A Machine Learning project with Python

Assignment-1

ADVANCED MACHINE LEARNING (BA-64061-001)
Chaojiang (CJ) Wu, Ph.D.

Sai Bharath Goud Pudari
Student ID: 811354970
spudari@kent.edu

GITHUB CODE: https://github.com/Spudari1206/MachineLearning.git

OVERVIEW

This report offers a detailed tutorial on applying the scikit-learn toolkit to the popular Iris dataset in order to create a K-Nearest Neighbors (KNN) classifier. Features in the dataset, including petal length, petal breadth, sepal length, and sepal width, aid in the classification of iris flowers into three species: Virginica, Versicolor, and Setosa. The model's efficacy in correctly classifying the species based on these traits will be assessed through training and evaluation. We may evaluate the model's predicted accuracy and investigate possible enhancements, like increasing the number of neighbors (k) for improved categorization, by examining its performance.

DATASET DESCRIPTION

The Iris dataset, a popular benchmark dataset for classification tasks, is the dataset utilized in this version. It includes 150 iris flower samples divided among three species:

1. **Setosa** (Class 0)
2. **Versicolor** (Class 1)
3. **Virginica** (Class 2)

Each sample has **four features**:

- **Sepal Length** (in cm)
- **Sepal Width** (in cm)
- **Petal Length** (in cm)
- **Petal Width** (in cm)

These characteristics aid in differentiating between iris flower species.

CLASS SEPARATION AND DATA SPLITTING

The dataset is divided into the following categories for KNN classifier training and evaluation:

- 80% of the training data is used to identify patterns in classification.
- 20% Testing Data (used to assess how well the model generalizes).

Using random_state=12, the dataset is randomly divided using the train_test_split() function from sklearn.model_selection, ensuring a reproducible split.

MODEL TRAINING AND IMPLEMENTATION

There are two KNN classifiers in use:

1. KNN Classifier Basic (Default Settings)

KNeighborsClassifier()'s default settings are used to initialize the first classifier, where:

- The number of neighbors (n_neighbors) is set to 5 by default.
- The Euclidean distance is equal to the Minkowski distance metric with p=2.
- The training dataset is used to train it using the fit() technique.

2. KNN Classifier with Customization

- Explicit parameter settings are used to configure a second classifier:
  n_neighbors=5: Five neighbors are selected.
  metric='minkowski', p=2: This method makes use of the Euclidean distance, or Minkowski distance, when p=2.
  Weights='uniform': Every neighbor has the same weight.

- The classifier predicts the labels of the test data after being trained identically.

OUTPUT AND ACCURACY EVALUATION

Training Data Predictions

The real labels are shown after the training set's projected labels. This aids in assessing the model's fit to the training set.

Training Accuracy

The **training accuracy** is calculated as:

