Homework 2

In this homework, I applied a memory-based learning approach to classify data using the K-Nearest Neighbors (KNN) algorithm. The task involved classifying data from a dataset that includes measurements of various flower characteristics such as sepal length, sepal width, petal length, and petal width. The dataset contains 150 data points, with 100 used for training and 50 for testing. The goal was to predict the class of flowers based on these features.

Data Preprocessing

The dataset was loaded and prepared for model training. The features were selected as the first four columns, while the target variable (class) was located in the last column. The dataset was split into training and testing sets using a 2:1 ratio. The following preprocessing steps were applied:

Standard Scaling: The features were standardized using StandardScaler to ensure that each feature contributed equally to the model by transforming them to a common scale.

Handling Class Imbalance: SMOTE (Synthetic Minority Over-sampling Technique) was applied to the training set to address potential class imbalance, ensuring a more balanced distribution of classes during model training.

Dimensionality Reduction: Principal Component Analysis (PCA) was used to reduce the feature space to 95% variance retention, simplifying the model without significant loss of information.]

Model Selection and Tuning

The K-Nearest Neighbors (KNN) classifier was chosen for this task due to its simplicity and effectiveness in classification problems. To optimize the KNN model, a grid search with cross-validation (CV) was conducted over several hyperparameters, including the number of neighbors (n_neighbors), the weight function (weights), and the distance metric (metric). The hyperparameter grid was as follows:

n_neighbors: [3, 5, 7, 9, 11]

weights: ['uniform', 'distance']

metric: ['euclidean', 'manhattan', 'minkowski']

1. n_neighbors: Number of Neighbors to Consider

This parameter defines how many nearest neighbors the KNN algorithm will use to make a prediction for a new data point.

Smaller values (e.g., 3): The model focuses more on local data, making predictions based on just a few neighbors. This can work well when the data is well-separated, but it can be sensitive to noise.

Larger values (e.g., 11): The model looks at a broader range of neighbors, which helps smooth out predictions and reduce the influence of noise, but it might miss finer details in the data.

2. weights: How to Weight the Influence of Neighbors

This parameter decides whether all neighbors are treated equally or if closer neighbors have more influence.

uniform: All neighbors have equal influence on the prediction, which works well when data is well-balanced.

distance: Neighbors closer to the point being predicted have more influence, making the model more sensitive to local patterns in the data.

3. metric: How to Measure Distance Between Points

This parameter defines how the distance between data points is measured. Different metrics can work better for different types of data.

euclidean: The most common distance metric, suitable for data where features are continuous and have similar scales.

manhattan: Measures distance in a grid-like fashion, useful for data where features are more discrete or categorical.

minkowski: A generalization of both Euclidean and Manhattan, allowing you to experiment with different types of distance metrics.

After fitting the grid search, the best parameters were identified, and the optimal KNN model was selected.

Code Explanation

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

The code imports necessary libraries (pandas, matplotlib, numpy).

```
from google.colab import drive
drive.mount('/content/drive')
```

Allows you to access files stored in Google Drive while working in Google Colab. Connects your Google Drive to Colab so you can access your files.

```
data = pd.read_csv('/content/drive/MyDrive/AI/HW-
classification.csv')
names = ['index','sepal leghth(cm)','sepal width(cm)','petal
length(cm)','petal width(cm)','class']
data.head()
```

Reads a CSV file (a data file) from your Google Drive and loads it into a table.

Assigns names to the columns of the data.

Shows the first 5 rows of the data to give you a quick look at it.

```
X = data.iloc[:, :-1].values
y = data.iloc[:, 4].values
```

X = data.iloc[:, :-1].values This picks all the data (except the last column) and stores it in X. This is your input data (features). y = data.iloc[:, 4].values: This picks only the 5th column (the class label) and stores it in y. This is your output data (what you're trying to predict).

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.metrics import accuracy_score
from imblearn.over_sampling import SMOTE
from sklearn.decomposition import PCA
```

KNeighborsClassifier: The K-Nearest Neighbors (KNN) algorithm, used for classification.

StandardScaler: A tool for scaling features so they have zero mean and unit variance (important for KNN).

train_test_split: Splits the dataset into training and test sets.

GridSearchCV: Searches for the best hyperparameters for a given model using cross-validation.

accuracy_score: Measures how well the model performs by comparing predicted values to actual values.

SMOTE: A technique to handle class imbalance by generating synthetic samples.

PCA: Principal Component Analysis (PCA), used for dimensionality reduction.

```

y = data['class'].values
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=1/3, random_state=42)
```

y: The target variable (class labels).

X_train, X_test, y_train, y_test: Splits the data into training and testing sets. 1/3 of the data will be used for testing, and the rest for training.So Test 100 data and Train 50 data

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

StandardScaler: Standardizes the features (scales them to have a mean of 0 and a standard deviation of 1).

X_train_scaled and X_test_scaled: Apply the scaling to the training and test data.

```
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled =
smote.fit_resample(X_train_scaled, y_train)
```

SMOTE: Generates synthetic data points to balance the class distribution in the training set, especially if one class is underrepresented.

