# ENEE 436 Project 2

Neural Networks, K-Means Clustering, and Decision Trees

Rayan Elias

05/10/2026

---

*Datasets used:* MNIST (60,000 train / 10,000 test) for Problems 2–3 and the UCI Iris dataset (150 samples) for Problem 4. Code is provided as three Python scripts in the `code/` directory; see Section 5.

## Contents

**1 Problem 1 – Neural Network Derivations** **1**

**2 Problem 2 – Single-Layer Sigmoid Network on MNIST** **3**

**3 Problem 3 – K-Means Classification of MNIST** **5**

**4 Problem 4 – Decision Tree on the Iris Dataset** **8**

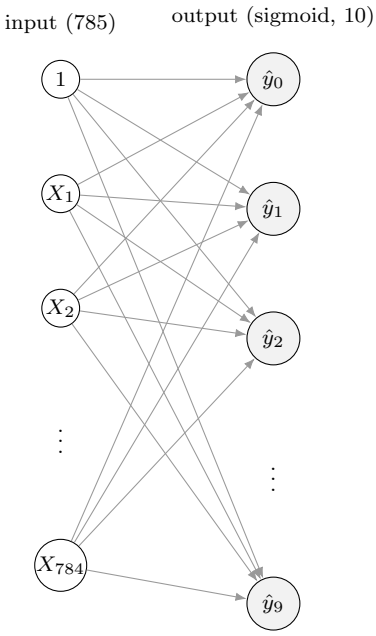
**5 How to run the code** **11**

---

# 1 Problem 1 – Neural Network Derivations

### (a) Network architecture and forward equations

The network has a single linear layer mapping the augmented input $X \in \mathbb{R}^{785}$ (component $X_0 = 1$ is the bias dummy) into 10 sigmoid output units, one per digit class. Let $W \in \mathbb{R}^{10\times 785}$ be the weight matrix, $W_{kj}$ the weight from input unit $j$ into output unit $k$, and let $f(z) = \sigma(z) = \frac{1}{1+e^{-z}}$.




For each output unit $k \in \{0, \ldots, 9\}$,

$$z_k \;=\; \sum_{j=0}^{784} W_{kj}\, X_j \quad = \quad (WX)_k, \qquad \hat{y}_k \;=\; f(z_k) \;=\; \frac{1}{1+e^{-z_k}}.$$

The predicted class is $\hat{c} = \arg\max_k \hat{y}_k$.

## (b) Backpropagation update rule (sigmoid)

Using squared-error loss against a one-hot target $t \in \{0,1\}^{10}$,

$$L \;=\; \tfrac{1}{2} \sum_{k=0}^{9} \big(\hat{y}_k - t_k\big)^2.$$

Because $\hat{y}_k$ depends on $W_{kj}$ only through $z_k$,

$$\frac{\partial L}{\partial W_{kj}} = \underbrace{(\hat{y}_k - t_k)}_{\partial L/\partial \hat{y}_k}\; \underbrace{f'(z_k)}_{\partial \hat{y}_k/\partial z_k}\; \underbrace{X_j}_{\partial z_k/\partial W_{kj}} \quad .$$

Since $f(z) = \sigma(z)$ satisfies $f'(z) = f(z)\big(1 - f(z)\big) = \hat{y}_k(1-\hat{y}_k)$,

$$\boxed{\frac{\partial L}{\partial W_{kj}} = (\hat{y}_k - t_k)\, \hat{y}_k(1-\hat{y}_k)\, X_j} \qquad \Longrightarrow \qquad W_{kj} \;\leftarrow\; W_{kj} \;-\; \eta\, (\hat{y}_k - t_k)\, \hat{y}_k(1-\hat{y}_k)\, X_j.$$

In matrix form, with $\delta = (\hat{y} - t) \odot \hat{y} \odot (1 - \hat{y}) \in \mathbb{R}^{10}$,

$$W \;\leftarrow\; W \;-\; \eta\, \delta\, X^{\top}.$$

### (c) Derivative of $\tanh$ in terms of $f(x)$

For $f(x) = \tanh x = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$, using the quotient rule

$$f'(x) = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} = 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right)^2 = 1 - f(x)^2.$$

So $\boxed{f'(x) = g\big(f(x)\big) = 1 - f(x)^2}$, depending only on $f(x)$.

