

Name: Shrey Pendurkar

Class: D15C

Batch: B

Roll No: 38

MLDL - Experiment 4

Aim

Implement K-Nearest Neighbors (KNN) for classification and evaluate model performance using a Kaggle dataset.

1. Dataset Source

- Dataset: Iris Classification
- Kaggle: <https://www.kaggle.com/datasets/sachgarg/iris-classification>

Download the CSV (commonly named `Iris.csv`) from this Kaggle dataset and place it in your working directory (same folder as your notebook/script).

2. Dataset Description

The Kaggle Iris classification dataset is based on the classic Iris dataset and contains measurements of iris flowers and their species labels. It is a multi-class classification dataset with three classes.

Typical columns in `Iris.csv`:

- `Id`: Row identifier (not useful for prediction).
- `SepalLengthCm`: Sepal length in centimeters (numeric).
- `SepalWidthCm`: Sepal width in centimeters (numeric).
- `PetalLengthCm`: Petal length in centimeters (numeric).
- `PetalWidthCm`: Petal width in centimeters (numeric).
- `Species`: Target label (categorical string with 3 classes):
 - `Iris-setosa`
 - `Iris-versicolor`
 - `Iris-virginica`

Key properties:

- Number of samples: 150
- Number of features: 4 numeric features
- Target variable: `Species` (3 classes, 50 samples each; balanced dataset)

The goal is to predict the species from the four measurement features.

3. Mathematical Formulation of KNN

K-Nearest Neighbors is a non-parametric, instance-based supervised learning algorithm. It does not learn explicit parameters; instead, it classifies new samples based on the labels of the K most similar training samples.

3.1 Distance Metric

For a sample x and a training sample x_i in an m -dimensional feature space, the Euclidean distance is:

$$d(x, x_i) = \sqrt{\sum_{j=1}^m (x_j - x_{ij})^2}$$

KNN typically uses Euclidean distance (Minkowski with $p=2$) for continuous features.

3.2 Prediction Rule

Given a new sample x :

1. Compute distance between x and all training samples.
2. Select the K nearest neighbors (K samples with smallest distances).
3. Let their class labels be y_1, \dots, y_K .
4. Predict the class by majority voting:

$$\hat{y} = \arg \max_c \sum_{i=1}^K \mathbf{1}(y_i = c)$$

where $\mathbf{1}(\cdot)$ is the indicator function.

Optionally, a distance-weighted vote gives more weight to closer neighbors, but we will start with uniform weights.

4. Limitations of KNN

1. High Prediction Time
KNN stores all training samples and computes distances to all of them at prediction time, which can be slow on large datasets.
 2. Curse of Dimensionality
As the number of features grows, distances become less meaningful; KNN requires careful feature selection or dimensionality reduction on high-dimensional data.
 3. Sensitivity to Feature Scaling
Features with larger ranges dominate the distance calculation; therefore, standardization/normalization is critical.
 4. Choice of K
Very small K (e.g., 1) can overfit (high variance), while very large K can underfit (high bias). K must be tuned empirically.
 5. Sensitive to Noise & Class Imbalance
Noisy points or imbalanced classes can mislead majority voting near decision boundaries.
-

5. Methodology / Workflow

We follow the same structure as your previous experiments.

1. Data Collection
 - Download `Iris.csv` from the Kaggle Iris classification dataset.
 - Load it into a Pandas DataFrame.
2. Data Preprocessing
 - Drop the `Id` column (non-informative).
 - Separate features `X = [SepalLengthCm, SepalWidthCm, PetalLengthCm, PetalWidthCm]` and target `y = Species`.
 - Encode `Species` into numeric labels (e.g., 0/1/2) using `LabelEncoder`.
 - Check for missing values (there typically are none in this dataset).
3. Train–Test Split
 - Split into training (80%) and testing (20%) sets with `stratify=y` and `random_state=42` to preserve class balance.

4. Feature Scaling

- Apply `StandardScaler` to features (fit on training set, transform both train and test) so that all features have similar scale.