```
pca = PCA(n_components=0.95)
X_train_pca = pca.fit_transform(X_train_resampled)
X_test_pca = pca.transform(X_test_scaled)
```

PCA: Reduces the number of features while retaining 95% of the data's variance, making it easier for the model to work with.

X_train_pca and X_test_pca: Apply PCA to the resampled training data and the scaled test data.

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

param_grid: A grid of possible values for the KNN model's hyperparameters. This includes:

```
grid_search = GridSearchCV(KNeighborsClassifier(), param_grid,
cv=5, scoring='accuracy')
grid_search.fit(X_train_pca, y_train_resampled)
```

GridSearchCV: Searches for the best combination of hyperparameters by evaluating the model using 5-fold cross-validation (i.e., splitting the training data into 5 parts and testing on each).

It fits the KNN model with the best hyperparameters on the resampled training data.

```
best_knn = grid_search.best_estimator_
print("Best Parameters:", best_params)
```

best_knn: After the grid search, this stores the best KNN model found based on the hyperparameters.

best_params: This would print the best set of hyperparameters (but there's a small issue—best_params should be grid_search.best_params_).

```
y_pred = best_knn.predict(X_test_pca)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

y_pred: This is the predicted class for each test sample using the best KNN model.

accuracy_score: Compares the predicted values (y_pred) with the true values (y_test) and calculates the accuracy.

```
print: Displays the accuracy of the model as a percentage.
y_pred = knn.predict(X_test_pca)
df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
df
```

y_pred = knn.predict(X_test_pca): Predicts the class for each test sample.

df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred}): Creates a table to compare the actual class (y_test) and the predicted class (y_pred) side by side.

Results and Analysis

Best Parameters: {'metric': 'euclidean', 'n_neighbors': 3, 'weights': 'uniform'}

metric: 'euclidean': The best distance metric for this dataset is Euclidean distance, which calculates the straight-line distance between points.

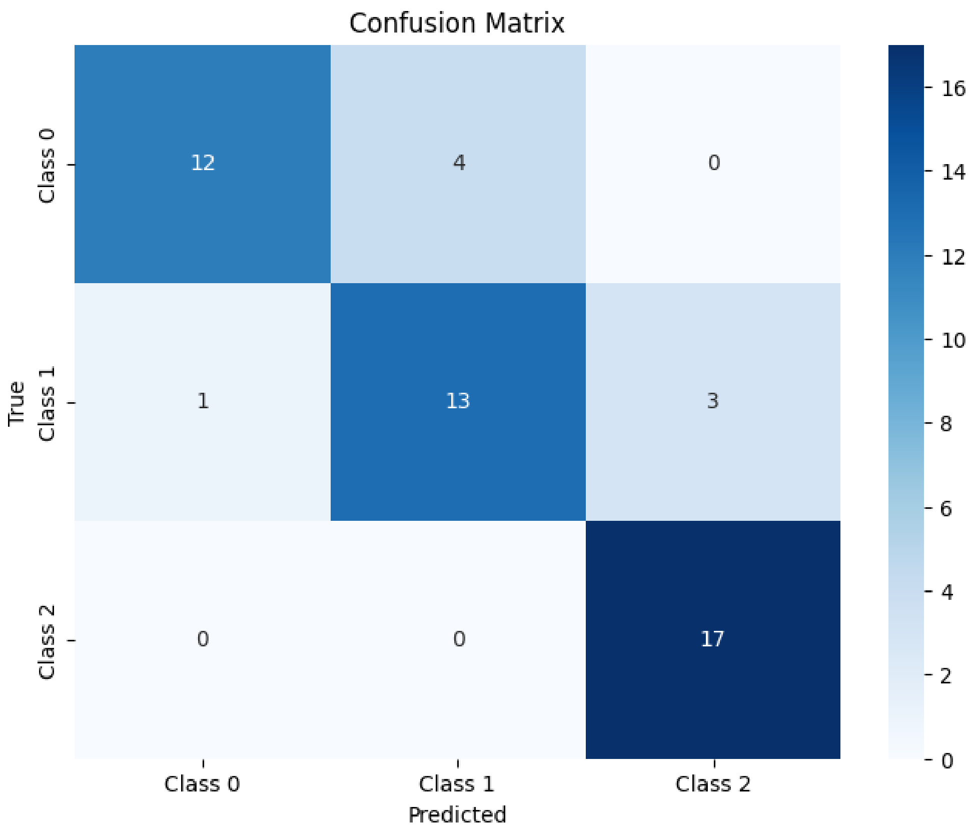
n_neighbors: 3: The best number of neighbors for classification is 3, meaning the algorithm considers the 3 nearest data points to determine the class of a new sample.

weights: 'uniform': All neighbors contribute equally to the decision (their distance doesn't affect their weight).

Test Accuracy: 98.00%

The model correctly classified 98% of the test samples.