### (d) Backprop update rule with $\tanh$ activation

The chain rule gives

$$\frac{\partial L}{\partial W_{kj}} = (\hat{y}_k - t_k)\left(1 - \hat{y}_k^2\right) X_j, \qquad \boxed{W_{kj} \leftarrow W_{kj} - \eta\,(\hat{y}_k - t_k)\,(1 - \hat{y}_k^2)\,X_j}.$$

In matrix form, with $\delta = (\hat{y} - t) \odot (1 - \hat{y} \odot \hat{y})$, $W \leftarrow W - \eta\,\delta\,X^\top$.

## 2 Problem 2 – Single-Layer Sigmoid Network on MNIST

### (a) Sigmoid derivative in terms of $f(x)$

$$f(x) = \frac{1}{1 + e^{-x}} \implies f'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \cdot \frac{e^{-x}}{1 + e^{-x}} = f(x)\big(1 - f(x)\big).$$

### (b) Backpropagation update rule

Identical to Problem 1(b). With augmented input $X \in \mathbb{R}^{785}$, $X_0 = 1$, weight matrix $W \in \mathbb{R}^{10\times 785}$, target $t \in \{0,1\}^{10}$, output $\hat{y} = \sigma(WX)$, and loss $L = \frac{1}{2}\|\hat{y} - t\|^2$,

$$W_{kj} \;\leftarrow\; W_{kj} \;-\; \eta\,(\hat{y}_k - t_k)\,\hat{y}_k(1 - \hat{y}_k)\,X_j.$$

### (c) The `vectorize_image` function

The function flattens a $d \times d$ image in row-major order and prepends a 1 to absorb the bias, returning a $(d^2 + 1)$-vector.

```python
def vectorize_image(matrix):
    matrix = np.asarray(matrix)
    flat = matrix.flatten()
    return np.concatenate(([1.0], flat))
```

Listing 1: `vectorize_image` (from `problem2_neural_network.py`)

For computational efficiency the script also provides a vectorized equivalent that builds the entire $N \times 785$ design matrix in one call (this is mathematically identical to applying `vectorize_image` to every image):

```python
def build_design(images_flat):
    N = images_flat.shape[0]
    ones = np.ones((N, 1), dtype=np.float32)
    return np.hstack([ones, images_flat.astype(np.float32) / np.float32(255.0)])
```

Listing 2: Vectorized design-matrix construction

## (d) Implementation of the classifier

Pixels are scaled to $[0, 1]$ and labels one-hot-encoded. Training is per-sample (stochastic) gradient descent following the update rule of part (b). Weights are saved to disk after every epoch so training can be resumed.

```
def sigmoid(z):
    z = np.clip(z, -500.0, 500.0)
    return 1.0 / (1.0 + np.exp(-z))

def train_chunk(W, state, X_train, Y_train, y_train, X_test, y_test, n_epochs):
    rng = np.random.default_rng(SEED + state["epoch"] + 1)
    N = X_train.shape[0]
    target = min(state["epoch"] + n_epochs, state["total_epochs"])
    lr = np.float32(state["lr"])
    one = np.float32(1.0)
    while state["epoch"] < target:
        order = rng.permutation(N)
        for i in order:
            x = X_train[i]
            t = Y_train[i]
            z = W @ x
            y_hat = sigmoid(z)
            delta = (y_hat - t) * y_hat * (one - y_hat)
            W -= lr * np.outer(delta, x)
        state["epoch"] += 1
        ...
```

