	5,522	1,381
1	7,877	6,302	1,575
2	6,990	5,592	1,398
3	7,141	5,713	1,428
4	6,824	5,459	1,365
5	6,313	5,050	1,263
6	6,876	5,501	1,375
7	7,293	5,834	1,459
8	6,825	5,460	1,365
9	6,958	5,566	1,392

2 Algorithm Implementation

2.1 Multiclass Logistic Regression

Multiclass Logistic Regression (MLR) is a generalized version of binary logistic regression that handles classification tasks with more than two classes. In this project, the algorithm is implemented using the softmax activation function and trained using gradient descent. To prevent overfitting, L2 regularization is incorporated into the loss function.

2.1.1 Softmax Function

The softmax function converts raw class scores into normalized probability distributions. For an input vector $z \in \mathbb{R}^K$, the softmax output is defined as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

For numerical stability, the maximum value in each row is subtracted before exponentiation.

```
def _f(self,z):
    z=z-np.max(z,axis=1, keepdims=True)
    return np.exp(z)/np.sum(np.exp(z),axis=1, keepdims=True)
```

2.1.2 L2 Regularization

To reduce overfitting and improve generalization, L2 regularization is applied to the weight matrix:

$$L_{\text{reg}} = \frac{\lambda}{2}\|\mathbf{W}\|^2$$

This term is added to the cross-entropy loss to penalize large weight values.

```
        epsilon=1e-8
        loss = -np.sum(onehot * np.log(self._f(X_norm @ self.w) + epsilon))
        loss += (self.reg_lam / 2) * np.sum(self.w * self.w)

        losses.append(loss)
```

2.1.3 Gradient Descent Optimization

Model parameters are optimized using batch-based gradient descent. The gradient of the loss with respect to the weight matrix is computed as:

$$\nabla_{\mathbf{W}} = -\mathbf{X}^\top(\mathbf{Y}_{\text{onehot}} - \hat{\mathbf{Y}})$$

The weights are updated at each step by:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta\nabla_{\mathbf{W}}$$

```
def fit(
        self,X,Y,epochs=100,lr=0.01,batch_size=32,normalization=True):
    self.X=X.astype(np.float32)
    self.Y=Y
    self.normalization=normalization
    self.nsample,self.ndim=X.shape
    self.nclass=np.max(Y)+1
    X_norm=self.normalize(X)
    onehot=np.zeros((self.nsample,self.nclass))
    onehot[np.arange(self.nsample),Y]=1
    b=np.ones((self.nsample,1))
    X_norm=np.hstack((b,X_norm))

    self.w=np.zeros((self.ndim+1,self.nclass))

    losses=[]

    for epoch in range(epochs):
```

```
            idx=np.arange(self.nsample)
            np.random.shuffle(idx)

            for s_idx in range(0,self.nsample,batch_size):
                e_idx=s_idx+batch_size
                batch_idx=idx[s_idx:e_idx]
                X_batch=X_norm[batch_idx]
                Y_batch=onehot[batch_idx]
                y_pred=self._f(X_batch@self.w)
                dw=-X_batch.T@(Y_batch-y_pred)
                self.w=self.w-lr*dw
            epsilon=1e-8
            loss = -np.sum(onehot * np.log(self._f(X_norm @ self.w) + epsilon))
            loss += (self.reg_lam / 2) * np.sum(self.w * self.w)

            losses.append(loss)

        return losses,self.w
```

2.1.4 Prediction Function

The prediction function is used to generate class labels for new input samples after the training phase. To ensure consistency with the training process, the input data are first normalized using the same mean and standard deviation learned earlier. A bias term is then added to the feature matrix, and the model computes class probabilities by applying the softmax function to the linear transformation$\mathbf{XW}$.

The predicted class for each sample corresponds to the index with the highest softmax probability.

```
def predict(self, X_pred):
X_norm = self._normalize(X_pred)
b = np.ones((X_norm.shape[0], 1))
X_norm = np.hstack((b, X_norm))
y_pred = self._f(X_norm @ self.w)
return np.argmax(y_pred, axis=1)
```

2.1.5 Accuracy Evaluation

The accuracy evaluation function provides a quantitative measure of the model's performance. It compares the predicted class labels with the true labels and computes the overall classification accuracy. In addition, the function returns the indices of correctly and incorrectly classified

samples. This information is helpful for analyzing the model's strengths and weaknesses, especially in cases of class imbalance or ambiguous samples.

```
def accuracy(self, x, real):
pred = self.predict(x)
accuracy = np.mean(pred == real) * 100
true_idx = np.where(pred == real)[0]
wrong_idx = np.where(pred != real)[0]
return accuracy, pred, real, true_idx, wrong_idx
```

2.2 Multi-Layer Perceptron (MLP)

The Multi-Layer Perceptron (MLP) is a feedforward neural network. It consists of an input layer, one or more hidden layers, and an output layer. Our implementation designs a modular `Layer` class. Hidden layers use ReLU activation, and the output layer uses Softmax for multi-class classification. The network is trained with cross-entropy loss and L2 regularization using mini-batch gradient descent.

