S5 -AI DS Lab - ADL331

Experiment 6:

Use the Naive Bayesian Classifier

Assuming a set of documents that need to be classified, use the naïve Bayesian Classifier model to perform this task. Calculate the accuracy, precision, and recall for your data set.

Algorithm:

1.Start

2.Import necessary libraries

- Import `pandas` for handling data.
- Import `load_iris` from `sklearn.datasets` to load the Iris dataset.
- Import `train_test_split` from `sklearn.model_selection` to split data.
- Import `GaussianNB` from `sklearn.naive_bayes` for the Naive Bayes model.
- Import `accuracy_score`, `classification_report`, and `confusion_matrix` from `sklearn.metrics` for model evaluation.

3.Load the dataset

- Load the Iris dataset using `load_iris()` and store it in a variable.

4.Convert dataset to DataFrame

- Create a DataFrame from the dataset using `pandas.DataFrame()` for better visualization and understanding.

5.Define features and target

- Set x as the feature variables (input data).
- Set y as the target variable (class labels).

6.Split the dataset

- Use `train_test_split()` to divide the dataset into training and testing sets (e.g., 60% training, 40% testing).

7.Create and train the model

- Initialize the Naive Bayes model using `GaussianNB()`.

- Train the model using `model.fit(x_train, y_train)`.

8.Make predictions

- Use `model.predict(x_test)` to predict class labels for the test data.

9.Evaluate the model

- Calculate the accuracy using `accuracy_score(y_test, y_pred)`.
- Display the confusion matrix using `confusion_matrix(y_test, y_pred)`.
- Display the classification report using `classification_report(y_test, y_pred)`.

10.Display results

- Print the model's accuracy, confusion matrix, and classification report.

11.Stop

Code:

```
import pandas as pd

from sklearn.datasets import load_iris

iris=load_iris()

data=pd.DataFrame(data=iris.data,columns=iris.feature_names)

data.head()

x=iris.data

y=iris.target

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.4,random_state=
1)

from sklearn.naive_bayes import GaussianNB

model=GaussianNB()

model.fit(x_train, y_train)

y_pred=model.predict(x_test)

print("The accuracy of the model is:
\n",accuracy_score(y_test,y_pred)*100,"%")

print("confusion matrix of the model is: \n",confusion_matrix(y_test,y_pred))

print("The classification report of the model is:
\n",classification_report(y_test,y_pred))
```

Output

Experiment 7:

Decision Tree mode to perform task

Assuming a set of data that need to be classified, use a decision tree model to perform this task. Preferably use any dataset like medical or others to evaluate the accuracy.

Algorithm:

1. Start
2. Import required libraries
    - Import `pandas` for loading and manipulating data.
    - Import `train_test_split` from `sklearn.model_selection` to split data into training and testing sets.
    - Import `DecisionTreeClassifier` from `sklearn.tree` to build the decision tree model.
    - Import `classification_report`, `confusion_matrix`, and `accuracy_score` from `sklearn.metrics` for evaluating the model.
3. Load the dataset
    - Read the dataset file `pima_indians_diabetes_sample.csv` using `pd.read_csv()`.
    - Display the first few rows using `pima.head()` to verify that the dataset is loaded correctly.
4. Select features and target variable
    - Define the feature columns as: `['insulin', 'bmi', 'age', 'glucose', 'bp', 'pedigree']`.
    - Set X as the feature set (independent variables).
    - Set y as the target variable (dependent variable) — the `'label'` column indicating diabetes presence (1) or absence (0).
5. Split the dataset
    - Divide the dataset into training and testing sets using `train_test_split()`.
    - Use 70% of the data for training and 30% for testing.
    - Set a `random_state` (e.g., 1) for reproducibility.
6. Create and train the model
    - Initialize the decision tree classifier using `DecisionTreeClassifier()`.
    - Train (fit) the model using the training data (`X_train`, `y_train`).
7. Make predictions

- Use the trained model to predict outcomes for the test data using `clf.predict(X_test)`.