Listing 3: Forward pass and per-sample SGD update

## (e) Training and evaluation

**Optimizer**
per-sample SGD (one update per example, random epoch shuffle)

**Initialization**
$W \sim \mathcal{N}(0, 0.01^2)$ in `float32`

**Learning rate**
$\eta = 0.1$

**Epochs**
20

**Train / test split**
60,000/10,000 (full MNIST CSV)

**Final train accuracy**
**0.9225**

**Final test accuracy**
**0.9217**

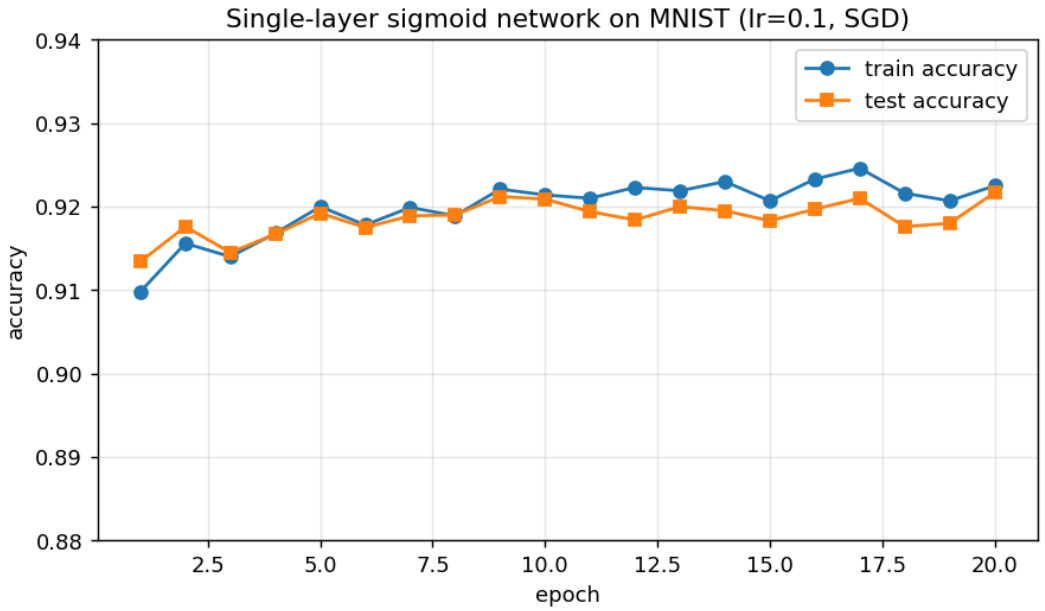



Figure 1: Per-epoch train and test accuracy for the single-layer sigmoid network on MNIST. The model converges within roughly five epochs and then fluctuates around $\sim 92\%$ test accuracy, which is close to the well-known ceiling for a purely linear softmax/sigmoid classifier on raw MNIST pixels.

| epoch | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 17 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|
| train acc. | 0.9098 | 0.9156 | 0.9168 | 0.9178 | 0.9189 | 0.9214 | 0.9223 | 0.9230 | 0.9246 | 0.9225 |
| test acc. | 0.9134 | 0.9176 | 0.9167 | 0.9175 | 0.9190 | 0.9209 | 0.9184 | 0.9195 | 0.9210 | 0.9217 |

Table 1: Selected entries from the training log (`results/problem2_training_log.txt`).

# 3 Problem 3 – K-Means Classification of MNIST

## (a)–(b) Loading and reshaping

The training data is loaded as an $N \times 28 \times 28$ array via Pandas/NumPy and reshaped to $N \times 784$:

```
def load_mnist():
    train = pd.read_csv(os.path.join(DATA_DIR, "mnist_train.csv"), header=None).values
    test = pd.read_csv(os.path.join(DATA_DIR, "mnist_test.csv"), header=None).values
    y_train = train[:, 0].astype(np.int64)
    y_test = test[:, 0].astype(np.int64)
    X_train_3d = train[:, 1:].astype(np.float32).reshape(-1, 28, 28)
    X_test_3d = test[:, 1:].astype(np.float32).reshape(-1, 28, 28)
    return X_train_3d, y_train, X_test_3d, y_test

X_train_flat = X_train_3d.reshape(N, 784)
X_test_flat = X_test_3d.reshape(X_test_3d.shape[0], 784)
```

Listing 4: Loading and flattening MNIST for K-Means

### (c) K-Means with $K = 10$

`MiniBatchKMeans` (a stochastic variant of K-Means with batch size 4096 and `n_init` $= 5$) is fit on the 60,000 $\times$ 784 training array. Cluster labels are then assigned by majority vote of the training labels falling into each cluster.

```python
def assign_cluster_labels(cluster_assignments, true_labels, n_clusters):
    cluster_to_label = np.zeros(n_clusters, dtype=np.int64)
    for c in range(n_clusters):
        mask = (cluster_assignments == c)
        if mask.sum() == 0:
            cluster_to_label[c] = -1
            continue
        counts = np.bincount(true_labels[mask], minlength=10)
        cluster_to_label[c] = int(np.argmax(counts))
    return cluster_to_label

km = MiniBatchKMeans(n_clusters=10, n_init=5, random_state=SEED,
                     batch_size=4096, max_iter=300)
km.fit(X_train_flat)
```