2.2.1 Layer Structure and Forward Propagation

Each layer performs a linear transformation followed by an activation function. Weights ($\mathbf{W}$) are initialized using He initialization ($\mathcal{N}(0, \sqrt{2/\text{input\_dim}})$), and biases ($\mathbf{b}$) are zero. For a layer with input $\mathbf{X}$:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{b}$$

$$\mathbf{A} = \text{activation}(\mathbf{Z})$$

```
class Layer:
    def __init__(self,input_dim,output_dim,activation='relu'):
        self.input_dim=input_dim
        self.output_dim=output_dim
        self.activation=activation

        self.W=np.random.randn(input_dim,output_dim)*np.sqrt(2./input_dim)
        self.b=np.zeros((1,output_dim))

        self.X=None
        self.Z=None
        self.A=None

        self.dw=None
        self.db=None
    def forward(self,X):
```

```
        self.X=X
        self.Z=X@self.W+self.b
        self.A=self._activate(self.Z)
        return self.A
```

Activation Functions:

- **ReLU (Hidden Layers):** $\text{ReLU}(z) = \max(0, z)$
- **Softmax (Output Layer):** $\text{Softmax}(\mathbf{z})_j = \frac{e^{z_j - \max(\mathbf{z})}}{\sum_k e^{z_k - \max(\mathbf{z})}}$

```
    def _activate(self,Z):
        if self.activation=='relu':
            return np.maximum(0,Z)
        elif self.activation=='softmax':
            Z_exp=np.exp(Z-np.max(Z,axis=1,keepdims=True))
            return Z_exp/np.sum(Z_exp,axis=1,keepdims=True)
```

2.2.2 Backward Propagation and Parameter Updates

Error gradients (dA) are propagated backward through layers. For each layer, gradients of the loss with respect to weights (**W**) and biases (**b**) are computed, then parameters are updated. Given $d\mathbf{Z}$ (gradient with respect to the linear output):

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{1}{N} \mathbf{X}^\top d\mathbf{Z}$$

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{1}{N} \sum_{\text{batch}} d\mathbf{Z}$$

The gradient passed to the previous layer is: $\frac{\partial L}{\partial \mathbf{X}} = d\mathbf{Z}\mathbf{W}^\top$. Parameters are updated using learning rate η:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \cdot \frac{\partial L}{\partial \mathbf{W}}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \eta \cdot \frac{\partial L}{\partial \mathbf{b}}$$

```
    def backward(self,dA,lr):
        if self.activation=='relu':
            dZ=dA.copy()
            dZ[self.Z<=0]=0
        elif self.activation=='softmax':
            dZ=dA
        self.dW=self.X.T@dZ/self.X.shape[0]
        self.db=np.sum(dZ,axis=0,keepdims=True)/self.X.shape[0]
```

```
        dX=dZ@self.W.T

        self.W-=lr*self.dW
        self.b-=lr*self.db

        return dX
```

2.2.3 Loss Function and Training

The network is trained to minimize the combined Cross-Entropy loss and L2 Regularization. Input features are optionally normalized (Z-score standardization).

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} Y_{\text{onehot},ij}\log\left(\hat{Y}_{ij}+\epsilon\right) + \frac{\lambda}{2}\sum_{\text{all layers}}\sum_{p,q} W_{pq}^2$$

Training proceeds via mini-batch gradient descent. The initial gradient for the output layer's backpropagation is:

$$dA_{\text{output}} = (\hat{Y} - Y_{\text{onehot}})/\text{batch\_size}$$

```
class MultiLayerPerceptron:
    def __init__(self,layer_dims,reg_lambda=0.001,normalization=True):
        self.layers=[]
        self.n_layers=len(layer_dims)-1
        self.reg_lam=reg_lambda
        self.normalizaiton=normalization
        self.mean=None
        self.std=None

        for ii in range(len(layer_dims)-1):
            act='softmax' if ii == len(layer_dims)-2 else 'relu'
            self.layers.append(Layer(layer_dims[ii],layer_dims[ii+1],activation=act))

    def fit(self,X,Y,epochs=100,lr=0.01,batch_size=128):
        X=self.normalize(X)
        nsample=X.shape[0]
        nclass=np.max(Y)+1
        onehot=np.zeros((nsample,nclass))
        onehot[np.arange(nsample),Y]=1
        losses=[]

        for epoch in range(epochs):
            idx=np.arange(nsample)
            np.random.shuffle(idx)
```

```
            for s_idx in range(0,nsample,batch_size):
                e_idx=s_idx+batch_size
                X_batch=X[s_idx:e_idx]
                Y_batch=onehot[s_idx:e_idx]

                Y_pred=self.forward(X_batch)
                self.backward(Y_pred,Y_batch,lr)

            Y_pred_full=self.forward(X)
            loss=self.cross_entropy_loss(Y_pred_full,onehot)
            losses.append(loss)
        return losses

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

2.3 K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a non-parametric, instance-based learning algorithm used for classification and regression. In this implementation, KNN is used for classification, where a data point is classified by a majority vote of its k nearest neighbors.

2.3.1 Data Preprocessing (Normalization)

Upon initialization, the training data $\mathbf{X}_{\text{train}}$ can be optionally normalized using Z-score standardization. This is done by subtracting the mean (μ) and dividing by the standard deviation (σ) for each feature.

$$X_{\text{norm},ij} = \frac{X_{ij} - \mu_j}{\sigma_j}$$

Normalization parameters (μ and σ) are learned from the training set and applied to query points during prediction.

```
class KnnClassifier:
    def __init__(self,train_set,train_label,use_kdtree=False,normalization=False,lead_size=1):
        self.label=train_label
        self.num=len(train_label)
        self.mode=use_kdtree
        self.X_train=train_set.astype(np.float32)
        self.mean=self.X_train.mean(axis=0)
        self.std=self.X_train.std(axis=0)
        self.normalization=normalization
        if normalization:
```

```
        for ii in range(len(self.std)):
            if self.std[ii]==0:
                self.X_train[:,ii]=0
            else:
                self.X_train[:,ii]=(self.X_train[:,ii]-self.mean[ii])/self.std[ii]
    self.kd_tree=None

    if self.mode:
        self.kd_tree=KdTree(self.X_train,self.label)
```

2.3.2 Distance Metric

The Euclidean distance is used to measure the similarity between data points. For two points $\mathbf{x}_1 = (x_{1,1}, \ldots, x_{1,D})$ and $\mathbf{x}_2 = (x_{2,1}, \ldots, x_{2,D})$, the squared Euclidean distance is calculated as:

$$d(\mathbf{x}_1, \mathbf{x}_2)^2 = \sum_{j=1}^{D} (x_{1,j} - x_{2,j})^2$$