8. Evaluate the model

   - Generate the confusion matrix using `confusion_matrix(y_test, y_pred)` to compare actual and predicted labels.
   - Generate the classification report using `classification_report(y_test, y_pred)` to display precision, recall, and F1-score.
   - Compute the accuracy using `accuracy_score(y_test, y_pred)`.

9. Display evaluation results

   - Print the confusion matrix, classification report, and accuracy score to interpret model performance.

10. Visualize the Decision Tree

    - Use `export_graphviz()` to export the trained decision tree structure into DOT format.
    - Use `pydotplus` and `Graphviz` to generate a visual representation of the decision tree.
    - Save the visualization as an image file (e.g., `decision_tree.png`).
    - Display the image in the notebook using `Image(graph.create_png())`.

11. Stop

Code:

```
# Step 1: Import required libraries

import pandas as pd  # For data loading and manipulation

from sklearn.model_selection import train_test_split  # For splitting the
dataset

from sklearn.tree import DecisionTreeClassifier  # To build the decision tree
classifier

from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score  # To evaluate the model


# Step 2: Load the uploaded dataset (already includes headers)

pima = pd.read_csv("/content/pima_indians_diabetes_sample.csv")  # Read the
CSV file into a DataFrame

print("Dataset Preview:\n", pima.head())  # Display the first few rows of the
dataset


# Step 3: Select features and target variable

feature_cols = ['insulin', 'bmi', 'age', 'glucose', 'bp', 'pedigree']  #
Define which columns to use as features

X = pima[feature_cols]   # Extract the feature columns
```

```
y = pima['label']        # Extract the target column ('label') which indicates
diabetes (1) or not (0)

# Step 4: Split the dataset into training (70%) and testing (30%) sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=1)  # Split the data

# Step 5: Create the Decision Tree Classifier model and fit it to the training
data

clf = DecisionTreeClassifier()  # Create the classifier object

clf = clf.fit(X_train, y_train)  # Train the classifier using the training
data

# Step 6: Make predictions using the testing data

y_pred = clf.predict(X_test)  # Predict the output for test data

# Step 7: Evaluate the model

# Compute and display the confusion matrix

result = confusion_matrix(y_test, y_pred)

print("Confusion Matrix:\n",result)  # Prints the matrix showing true vs.
predicted values

# Compute and display the classification report (precision, recall, f1-score)

result1 = classification_report(y_test, y_pred)

print("Classification Report:\n",result1)  # Prints metrics for model
evaluation

# Compute and display the accuracy of the model

result2 = accuracy_score(y_test,  y_pred)

print("Accuracy:\n", result2)  # Prints overall accuracy of prediction

# Step 8: Visualize the Decision Tree

# This part uses Graphviz to convert the decision tree into an image


# Import necessary modules for visualization

from sklearn.tree import export_graphviz  # To export the tree structure in
DOT format

import six  # Compatibility module

import sys

sys.modules['sklearn.externals.six'] = six  # Patch to support old sklearn
versions


from sklearn.externals.six import StringIO  # Used to capture DOT data

from IPython.display import Image  # For displaying image in Colab

import pydotplus  # To create the graph from DOT data


# Export the trained decision tree to DOT format

dot_data = StringIO()

export_graphviz(clf, out_file=dot_data,

                filled=True,             # Fill the nodes with color

                rounded=True,            # Round the edges of boxes
```

```
                special_characters=True, # Support for symbols
                feature_names=feature_cols,  # Feature names for splits
                class_names=['0','1'])       # Class names for target labels
# Create graph from DOT dataz
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
# Save the decision tree as a PNG image
plt.savefig("decision_tree.png", dpi=300, bbox_inches='tight')
# Display the decision tree image in notebook
Image(graph.create_png())
```

Output:

Experiment 8:

Hill climbing algorithm

Implement a program to perform Hill climbing algorithm.

Algorithm:

1.Start

2.Initialize

- Import the required modules: `math` and `random`.
- Define a list of cities with random coordinates.

3.Define helper functions

- Distance(city1, city2):
  Calculate the Euclidean distance between two cities.
- Total_distance(tour, cities):
  Compute the total distance of the tour (path covering all cities and returning to start).
- Swap_cities(tour):
  Create a new tour by swapping two randomly chosen cities.