Listing 5: K-Means fit + majority-vote labelling

### (d) Centroid inspection and cluster→label mapping

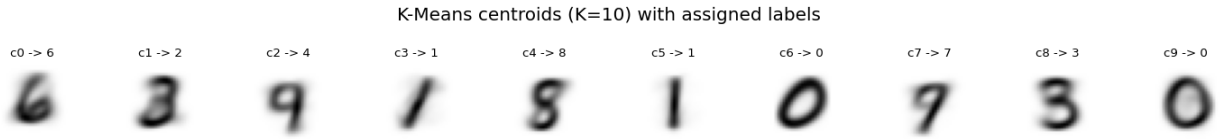



Figure 2: The 10 cluster centroids reshaped back to 28 $\times$ 28 images, together with the majority digit they were assigned. Notice that several digits are missing entirely (no centroid for "5" or "9") while others ("0" and "1") get *two* centroids.

The cluster→label mapping was

$$[\,c_0 \to 6,\ c_1 \to 2,\ c_2 \to 4,\ c_3 \to 1,\ c_4 \to 8,\ c_5 \to 1,\ c_6 \to 0,\ c_7 \to 7,\ c_8 \to 3,\ c_9 \to 0\,].$$

Only **8** of the 10 digit classes ($\{0, 1, 2, 3, 4, 6, 7, 8\}$) are represented; digits 5 and 9 have *no* centroid pointing to them, so they can never be predicted.

### (e) Test accuracy of $K = 10$ classifier

Using nearest-centroid assignment followed by the cluster→label lookup,

$$\text{train accuracy} = 0.5382, \qquad \text{test accuracy} = \mathbf{0.5496}.$$

### (f) Mitigation strategy

The failure mode is that with only $K = 10$ clusters and 10 digit classes, K-Means is forced to commit one centroid per cluster, but it is free (and locally optimal) to place two centroids on visually-distinct sub-modes of the same digit (e.g. two "0" shapes) while collapsing visually-similar

digits together (the "4"/"9"/"7" trio is notorious). Once that happens, majority-vote labelling can never recover the missing classes.

**Proposed mitigation: *over-clustering*.** Run K-Means with a number of clusters substantially larger than the number of classes (e.g. $K = 50$), then assign each cluster the majority label of its members. This decouples the *visual modes* discovered by K-Means from the *semantic classes*. Each class is then allowed to occupy several centroids covering its different writing styles, and the chance that no centroid lands near a particular digit drops to (essentially) zero.

Two further variants we considered: (i) running K-Means many times with different initialisations and picking the run that maximises the number of distinct labels covered; (ii) using K-Means++ initialisation. Both help marginally, but *over-clustering* is the cleanest fix because it directly addresses the structural cause of the failure.

### (g) Improved accuracy with $K = 50$

Re-running with $K = 50$ produces:

$$\text{train accuracy} = 0.7965, \qquad \text{test accuracy} = \mathbf{0.8002},$$

and *all 10 digit classes* are covered by at least one centroid. This is a +25.1 percentage-point improvement over the $K = 10$ baseline.