```
def _Euclidean_distance(self,x):
    diff=self.X_train-x
    diff2=diff**2
    dist=np.sum(diff2,axis=1)
    return dist
```

2.3.3 Nearest Neighbor Search

The classifier supports two methods for finding k nearest neighbors:

1. **Brute-force Search:** For each query point, the Euclidean distance to all training samples is computed, and the indices of the k smallest distances are selected.

   ```
   def get_nearest(self,x,k):
       dis_list=self._Euclidean_distance(x)
       k_nearest_idx=np.argsort(dis_list)[:k]
       return k_nearest_idx
   ```

2. **Kd-Tree Optimized Search:** When ‘use$_kdtree‘isenabled, aKd-Treedatastructureisusedtoacceler$

2.3.4 Prediction

For a given query point, the k nearest neighbors' labels are retrieved. The final prediction is determined by a majority vote among these k labels using 'collections.Counter'.

```
def pred(self,x,k):
    N=x.shape[0]
    x_copy = x.copy().astype(np.float32)
    if self.normalization:
        for ii in range(len(self.std)):
            if self.std[ii] == 0:
                x_copy[:, ii] = 0
            else:
                x_copy[:, ii] = (x_copy[:, ii] - self.mean[ii]) / self.std[ii]
    pred=[]
    for ii in range(N):
        test=x_copy[ii,:]
        if self.mode:
            k_nearest_label=self.kd_tree.query(test,k)
            pred.append(Counter(k_nearest_label).most_common(1)[0][0])
        else:
            k_nearest_idx=self.get_nearest(test,k)
            k_nearest_label=self.label[k_nearest_idx]
            pred.append(Counter(k_nearest_label).most_common(1)[0][0])
    return np.array(pred)

def accuracy(self,x,real,k):
    pred=self.pred(x,k)
    accuracy=np.mean(pred==real)*100
    true_idx=np.where(pred==real)[0]
    wrong_idx=np.where(pred!=real)[0]
    return accuracy,pred,real,true_idx,wrong_idx
```

2.3.5 Kd-Tree Implementation

A Kd-Tree (k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. It is used to perform range searches and nearest neighbor searches more efficiently than a brute-force approach.

Tree Construction (build tree): The tree is built recursively. At each node, a splitting axis is chosen . The training data points are sorted along this axis, and the median point becomes the current node. Points smaller than the median go to the left subtree, and larger points go to the right subtree.

```
class KdTree:
    def __init__(self, data, labels):
        if data.shape[0] != len(labels):
            raise ValueError("Data and labels must have the same number of samples.")
        self.data = data
        self.labels = labels
        self.dim = data.shape[1]
        self.root = self._build_tree(list(range(len(data))))

    def _build_tree(self, indices, depth=0):
        if not indices:
            return None

        axis = depth % self.dim

        indices.sort(key=lambda i: self.data[i, axis])
        median_idx = len(indices) // 2
        median_point_idx = indices[median_idx]

        node = KdNode(
            point=self.data[median_point_idx],
            label=self.labels[median_point_idx],
            axis=axis
        )
        node.left = self._build_tree(indices[:median_idx], depth + 1)
        node.right = self._build_tree(indices[median_idx + 1:], depth + 1)

        return node
```

Nearest Neighbor Query (query): To find the k nearest neighbors for a query point, the search function is called recursively. It maintains a min-priority queue (implemented using ‘heapq‘ as a max-heap of negative distances) to store the k nearest neighbors found so far. The search preferentially explores the subtree on the side of the query point. After exploring the closer subtree, it checks if the hyperplane (defined by the node’s splitting axis and value) is closer to the query point than the current k-th nearest neighbor. If so, the further subtree is also explored. This pruning strategy helps avoid unnecessary computations.

```
    def query(self, query_point, k=1):
        if self.root is None:
            return []

        self.k_nearest = []

        def search(node):
```

```
        if node is None:
            return

        dist = self._distance(query_point, node.point)
        if len(self.k_nearest) < k:
            heapq.heappush(self.k_nearest, (-dist, node.label))
        elif dist < -self.k_nearest[0][0]:
            heapq.heapreplace(self.k_nearest, (-dist, node.label))
        axis = node.axis
        if query_point[axis] < node.point[axis]:
            closer_subtree = node.left
            further_subtree = node.right
        else:
            closer_subtree = node.right
            further_subtree = node.left
        search(closer_subtree)
        hyperplane_dist_sq = (query_point[axis] - node.point[axis])**2

        if len(self.k_nearest) < k or hyperplane_dist_sq < -self.k_nearest[0][0]:
            search(further_subtree)

    search(self.root)
    return [label for neg_dist, label in sorted(self.k_nearest, key=lambda x: -x[0])]
```

2.4 Decision Tree

A Decision Tree is a non-parametric supervised learning method used for classification and regression. It constructs a model in the form of a tree structure, where internal nodes represent tests on attributes, branches represent outcomes of the test, and leaf nodes represent class labels. This implementation focuses on classification using entropy and information gain.

2.4.1 Tree Node Structure

The ‘TreeNode‘ class defines the components of each node in the decision tree. An internal node stores the ‘feature idx‘ (index of the feature to split on) and ‘threshold‘ (the value to split on), along with references to its ‘left‘ and ‘right‘ child nodes. A leaf node stores a ‘value‘, which represents the predicted class label for that branch.