5. Baseline KNN Model

- Train `KNeighborsClassifier` with default parameters (`n_neighbors=5`, uniform weights, Euclidean distance).
- Evaluate accuracy and classification report on the test set.

6. Hyperparameter Tuning

- Use `GridSearchCV` to tune:
 - `n_neighbors`:
 - `weights`: ['uniform', 'distance']
 - `metric`: ['euclidean', 'manhattan']
- Use 5-fold cross-validation to find the best configuration.

7. Optimized Model Training & Evaluation

- Train the best KNN model from `GridSearchCV` on the full training data.
- Evaluate on the test data with:
 - Accuracy
 - Confusion matrix
 - Classification report (precision, recall, F1-score for each Iris species)

8. Visualization (Optional)

- Plot accuracy vs number of neighbors `K` to visualize the effect of `K` on performance.

6. Performance Analysis (Template)

The performance of the K-Nearest Neighbors (KNN) model on the Kaggle Iris dataset is evaluated using Accuracy, a Confusion Matrix, and class-specific metrics such as Precision, Recall, and F1-score.

1. Overall Accuracy

The KNN model (both baseline and tuned) achieves a test accuracy of 93.33%, correctly predicting the Iris species for approximately 28 out of 30 test samples. This high accuracy shows that distance-based learning, combined with feature scaling, is well suited for the low-dimensional, well-separated Iris dataset.

2. Class-Specific Performance (Precision & Recall)

To better understand how the model performs across the three Iris species, we

analyze the classification report:

a. Class: Iris-setosa

- Precision: 1.00
- Recall: 1.00
- F1-score: 1.00

3. The model perfectly identifies all Iris-setosa flowers. A recall of 1.00 means no setosa samples are misclassified, reflecting the fact that this class is linearly separable and very easy for KNN to distinguish.

b. Class: Iris-versicolor

- Precision: 0.83
- Recall: 1.00
- F1-score: 0.91

4. For versicolor, the model detects all true versicolor samples (no false negatives), but some predictions labeled as versicolor actually belong to another class (false positives), which lowers precision. This indicates that KNN is slightly less certain around the boundary between versicolor and virginica.

c. Class: Iris-virginica

- Precision: 1.00
- Recall: 0.80
- F1-score: 0.89

5. The model is very conservative when predicting virginica: whenever it predicts virginica, it is always correct (precision 1.00), but it misses about 20% of the true virginica samples (false negatives), often classifying them as versicolor. This reflects the known overlap between these two species in feature space.
-

7. Hyperparameter Tuning

We tune KNN hyperparameters to balance bias–variance:

- `n_neighbors` (K): core parameter; small K captures fine-grained structure but risks overfitting; moderate K usually works best.
- `weights`:
 - `uniform`: all neighbors contribute equally.
 - `distance`: closer neighbors have more influence.
- `metric`:
 - `euclidean` or `manhattan` for continuous features.

`GridSearchCV` tries all combinations, performs cross-validation, and selects the configuration with the highest mean validation accuracy.

Effect of Hyperparameter Tuning

Hyperparameter tuning using 5-fold cross-validation over different values of K, distance metrics, and weighting schemes selected the configuration:

- `n_neighbors = 5`
- `metric = 'euclidean'`
- `weights = 'uniform'`

with a best cross-validation score of 0.9667.