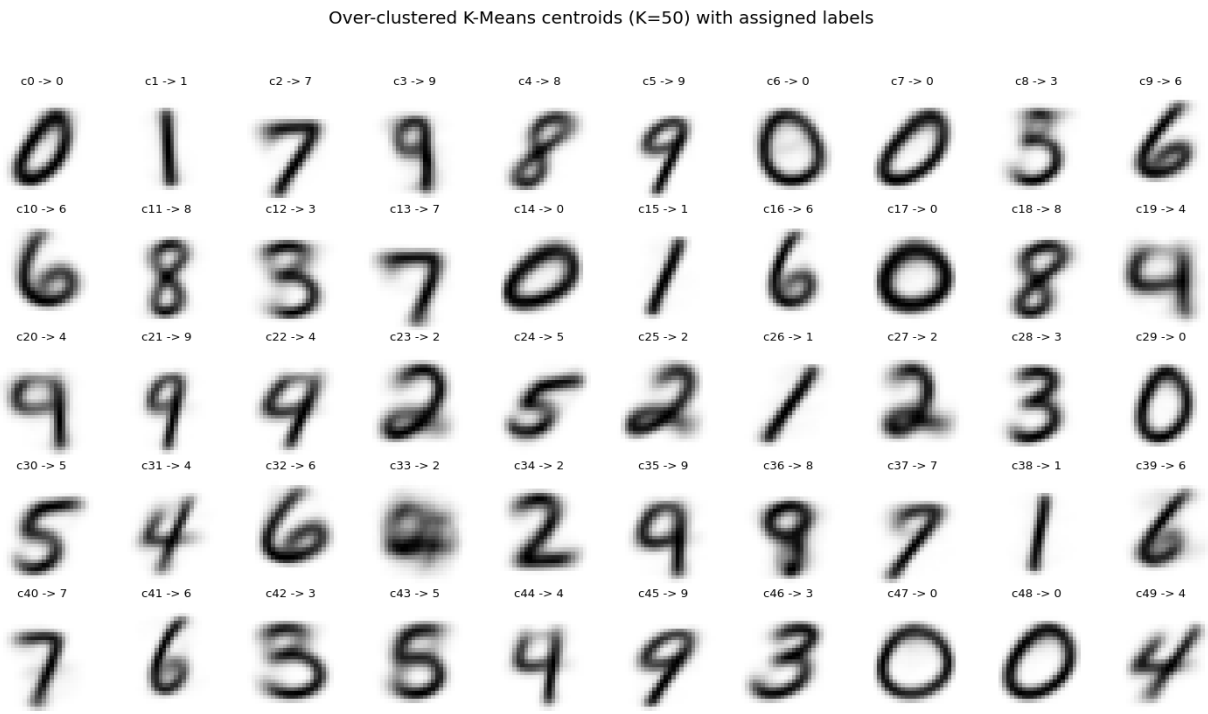



Figure 3: The 50 over-clustered centroids. Multiple writing styles per digit are visible, and every digit class is now represented.

| Method | Train acc. | Test acc. | Digits covered |
|---|---|---|---|
| K-Means $K = 10$ | 0.5382 | 0.5496 | 8 / 10 |
| K-Means $K = 50$ (over-cluster) | 0.7965 | **0.8002** | **10 / 10** |

Table 2: K-Means classifier results on MNIST.

# 4 Problem 4 – Decision Tree on the Iris Dataset

## (a) Loading and fitting

The CSV file `iris.data` is loaded and the four numerical features (`sepal_length`, `sepal_width`, `petal_length`, `petal_width`) are fit against the three-class species label using a scikit-learn `DecisionTreeClassifier`. We use `max_features=2`: at every split the tree picks the best of two randomly-chosen features. This makes the choice of split sensitive to the random seed (and indirectly to data ordering), which is the regime the question asks us to explore.

```
def fit_tree(X, y, seed, max_features=2):
    clf = DecisionTreeClassifier(random_state=seed, max_features=max_features)
    clf.fit(X, y)
    return clf
```