```
class TreeNode:
    def __init__(self, feature_idx=None, threshold=None, left=None, right=None, *, value=None):
        self.feature_idx = feature_idx
        self.threshold = threshold
        self.left = left
        self.right = right
```

```
        self.value = value
```

2.4.2 Entropy and Information Gain

The decision tree uses Information Gain based on Entropy to determine the best split at each node. Entropy measures the impurity or randomness of a set of labels y.

$$H(y) = -\sum_{c=1}^{C} p_c \log_2(p_c)$$

where p_c is the proportion of samples belonging to class c.

Information Gain is the reduction in entropy achieved by splitting the data based on a particular feature and threshold.

$$\text{IG}(y, \text{split}) = H(y) - \left(\frac{|y_{\text{left}}|}{|y|} H(y_{\text{left}}) + \frac{|y_{\text{right}}|}{|y|} H(y_{\text{right}}) \right)$$

The algorithm searches for the split (feature and threshold) that maximizes this Information Gain.

```
    def _entropy(self, y):
        hist = np.bincount(y)
        ps = hist / len(y)
        ps = ps[ps > 0]
        return -np.sum(ps * np.log2(ps))

    def _information_gain(self, y, y_left, y_right):
        H = self._entropy(y)
        H_left = self._entropy(y_left)
        H_right = self._entropy(y_right)
        p_left = len(y_left) / len(y)
        p_right = len(y_right) / len(y)
        return H - (p_left * H_left + p_right * H_right)
```

2.4.3 Recursive Tree Construction

The ‘build tree‘ method recursively constructs the decision tree. At each step, it finds the ‘best split‘ that yields the highest information gain.

1. **Splitting Criteria:** For each feature, it considers all unique values as potential thresholds for splitting. For continuous features, this implicitly handles them by considering discrete threshold points.

2. **Stopping Conditions (Regularization):** The recursion stops and a leaf node is created if:

   - All samples in the current node belong to the same class.
   - The number of samples in the current node is less than ‘min samples split‘.
   - The ‘max depth‘ limit is reached.

3. **Leaf Node Value:** A leaf node’s ‘value‘ is determined by the majority class of the samples reaching that node.

```
class DecisionTree:
    def __init__(self, max_depth=None, min_samples_split=2):
        self.root = None
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split

    def _best_split(self, X, y):
        n_samples, n_features = X.shape
        best_gain = -1
        split_idx, split_thr = None, None

        for feature_idx in range(n_features):
            thresholds = np.unique(X[:, feature_idx])
            for thr in thresholds:
                left_idx = X[:, feature_idx] <= thr
                right_idx = X[:, feature_idx] > thr
                if np.sum(left_idx) == 0 or np.sum(right_idx) == 0:
                    continue
                y_left, y_right = y[left_idx], y[right_idx]
                gain = self._information_gain(y, y_left, y_right)
                if gain > best_gain:
                    best_gain = gain
                    split_idx = feature_idx
                    split_thr = thr
        return split_idx, split_thr, best_gain

    def _build_tree(self, X, y, depth=0):
        n_samples = X.shape[0]
        n_labels = len(np.unique(y))

        if n_labels == 1 or n_samples < self.min_samples_split or (self.max_depth is not None
            and depth >= self.max_depth):
            leaf_value = np.bincount(y).argmax()
            return TreeNode(value=leaf_value)
```

```
    feature_idx, threshold, gain = self._best_split(X, y)
    if gain == -1:
        leaf_value = np.bincount(y).argmax()
        return TreeNode(value=leaf_value)

    left_idx = X[:, feature_idx] <= threshold
    right_idx = X[:, feature_idx] > threshold
    left = self._build_tree(X[left_idx], y[left_idx], depth + 1)
    right = self._build_tree(X[right_idx], y[right_idx], depth + 1)
    return TreeNode(feature_idx, threshold, left, right)
```

2.4.4 Prediction

To predict the class of a new sample, it traverses the tree from the root. At each internal node, it compares the sample's feature value with the 'threshold' and moves to the left or right child accordingly. This continues until a leaf node is reached, whose 'value' is returned as the prediction.

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

def predict(self, X):
    return np.array([self._predict_sample(x, self.root) for x in X])

def accuracy(self, X, real):
    pred = self.predict(X)
    accuracy = np.mean(pred == real) * 100
    true_idx = np.where(pred == real)[0]
    wrong_idx = np.where(pred != real)[0]
    return accuracy, pred, real, true_idx, wrong_idx
```

3 Evaluation Metrics

3.1 Overall Accuracy

Measures the proportion of correctly classified instances out of the total number of instances.

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

While simple, accuracy can be misleading in datasets with imbalanced class distributions.

3.2 Per-class Precision, Recall, and F1-score

These metrics are calculated for each individual class and provide a detailed view of classifier performance, especially useful for multiclass problems. For a specific class C:

- **True Positives (TP):** Instances correctly predicted as C.
- **False Positives (FP):** Instances incorrectly predicted as C (actually belong to another class).
- **False Negatives (FN):** Instances that are actually C but predicted as another class.