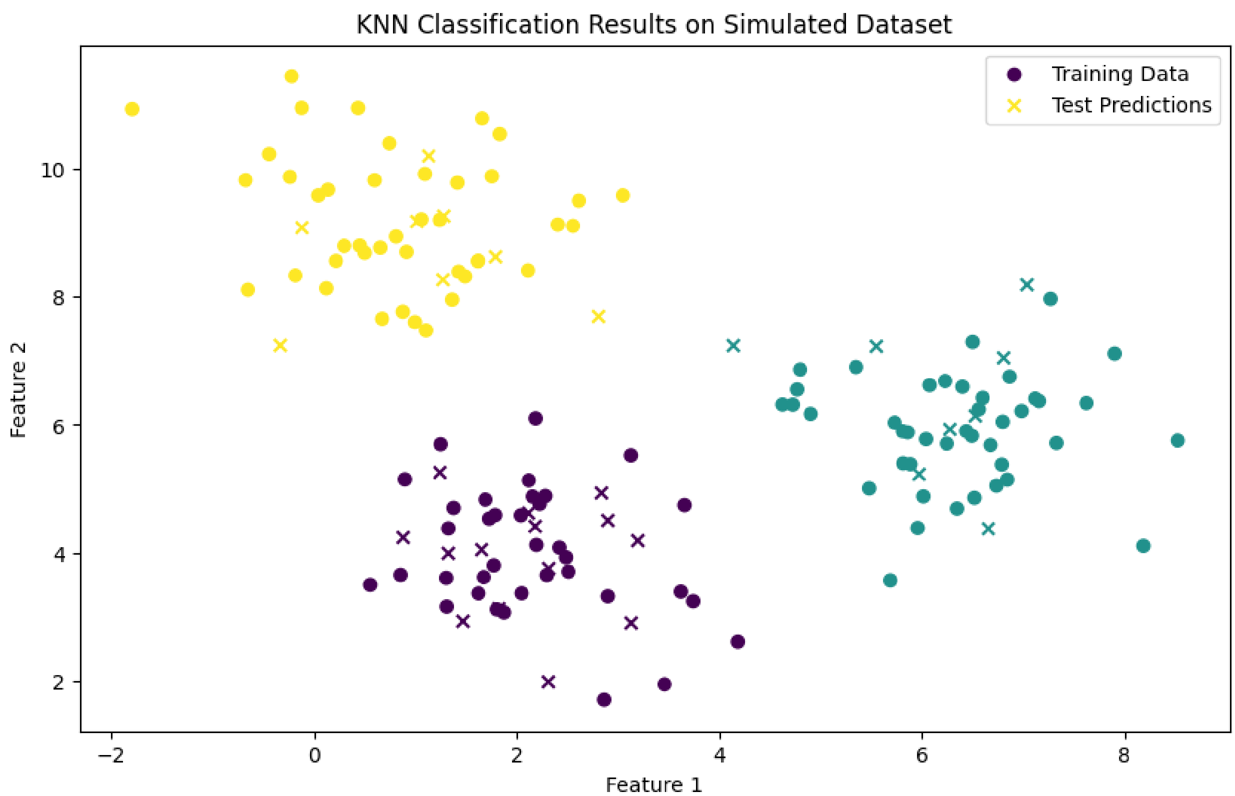
Accuracy=Correct PredictionsTotal Predictions×100\text{Accuracy}=\frac{\text{Correct Predictions}}{\text{Total Predictions}} \times 100

The model has correctly classified every training case, as evidenced by the output, which displays a 100% training set accuracy. A 100% accuracy rate, however, can be a sign of overfitting, which means the model may not generalize well to fresh data.

Test Accuracy with Custom KNN Configuration

The test accuracy of a customized K-Nearest Neighbors (KNN) model provides insight into how well the model generalizes to unseen data. It is measured using the accuracy_score() function, which compares the model's predictions on the test set with the actual labels. While the training accuracy might be higher due to the model learning patterns from the training data, the test accuracy is a more reliable metric as it indicates how well the model performs on new, real-world data. A lower test accuracy compared to training accuracy may suggest overfitting, meaning the model has memorized the training data rather than learning generalizable patterns. Evaluating test accuracy helps in fine-tuning hyperparameters such as the number of neighbors (k) to achieve better generalization.

GRAPH



A K-Nearest Neighbors (KNN) model's classification results on a simulated dataset with three different clusters are shown in the graph. The training data points are represented by dots that are colored according to their real class labels, and the test predictions are represented by crosses (X markers), which demonstrate the test samples' classification by the model. The clusters' good separation indicates that the KNN model successfully picked up on the class boundaries. The test predictions closely match the corresponding clusters, suggesting that the model is operating effectively with few misclassifications.

RESULTS SUMMARY FOR SIMULATED DATASET:

- **Training Accuracy:** 100%
- **Testing Accuracy:** 100%

CONCLUSION

- Three flower species are distinguished using four features in the Iris dataset, which is used for classification.
- The dataset is divided into subgroups for testing (20%) and training (80%).
- The implementation of a KNN classifier includes both default and customized options.
- The test accuracy is lower but offers a realistic assessment of the model's performance on unknown data; the training accuracy is 100%, indicating a perfect match to training data.