Listing 6: Fitting a decision tree on Iris

## (b) Tree on original ordering

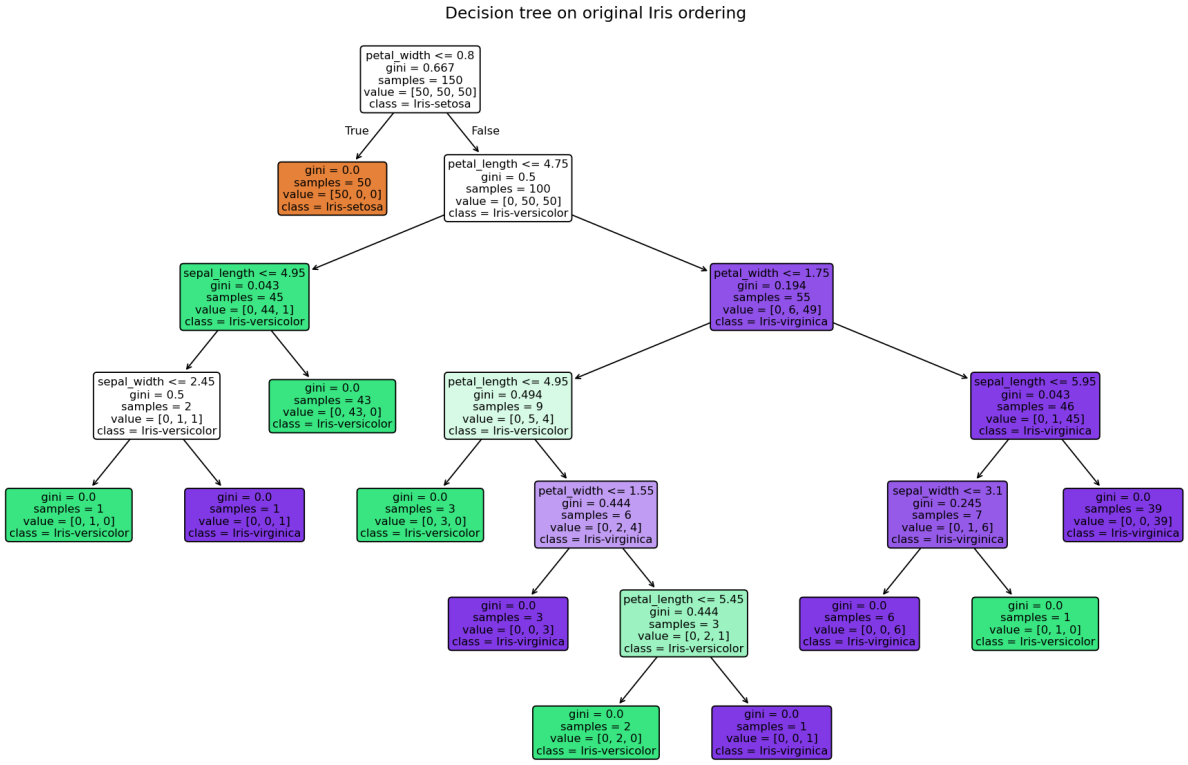



Figure 4: Decision tree fit on the Iris data in its original ordering (`random_state=1`, `max_features=2`). Train accuracy: 1.0000.

## (c) Tree on permuted ordering

```
rng = np.random.default_rng(PERMUTE_SEED)
perm = rng.permutation(X.shape[0])
X_perm, y_perm = X[perm], y[perm]
tree_b = fit_tree(X_perm, y_perm, SEED_B)
```