Precision (P_C): The proportion of positive identifications that were actually correct for class C.

$$P_C = \frac{\text{TP}_C}{\text{TP}_C + \text{FP}_C}$$

Recall (R_C, Sensitivity): The proportion of actual positives for class C that were identified correctly.

$$R_C = \frac{\text{TP}_C}{\text{TP}_C + \text{FN}_C}$$

F1-score ($F1_C$): The harmonic mean of Precision and Recall for class C, providing a balance between them.

$$F1_C = 2 \cdot \frac{P_C \cdot R_C}{P_C + R_C}$$

3.3 Macro-averaged F1-score

Calculated as the unweighted mean of the F1-scores for each class. It treats all classes equally, regardless of their size, and is useful for evaluating performance on imbalanced datasets when all classes are considered equally important.

$$\text{F1-score}_{\text{macro}} = \frac{1}{N_{\text{classes}}} \sum_{C=1}^{N_{\text{classes}}} \text{F1-score}_C$$

3.4 Weighted-average F1-score

Computed as the average of the F1-scores for each class, weighted by the number of true instances (support) for each class. This metric accounts for class imbalance by giving more weight to larger classes, making it a good choice when the importance of classes is propor-

tional to their prevalence.

$$\text{F1-score}_{\text{weighted}} = \sum_{C=1}^{N_{\text{classes}}} (\text{F1-score}_C \cdot \frac{\text{Support}_C}{\text{Total Samples}})$$

3.5 Confusion Matrix

A table used to describe the performance of a classification model. Each row represents the instances in an actual class, while each column represents the instances in a predicted class. The cells contain counts, visually revealing misclassifications between specific classes. It provides a detailed breakdown of correct and incorrect predictions for each class.

4 Results show

4.1 Multiclass Logistic Regression

Training Performance Across Epochs

Epochs	Accuracy (%)	Training Time (s)
0	9.8636	0.05
100	84.3511	5.68
200	84.4440	17.60
300	84.2868	23.10
400	84.2940	23.79
500	84.3011	30.08
600	84.3511	37.42
700	84.3725	46.45
800	84.4440	50.47
900	84.3654	56.27
1000	84.4225	62.25

Table 2 **Accuracy and training time under different epoch settings. The best model is obtained at epoch 200.**

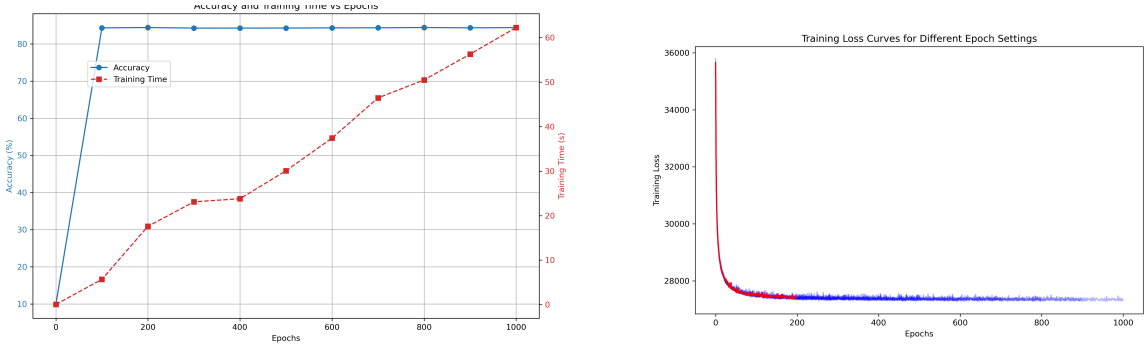


Figure 5 **Left: Test accuracy and training time versus number of epochs. Accuracy saturates after around 200 epochs. Right: Training loss curves for different epoch settings. Loss decreases quickly in the first 200 epochs and stabilizes afterward.**

Evaluation Metrics for Best Epoch (Epoch 200)

Class	Precision	Recall	F1-score
0	0.9415	0.9551	0.9482
1	0.9017	0.9549	0.9275
2	0.8647	0.8319	0.8480
3	0.7637	0.7241	0.7434
4	0.8819	0.8696	0.8757
5	0.7156	0.6991	0.7072
6	0.9210	0.9244	0.9227
7	0.8487	0.8382	0.8434
8	0.8230	0.8007	0.8117
9	0.7620	0.8233	0.7914
Macro F1 = 0.8419,		Weighted F1 = 0.8439	

Table 3 Per-class precision, recall, and F1-score for the best model (epoch 200).

Visualization of Results

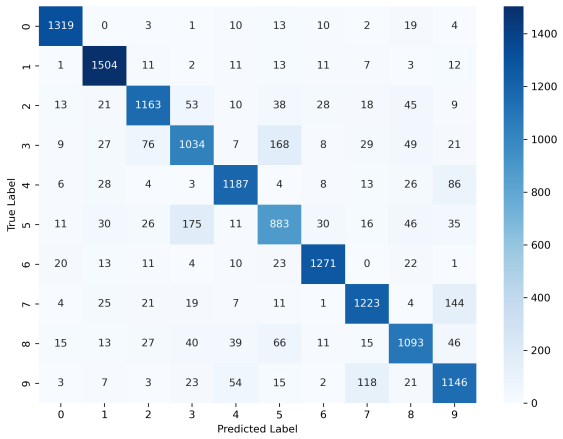


Figure 6 Confusion matrix for the best MLR model at epoch 200.

The MLR model achieves the best performance at epoch 200 with a test accuracy of 84.44%. Beyond this epoch, accuracy gains are minimal, while training time increases significantly. Per-class metrics indicate strong performance on most classes, with slightly lower F1-scores for classes 3, 5, and 9.

4.2 Multi-Layer Perceptron (MLP)

This section presents the performance of Multi-Layer Perceptron (MLP) models with different architectures and training epochs. The MLP models were trained with one to two hidden layers using ReLU activation and a Softmax output layer. Cross-entropy loss with L2 regularization was applied.

Test Accuracy Across Architectures and Epochs

Layers	Epochs	Test Accuracy (%)	Training Time (s)
112, 64, 10	100	24.63	20.57
112, 128, 10	100	18.66	34.61
112, 128, 64, 10	100	19.97	46.08
112, 64, 10	500	52.50	116.47
112, 128, 10	500	59.11	145.86
112, 128, 64, 10	500	59.90	217.73
112, 64, 10	1000	67.64	231.25
112, 128, 10	1000	71.63	321.70
112, 128, 64, 10	1000	72.02	457.29

Table 4 Test accuracy and training time for different MLP architectures and epochs. The best model is obtained with layers [112, 128, 64, 10] at 1000 epochs.