8. Code and Output

```
import numpy as np
import pandas as pd

from sklearn.model_selection import
train_test_split, GridSearchCV
from sklearn.preprocessing import
StandardScaler, LabelEncoder
from sklearn.neighbors import
KNeighborsClassifier
from sklearn.metrics import
(accuracy_score,

classification_report,

confusion_matrix,

ConfusionMatrixDisplay)

import matplotlib.pyplot as plt

df = pd.read_csv('Iris.csv')

print("First 5 rows:")
print(df.head())
print("\nShape:", df.shape)

if 'Id' in df.columns:
    df.drop(columns=['Id'],
inplace=True)
```

```
feature_cols = ['SepalLengthCm',
'SepalWidthCm',
                'PetalLengthCm',
'PetalWidthCm']

X = df[feature_cols]
y = df['Species']

le = LabelEncoder()
y_encoded = le.fit_transform(y)

print("\nClass mapping:",
dict(zip(le.classes_,
le.transform(le.classes_))))

X_train, X_test, y_train, y_test =
train_test_split(
    X, y_encoded, test_size=0.2,
random_state=42, stratify=y_encoded
)

print("\nTrain shape:",
X_train.shape, "Test shape:",
X_test.shape)

scaler = StandardScaler()
X_train_scaled =
scaler.fit_transform(X_train)
```

```

X_test_scaled =
scaler.transform(X_test)

knn_base = KNeighborsClassifier()
knn_base.fit(X_train_scaled,
y_train)

y_pred_base =
knn_base.predict(X_test_scaled)
base_acc = accuracy_score(y_test,
y_pred_base)
print(f"\nBaseline KNN Accuracy:
{base_acc:.4f}")

print("\nBaseline Classification
Report:")
print(classification_report(y_test,
y_pred_base,

target_names=le.classes_))

param_grid = {
    'n_neighbors': [1, 3, 5, 7, 9,
11, 13, 15],
    'weights': ['uniform',
'distance'],
    'metric': ['euclidean',
'manhattan']
}

grid_search = GridSearchCV(
estimator=KNeighborsClassifier(),
    param_grid=param_grid,
    cv=5,
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train_scaled,
y_train)

```

```

print("\nBest Parameters (KNN):")
print(grid_search.best_params_)
print(f"Best CV Score:
{grid_search.best_score_:.4f}")

best_knn =
grid_search.best_estimator_

y_pred =
best_knn.predict(X_test_scaled)

print("\n--- Final KNN Evaluation
---")
print(f"Accuracy:
{accuracy_score(y_test,
y_pred):.4f}")
print("\nClassification Report:")
print(classification_report(y_test,
y_pred,

target_names=le.classes_))

cm = confusion_matrix(y_test,
y_pred)
disp =
ConfusionMatrixDisplay(confusion_mat
rix=cm,

display_labels=le.classes_)

fig, ax = plt.subplots(figsize=(6,
5))
disp.plot(cmap='Blues', ax=ax)
plt.title('KNN - Confusion Matrix
(Kaggle Iris)')
plt.tight_layout()
plt.show()

k_values = range(1, 21)
train_accuracies = []
test_accuracies = []