4.Initialize the algorithm

- Create an initial random tour of all cities.
- Calculate its total distance and store it as the current best distance.

5.Repeat for a fixed number of iterations (`max_iter`)

- Generate a new tour by swapping two cities.
- Calculate the total distance of the new tour.
- If the new distance is shorter than the current distance:
  - Accept the new tour as the current tour.

- Update the current best distance.
- Print the improved distance and iteration number.

6.After all iterations

- Return the best tour and its total distance as the final result.

7.Display results

- Print the best tour and the best (shortest) distance found.

8.Stop

Code:

```
import math
import random

# Distance between two cities
def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

# Total distance of a tour
def total_distance(tour, cities):
    dist = 0
    for i in range(len(tour)):
        dist += distance(cities[tour[i]], cities[tour[(i + 1) % len(tour)]])
    return dist

# Swap two cities in the tour
def swap_cities(tour):
    new_tour = tour[:]
    i, j = random.sample(range(len(tour)), 2)
    new_tour[i], new_tour[j] = new_tour[j], new_tour[i]
    return new_tour

# Hill climbing algorithm
def hill_climbing(cities, max_iter):
    tour = list(range(len(cities)))
    random.shuffle(tour)
    current_distance = total_distance(tour, cities)

    for iteration in range(max_iter):
        new_tour = swap_cities(tour)
        new_distance = total_distance(new_tour, cities)

        if new_distance < current_distance:
            tour = new_tour
            current_distance = new_distance
            print(f"Iteration {iteration+1}: Distance = {current_distance}")

    return tour, current_distance

# Main program
if __name__ == "__main__":
    cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in
range(10)]
    best_tour, best_distance = hill_climbing(cities, 10)
    print("Best tour found:", best_tour)
    print("Best distance found:", best_distance)
```

Output

Experiment 9:

Correlation and Covariance
Write a program to find Correlation and Covariance between different features of a dataset in csv format.

Algorithm:

1.Start

2.Import the required library

- Import the `pandas` library for data handling and statistical analysis.

3.Load the dataset

- Read the file `iris.csv` using `pd.read_csv()` and store it in a DataFrame `df`.
- Display the first few rows using `df.head()` to verify the data.

4.Set index column

- Use `set_index("Id")` to make the `'Id'` column the index for better organization.

5.Select numerical features

- Extract the numerical columns: `SepalLengthCm`, `SepalWidthCm`, `PetalLengthCm`, and `PetalWidthCm` into variable `x`.

6.Calculate covariance

- Use `x.cov()` to find how two variables vary together.

7.Calculate correlation

- Use `x.corr(method='pearson')` to measure the strength and direction of the linear relationship between variables.

8.Display results

- Print the covariance and correlation matrices for analysis.

9.Stop

Code:

```
# Import pandas

import pandas as pd

# Step 1: Load the dataset
```

```
df = pd.read_csv("iris.csv")
# Step 2: Display first 5 rows
print("First five rows of the dataset:")
print(df.head())
# Step 3: Set 'Id' as index
df.set_index("Id", inplace=True)
print("\nDataset after setting 'Id' as index:")
print(df.head())
# Step 4: Select numerical features
x = df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
# Step 5: Covariance
print("\nCovariance between features:")
print(x.cov())
# Step 6: Correlation
print("\nCorrelation between features (Pearson method):")
print(x.corr(method='pearson'))
```

Experiment 10

Program to implement feature reduction using PCA

Write a program to implement feature reduction using PCA. Calculate the covariance between features to find the optimal number of PCA components

Algorithm:

1.Start

2.Import Required Libraries

- Import `pandas` and `numpy` for data handling.
- Import `matplotlib.pyplot` and `seaborn` for visualization.
- Import `load_breast_cancer` from `sklearn.datasets` to load the dataset.
- Import `StandardScaler` from `sklearn.preprocessing` for feature standardization.
- Import PCA from `sklearn.decomposition` for principal component analysis.

3.Load the Dataset

- Load the Breast Cancer dataset using `load_breast_cancer()`.
- Convert it into a DataFrame with feature names.
- Display the first few rows to verify the dataset.