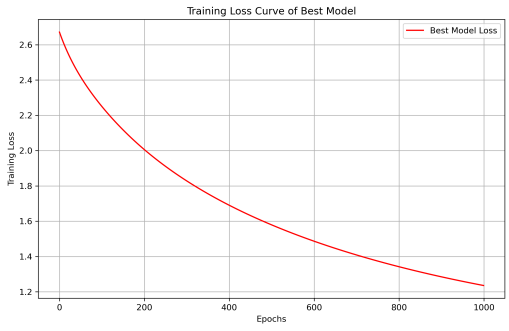


Figure 7 Training loss curve of the best MLP model. Loss decreases steadily and stabilizes towards the end of training.

Evaluation Metrics of the Best Model

Class	Precision	Recall	F1-score
0	0.8414	0.9146	0.8765
1	0.8572	0.8806	0.8688
2	0.6574	0.6917	0.6741
3	0.5782	0.5875	0.5828
4	0.7258	0.7971	0.7598
5	0.5342	0.2043	0.2955
6	0.8529	0.9069	0.8791
7	0.6978	0.7423	0.7194
8	0.6759	0.7136	0.6942
9	0.6415	0.7019	0.6703

Table 5 Per-class precision, recall, and F1-score of the best MLP model. Macro-averaged F1-score: 0.7021; Weighted-average F1-score: 0.7073.

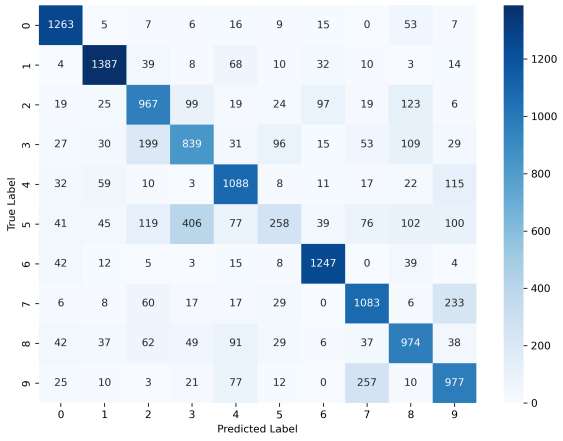


Figure 8 Confusion matrix of the best MLP model at 1000 epochs. Most classes are correctly classified with minor misclassifications in difficult classes.

The best MLP model achieved a test accuracy of 72.02% with the architecture [112, 128, 64, 10] trained for 1000 epochs. Adding more layers and increasing epochs improves performance but also significantly increases training time. The per-class evaluation shows good classification for most digits, though some classes (e.g., class 5) remain challenging. Training loss converges smoothly, indicating effective learning and optimization.

4.3 K-Nearest Neighbors (KNN)

The K-Nearest Neighbors (KNN) algorithm was evaluated under two main experiments: the impact of different k values on accuracy and training/testing time, and the effect of training size on performance with the optimal k. Normalization was applied in some configurations to observe its effect.

Experiment 1: Effect of K-value on Accuracy and Time

K	KdTree_NoNorm	KdTree_WithNorm	BruteForce_NoNorm	BruteForce_WithNorm
1	82.00% (1.738s)	84.50% (1.700s)	82.00% (0.092s)	84.50% (0.090s)
2	82.00% (1.739s)	84.50% (1.693s)	82.00% (0.089s)	84.50% (0.097s)
3	85.50% (1.814s)	88.00% (1.821s)	85.50% (0.095s)	88.00% (0.100s)
4	86.50% (1.818s)	87.50% (1.852s)	86.50% (0.123s)	87.50% (0.118s)
5	84.00% (1.884s)	87.00% (1.937s)	84.00% (0.106s)	87.00% (0.101s)
6	83.00% (1.955s)	85.50% (1.908s)	83.00% (0.094s)	85.50% (0.101s)
7	85.50% (1.788s)	86.50% (1.748s)	85.50% (0.093s)	86.50% (0.093s)
8	85.00% (1.737s)	86.50% (1.717s)	85.00% (0.091s)	86.50% (0.094s)
9	83.50% (1.744s)	86.50% (1.736s)	83.50% (0.093s)	86.50% (0.093s)

Table 6 Accuracy and time for different k values under various KNN configurations. Best performance achieved with BruteForce_WithNorm at $k = 3$.

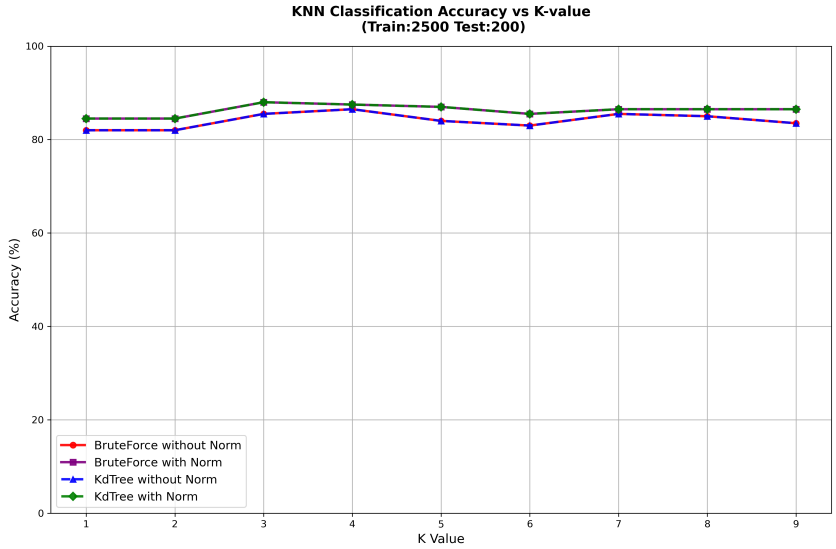


Figure 9 **Accuracy comparison across different k values for all KNN configurations. Normalization improves performance slightly.**