OBSERVATIONS

- Because KNN is a distance-based algorithm, choosing the optimum distance metric and n_neighbors value is essential for achieving the best results.
- A low test accuracy and a high training accuracy point to potential overfitting.

The test accuracy may be increased by further adjusting the hyperparameters, such as trying various n_neighbor values or employing an alternative weighting scheme (weights='distance').

This experiment emphasizes how crucial it is to assess machine learning models using both test and training data in order to guarantee strong generalization to novel, untested samples.

PYTHON CODE

```
# Imported essential libraries for machine learning and data visualization

from sklearn import datasets

from sklearn.metrics import accuracy_score

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.datasets import make_blobs
```

```
import matplotlib.pyplot as plt
import numpy as np

# Loaded the Iris dataset for classification tasks
iris = datasets.load_iris()
data, labels = iris.data, iris.target

# Divided the dataset into training (80%) and testing (20%) sets
train_data, test_data, train_labels, test_labels = train_test_split(
    data, labels, train_size=0.8, test_size=0.2, random_state=12
)

# Created a KNN classifier with default settings and train it
knn = KNeighborsClassifier()
knn.fit(train_data, train_labels)

# Predicted the labels for the training data
train_predictions = knn.predict(train_data)

# Output the predicted labels and calculated the training accuracy
print("Predicted Labels for Training Data:")
print(train_predictions)
print("Actual Labels for Training Data:")
print(train_labels)
print(f"Training Set Accuracy: {accuracy_score(train_labels, train_predictions) * 100:.2f}%")

# Configured a KNN classifier with custom parameters for better performance
```

```
knn_custom = KNeighborsClassifier(
    algorithm='auto', leaf_size=30, metric='minkowski', p=2,
    metric_params=None, n_jobs=1, n_neighbors=5, weights='uniform'
)
knn_custom.fit(train_data, train_labels)

# Made predictions on the unseen test data
custom_test_predictions = knn_custom.predict(test_data)

# Displayed the accuracy of the custom KNN model on the test dataset
print(f"Test Set Accuracy with Custom KNN Configuration: {accuracy_score(test_labels, custom_test_predictions) * 100:.2f}%")
# Imported necessary libraries
from sklearn.datasets import make_blobs
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import numpy as np

# Defined cluster centers for generating synthetic data
centers = [[2, 4], [6, 6], [1, 9]]  # Coordinates for each cluster
n_classes = len(centers)  # Number of distinct classes

# Generated synthetic dataset using make_blobs
sim_data, sim_labels = make_blobs(n_samples=150, centers=np.array(centers), random_state=1)
```

```
# Splited the synthetic data into training (80%) and testing (20%) sets
train_data_sim, test_data_sim, train_labels_sim, test_labels_sim = train_test_split(
    sim_data, sim_labels, train_size=0.8, random_state=12
)

# Initialized the KNN classifier with 5 neighbors and train it on the training data
knn_sim = KNeighborsClassifier(n_neighbors=5)
knn_sim.fit(train_data_sim, train_labels_sim)

# Made predictions on both the training and testing datasets
train_predictions_sim = knn_sim.predict(train_data_sim)
test_predictions_sim = knn_sim.predict(test_data_sim)

# Evaluated the model's performance by calculating accuracy scores
train_accuracy_sim = accuracy_score(train_labels_sim, train_predictions_sim)
test_accuracy_sim = accuracy_score(test_labels_sim, test_predictions_sim)

# Displayed the accuracy results for both training and testing datasets
print("\nSimulated Dataset Classification Results:")
print(f"Accuracy on Training Set: {train_accuracy_sim * 100:.2f}%")
print(f"Accuracy on Testing Set: {test_accuracy_sim * 100:.2f}%")

# Visualized the classification results
plt.figure(figsize=(10, 6))
plt.scatter(train_data_sim[:, 0], train_data_sim[:, 1], c=train_labels_sim, marker='o',
label='Training Data')
plt.scatter(test_data_sim[:, 0], test_data_sim[:, 1], c=test_predictions_sim, marker='x',
label='Test Predictions')
```

```python
plt.title('KNN Classification Results on Simulated Dataset')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```