Listing 7: Permute training rows and refit

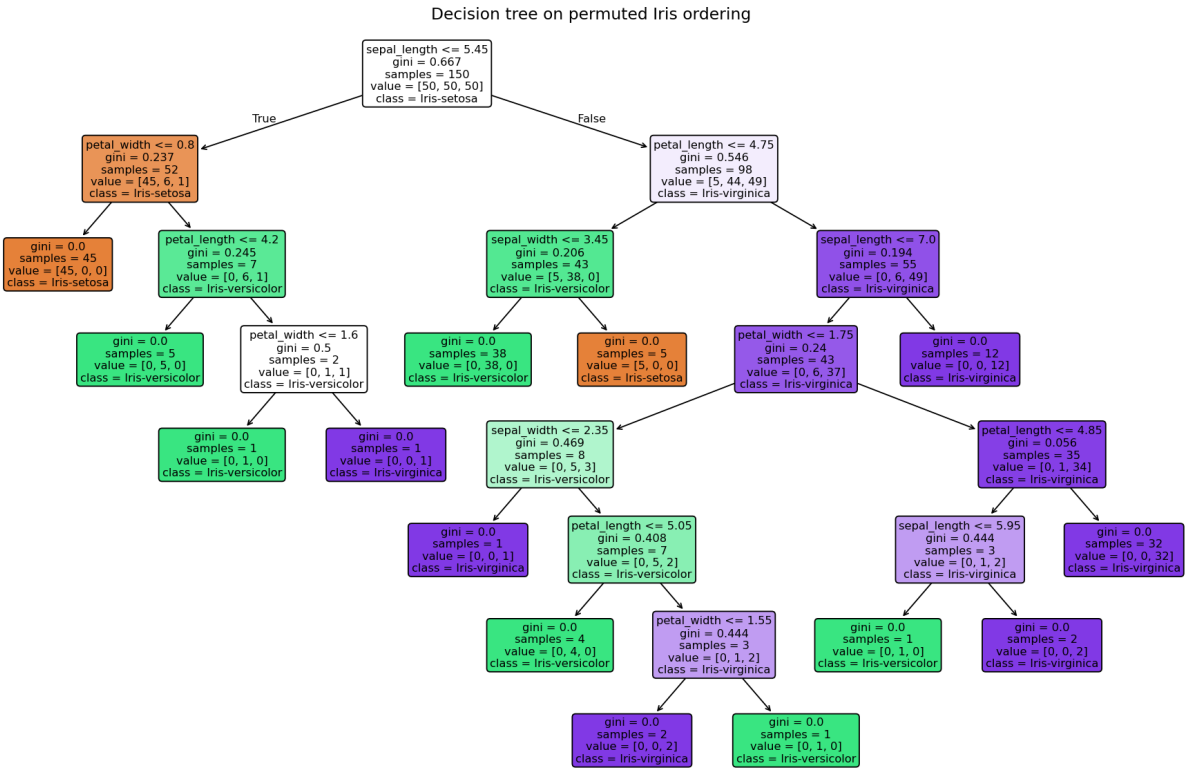



Figure 5: Decision tree fit on the same data after a random permutation (`permute_seed=2024`, `random_state=2024`). Train accuracy: 1.0000. The two trees achieve identical training accuracy but choose different root features and different internal splits; their decision regions therefore disagree on a non-trivial portion of feature space.

### (d) An example classified differently by the two trees

The script enumerates a $40^4 = 2{,}560{,}000$-point grid spanning the per-feature min/max of the dataset and finds all points where the two trees disagree, then picks the one closest to a real training example.

**Divergent example (sepal length, sepal width, petal length, petal width):**

$$x^* = (6.146,\ 2.185,\ 6.749,\ 0.654).$$

- **Tree from part (b)** predicts **Iris-setosa**.
- **Tree from part (c)** predicts **Iris-virginica**.

Across the full grid the two trees disagree on 1,076,158 of 2,560,000 points ($\approx 42\%$), so the example above is one of many such mismatches.

**Why they disagree.** The example combines a *very long* petal length (6.749, characteristic of virginica) with an *unusually narrow* petal width (0.654, characteristic of setosa). The original tree splits first on `petal_width` and routes anything $\leq 0.8$ into the setosa leaf; the permuted tree splits first on `petal_length` and routes anything $> 5.45$ into the virginica subtree. Because the example's two "signature" features point to different classes, and the two trees give priority to different features at the root, the trees commit to different leaves.

**Take-away.** Decision trees with randomised feature subsampling are not unique: changing the data ordering or the random seed can produce structurally different trees that nevertheless fit the training set perfectly. They generalise to different decision regions for unseen examples, which is precisely the variance that ensemble methods (Random Forests, Extra-Trees) exploit.

# 5 How to run the code

**Requirements**

Python 3.9+, `numpy`, `pandas`, `scikit-learn`, `matplotlib`.
`pip install numpy pandas scikit-learn matplotlib`

**Directory layout**

expected by the scripts:

```
436Project2/
  MNIST_CSV/          (mnist_train.csv, mnist_test.csv)
  iris/               (iris.data)
  code/
    problem2_neural_network.py
    problem3_kmeans.py
    problem4_decision_tree.py
    make_learning_curve.py
  results/            (created automatically)
  figures/            (created automatically)
```

**Problem 2**

`python code/problem2_neural_network.py 20`
trains the single-layer sigmoid net for 20 epochs (resumable; pass any integer to train that many additional epochs).

**Problem 3**

`python code/problem3_kmeans.py all`
or run the three checkpointed steps individually: `baseline`, `overcluster`, `finalize`.

**Problem 4**

`python code/problem4_decision_tree.py`
fits both decision trees, saves them as PNG, and writes the divergent example to `results/problem4_summary.txt`.

**Plots**

`python code/make_learning_curve.py` regenerates the NN learning curve.

All numerical results in this report are reproducible with the seeds embedded in the scripts (`SEED=42` for Problems 2 and 3, `SEED_A=1` / `SEED_B=2024` / `PERMUTE_SEED=2024` for Problem 4).