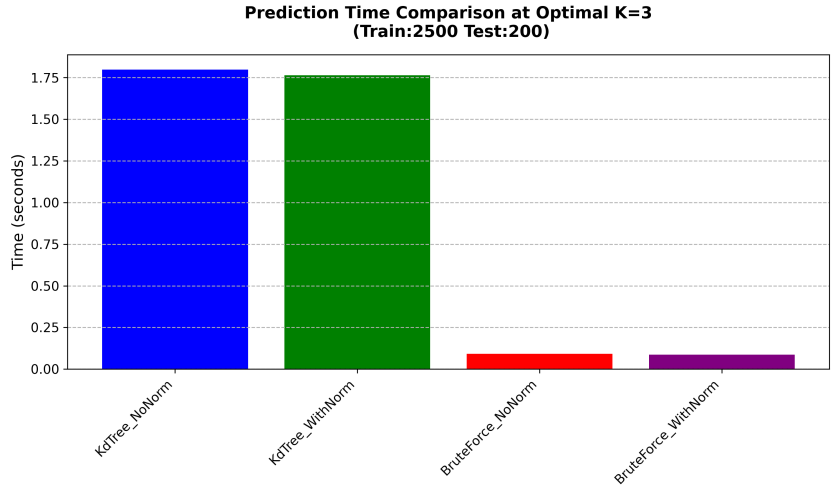


Figure 10 **Time comparison for different KNN configurations at optimal $k = 3$.**

Experiment 2: Effect of Training Size on Accuracy and Time (Fixed K=3)

Training Size	Accuracy (%)	Time (s)
200	75.00	0.0045
2000	86.50	0.0363
5000	89.00	0.1989
10000	90.50	0.4000

Table 7 **Effect of training size on accuracy and prediction time for BruteForce_WithNorm with $k = 3$. Larger training sets improve accuracy but increase computation time.**

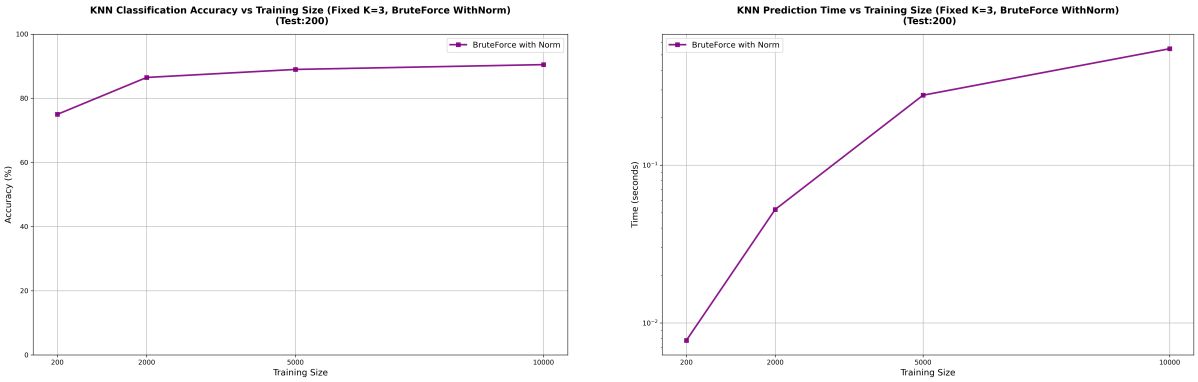


Figure 11 Left: Accuracy increases with larger training sizes. Right: Prediction time increases with larger training sizes.

Detailed Evaluation of Best Model

The best configuration is BruteForce_WithNorm with $k = 3$. The overall accuracy achieved is 88.00%. Per-class metrics indicate most classes are classified well, with slightly lower performance for class 2.

Class	Precision	Recall	F1-score
0	0.9286	1.0000	0.9630
1	0.8696	1.0000	0.9302
2	0.8571	0.6000	0.7059
3	1.0000	0.7895	0.8824
4	0.8182	0.8571	0.8372
5	0.8571	1.0000	0.9231
6	0.9600	0.9600	0.9600
7	0.8800	0.8800	0.8800
8	0.8400	0.9130	0.8750
9	0.8261	0.8636	0.8444

Table 8 Per-class evaluation metrics of the best KNN model. Macro F1-score: 0.8801; Weighted F1-score: 0.8768.

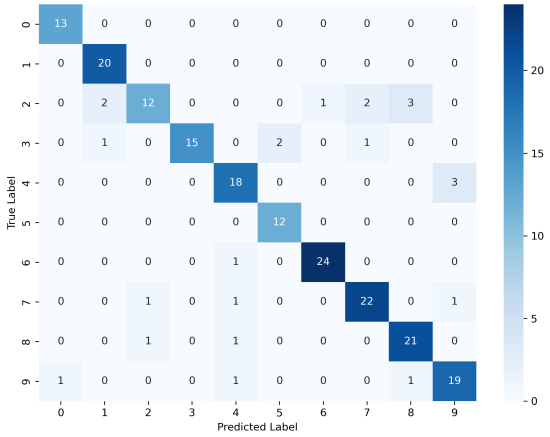


Figure 12 Confusion matrix of BruteForce_WithNorm at $k = 3$. Most misclassifications occur in classes 2 and 4.

Experiment 1 shows that $k = 3$ yields the best performance for BruteForce_WithNorm. Experiment 2 demonstrates that increasing the training size improves accuracy at the cost of increased computation time. The final best model achieves 88% test accuracy with strong per-class performance, validating the effectiveness of normalization and an optimal k selection.