4.Standardize the Dataset

- Use `StandardScaler()` to scale the features to have mean 0 and variance 1.
- Fit and transform the dataset to obtain standardized features.

5.Apply PCA with 2 Components

- Initialize `PCA(n_components=2)` and fit it to the standardized data.
- Transform the data to obtain the first two principal components.
- Print a command-line message indicating completion.

6.Visualize 2D PCA Results

- Plot a 2D scatter plot using the first two principal components.
- Color the points based on the target labels to differentiate classes.

7.Display PCA Components Heatmap

- Extract the principal component coefficients into a DataFrame.
- Visualize the component contributions using a heatmap.

8.Apply PCA with 3 Components

- Initialize `PCA(n_components=3)` and fit it to the standardized data.
- Transform the data to obtain three principal components.

9.Visualize 3D PCA Results

- Plot a 3D scatter plot using the three principal components.
- Color the points based on target labels.

10.Calculate Correlation Matrix

- Compute and display the Pearson correlation matrix for the original features to understand feature relationships.

11.Apply PCA with 30 Components for Variance Analysis

- Fit `PCA(n_components=30)` to the standardized data.
- Display the explained variance ratio for each component to analyze how much variance is captured.

12.Display Shapes of Datasets

- Print the shapes of the original dataset, standardized features, and PCA-transformed dataset.

13.Stop

Code:

```
# ========================================
# Step 1: Import required libraries
# ========================================
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer

# ========================================
# Step 2: Load the dataset
# ========================================
cancer = load_breast_cancer()
print("Keys of the dataset:\n", cancer.keys())    # Command line output
df = pd.DataFrame(cancer['data'], columns=cancer['feature_names'])
print("\nFirst five rows of dataset:\n", df.head())    # Display sample data

# ========================================
# Step 3: Standardize the dataset
# ========================================
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(df)
scaled_features = scaler.transform(df)
print("\nData standardized successfully!")    # Command line message

# ========================================
# Step 4: Apply PCA with 2 components
# ========================================
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(scaled_features)
x_pca = pca.transform(scaled_features)
```

```
print("\nPCA transformation with 2 components completed.")  # Command line output
# ========================================
# Step 5: Visualize 2D PCA results
# ========================================
plt.figure(figsize=(8,6))
plt.scatter(x_pca[:,0], x_pca[:,1], c=cancer['target'], cmap='plasma')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.title('2D PCA Plot - Breast Cancer Dataset')
plt.show()

# ========================================
# Step 6: Display PCA Components Heatmap
# ========================================
df_comp = pd.DataFrame(pca.components_, columns=cancer['feature_names'])
sns.heatmap(df_comp, cmap='plasma')
plt.title('PCA Components Heatmap')
plt.show()

# ========================================
# Step 7: Apply PCA with 3 components and visualize
# ========================================
pca = PCA(n_components=3)
pca.fit(scaled_features)
x_pca = pca.transform(scaled_features)
print("\nPCA transformation with 3 components completed.")

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x_pca[:,0], x_pca[:,1], x_pca[:,2], c=cancer['target'], cmap='plasma')
ax.set_xlabel('First Principal Component')
ax.set_ylabel('Second Principal Component')
ax.set_zlabel('Third Principal Component')
plt.title('3D PCA Plot - Breast Cancer Dataset')
plt.show()
# ========================================
# Step 8: Display correlation matrix
# ========================================
print("\nCorrelation matrix using Pearson method:")
```

```
print(df.corr(method='pearson'))


# ========================================
# Step 9: Apply PCA with 30 components to analyze variance
# ========================================
pca = PCA(n_components=30)
pca.fit(scaled_features)
print("\nExplained variance ratio for 30 components:")
print(pca.explained_variance_ratio_)

# ========================================
# Step 10: Display shapes of datasets
# ========================================
print("\nOriginal dataset shape:", df.shape)
print("Scaled features shape:", scaled_features.shape)
x_pca = pca.transform(scaled_features)
print("Transformed dataset shape after PCA:", x_pca.shape)
```

Output