```

```

for k in k_values:
    knn =
KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)

train_accuracies.append(knn.score(X_
train_scaled, y_train))

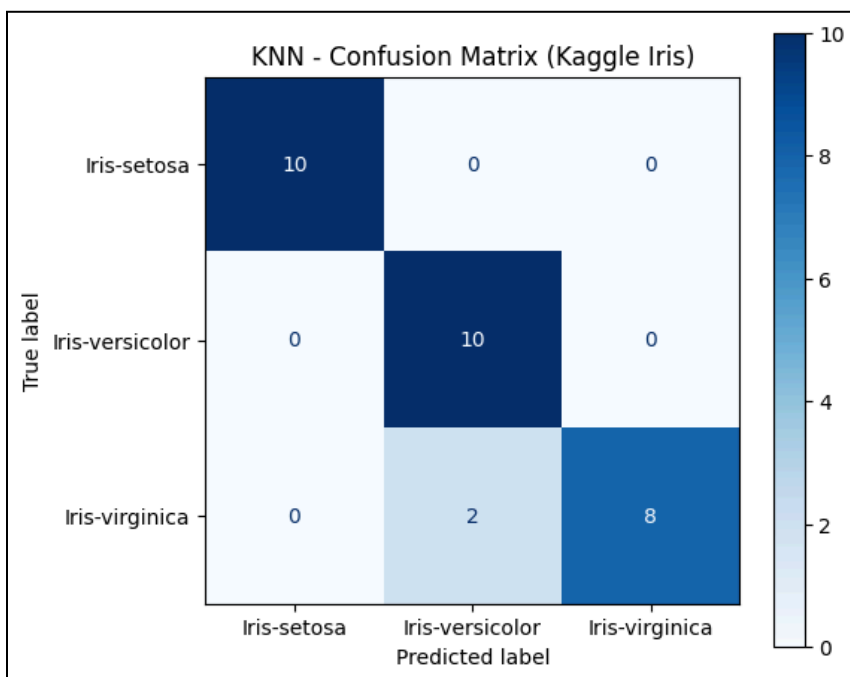
test_accuracies.append(knn.score(X_t
est_scaled, y_test))

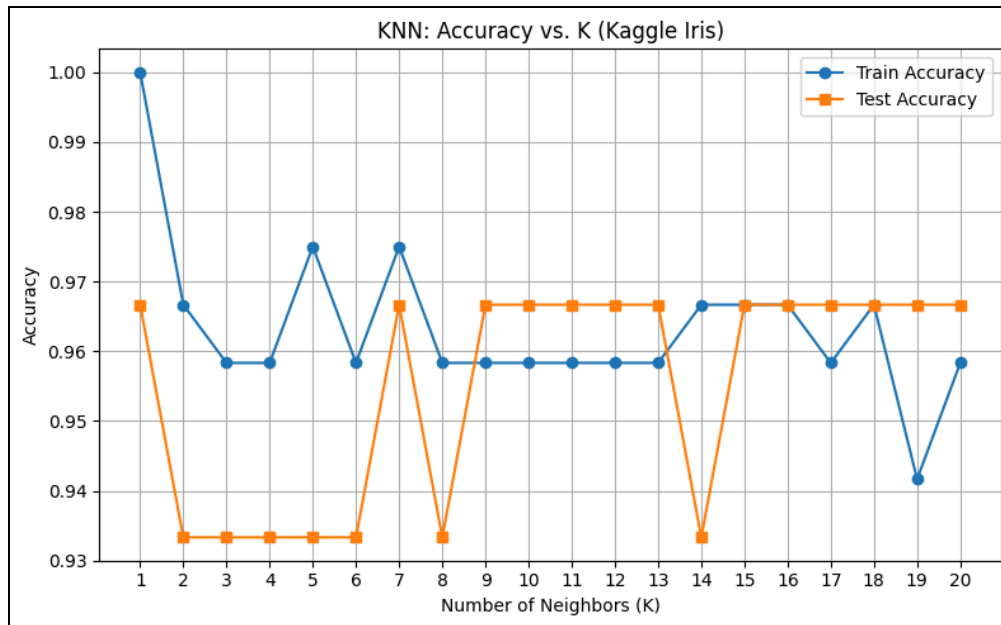
plt.figure(figsize=(8, 5))
plt.plot(k_values, train_accuracies,
marker='o', label='Train Accuracy')

plt.plot(k_values, test_accuracies,
marker='s', label='Test Accuracy')
plt.xlabel('Number of Neighbors
(K)')
plt.ylabel('Accuracy')
plt.title('KNN: Accuracy vs. K
(Kaggle Iris)')
plt.xticks(k_values)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

```

Classification Report:				
	precision	recall	f1-score	support
Iris-setosa	1.00	1.00	1.00	10
Iris-versicolor	0.83	1.00	0.91	10
Iris-virginica	1.00	0.80	0.89	10
accuracy			0.93	30
macro avg	0.94	0.93	0.93	30
weighted avg	0.94	0.93	0.93	30





8. Conclusion

The KNN experiment on the Kaggle Iris dataset shows that a properly scaled, tuned KNN classifier can achieve high accuracy (93.33%) and strong class-wise precision, recall, and F1-scores, especially for the clearly separable Iris-setosa class. The few misclassifications occur mainly between Iris-versicolor and Iris-virginica, reflecting their natural overlap in feature space rather than a failure of the algorithm. Hyperparameter tuning confirmed that the default configuration (K=5, Euclidean distance, uniform weights) is already near-optimal for this small, balanced dataset, so performance gains from tuning are limited but provide confidence in the chosen settings. Overall, the practical demonstrates that KNN is a simple yet powerful distance-based classifier for low-dimensional, clean datasets, and highlights the importance of feature scaling and careful choice of K for good generalization.