4.4 Decision Tree Classifier

Decision Tree classifiers were trained with varying maximum depths to evaluate their impact on test accuracy and training time. The best performing model was selected based on test accuracy.

Experiment: Effect of Maximum Depth

Max Depth	Accuracy (%)	Training Time (s)
2	36.25	2.09
5	73.34	5.55
10	88.94	19.01
15	90.76	36.34
None	90.72	52.39

Table 9 Test accuracy and training time for Decision Tree classifiers with different maximum depths. The best performance is achieved at max_depth=15.

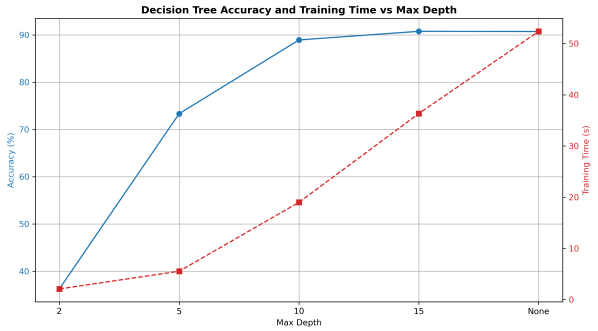


Figure 13 Test accuracy and training time versus max depth. Accuracy improves with depth but saturates beyond 15, while training time increases.

Detailed Evaluation of Best Model

The Decision Tree with max_depth=15 achieved the highest test accuracy of 90.76%. Per-class metrics are summarized below:

Class	Precision	Recall	F1-score
0	0.9582	0.9638	0.9610
1	0.9369	0.9613	0.9489
2	0.9036	0.8984	0.9010
3	0.8770	0.8536	0.8652
4	0.9001	0.8982	0.8992
5	0.8686	0.8424	0.8553
6	0.9481	0.9433	0.9457
7	0.9362	0.9150	0.9255
8	0.8743	0.8864	0.8803
9	0.8656	0.9023	0.8836

Table 10 Per-class evaluation metrics of the best Decision Tree model (max_depth=15). Macro F1-score: 0.9066; Weighted F1-score: 0.9075.

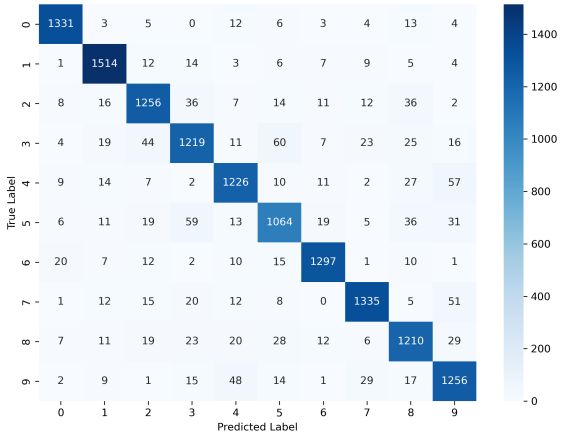


Figure 14 Confusion matrix of the Decision Tree classifier with max_depth=15. Most classes are well-classified with minor misclassifications in classes 3, 5, and 9.

Summary

Increasing the maximum depth of the Decision Tree improves test accuracy up to a depth of 15. Beyond this point, accuracy saturates while training time continues to increase. The final model achieves a high overall accuracy of 90.76%, with strong per-class performance.

5 Comparative Analysis

In this section, we compare the performance of the four implemented algorithms: Decision Tree (DT), K-Nearest Neighbors (KNN), Multiclass Logistic Regression (MLR), and Multi-Layer Perceptron (MLP). The comparison considers overall accuracy, macro-averaged F1-score, weighted F1-score, and computational efficiency.

5.1 Overall Performance Comparison

Table 11 Comparative summary of test accuracy, training time, and F1-scores for all algorithms.

Algorithm Configuration	Acc. (%)	Train Time	Macro F1	Weighted F1
Decision Tree, depth=15	90.76	36.34 s	0.9066	0.9075
KNN (BruteForce w/Norm, K=3, TS=10k)	90.50	0.400 s	0.8801	0.8768
MLR, epochs=200	84.44	17.60 s	0.8419	0.8439
MLP (L=[112,128,64,10], E=1000)	72.02	457.29 s	0.7021	0.7073

5.2 Discussion

Best-performing Algorithm: K-Nearest Neighbors achieves the best balance between accuracy and computational efficiency. With a training size of 10,000 and K=3, it reaches 90.50% test accuracy with a very short training time (0.400 s), making it highly practical for our computational environment.

Trade-offs:

- Decision Tree achieves slightly higher accuracy (90.76%) and F1-scores, but its training time is much longer (36.34 s), which is a disadvantage under limited computational resources.
- MLR and MLP both have significantly lower accuracy and F1-scores. Due to our limited computational resources, it was not feasible to further increase epochs or model size to improve their performance.

Challenging Classes: Across all algorithms, certain classes remain difficult, particularly those with overlapping features. KNN handles these moderately well, while MLR and MLP struggle more, likely due to linear boundaries and insufficient network complexity.

Algorithm Strengths and Weaknesses:

- **KNN:** High accuracy and very fast training for moderate dataset size; simple implementation and robust to parameter changes.
- **Decision Tree:** High accuracy but computationally expensive; interpretable but slow for larger datasets.
- **MLR:** Fast to train but limited accuracy due to linear model assumptions.
- **MLP:** Can model complex patterns, but extremely slow training and underperformed due to limited computational resources.

Conclusion: Considering both accuracy and efficiency, KNN is the most suitable algorithm for our current environment. Decision Tree is accurate but too slow, while MLR and MLP are limited in both precision and training feasibility. Resource constraints prevented further tuning of MLR and MLP to achieve better results.