The dataset is likely well-separated in feature space, and the preprocessing steps (scaling, SMOTE, PCA) improved the model's performance.Using 3 neighbors with equal weighting and Euclidean distance appears tobe a good fit for the structure of this data.



Class 0:

Correctly predicted 12 samples as Class 0.

Misclassified 4 samples as Class 1.

Misclassified 0 samples as Class 2.

Class 1:

Correctly predicted 13 samples as Class 1.

Misclassified 1 sample as Class 0.

Misclassified 3 samples as Class 2.

Class 2:

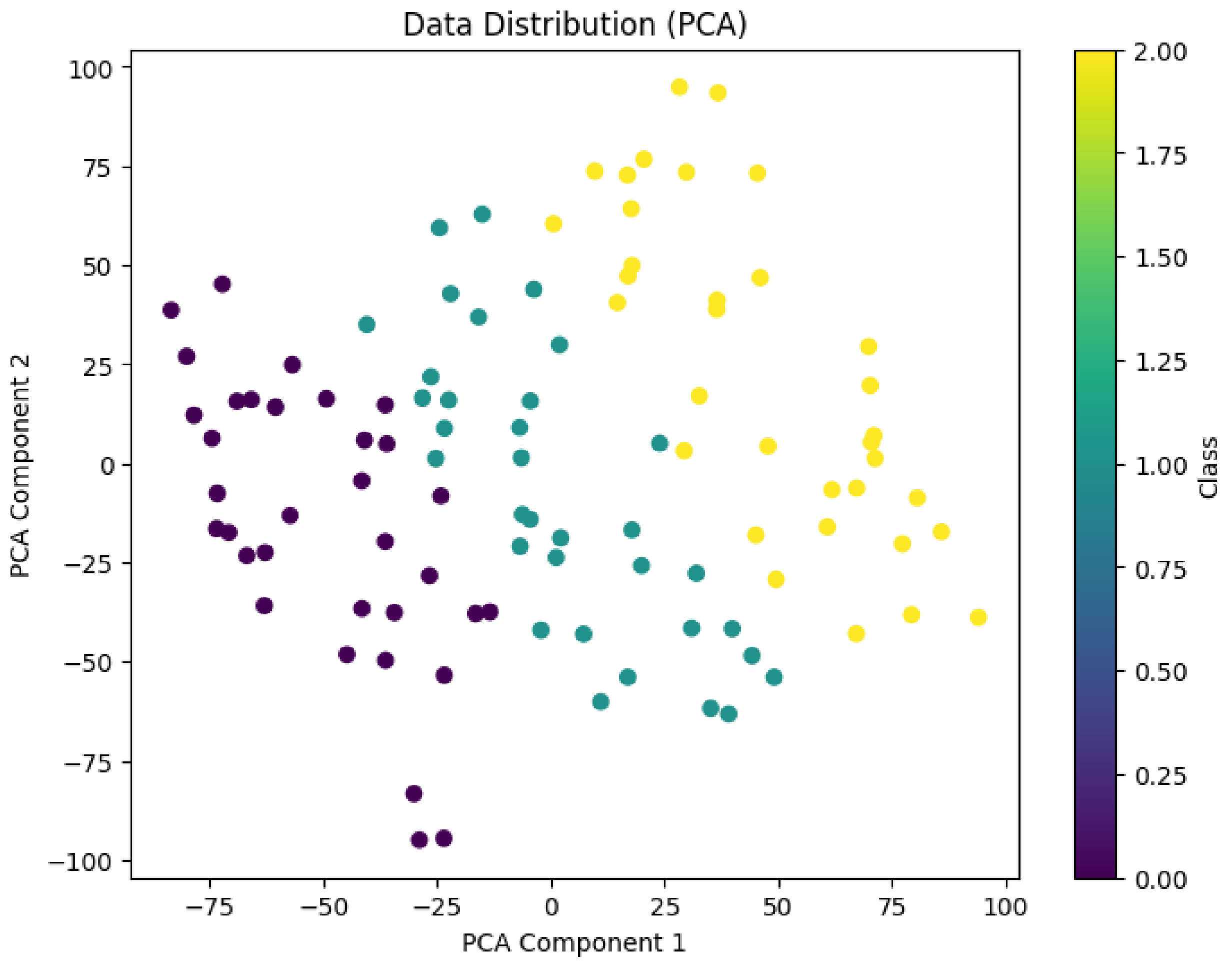
Correctly predicted 17 samples as Class 2.

Misclassified 0 samples as Class 0 or Class 1.

Summary:

The model performs well overall, especially for Class 2, where all predictions are correct.

There are minor misclassifications between Class 0 and Class 1, and a few between Class 1 and Class 2.



We can clearly see that the data points are classified distinctly: purple represents Class 0, green represents Class 1, and yellow represents Class 2.

Conclusion

The KNN classifier, after parameter tuning and preprocessing, performed well on the given dataset. The use of SMOTE helped mitigate class imbalance, and PCA reduced the complexity of the model without losing significant information. Future improvements could involve trying other classification models and experimenting with different feature selection or dimensionality reduction techniques.