Lab: Data Mining

Aim: To have a practical knowledge on the important concept of Data Mining

Requirements: Google Colab

1. Clustering:

a. Install and Import libraries:

```
pip install palmerpenguins  # (if it isn't working, use the next 2 line)
#import sys
#!{sys.executable} -m pip install palmerpenguins

from palmerpenguins import load_penguins  # For penguins dataset
import pandas as pd                       # For dataframes
import matplotlib.pyplot as plt           # For plotting functions
import seaborn as sns                     # For additional plotting functions
from sklearn.cluster import KMeans                # For k-Means
from sklearn.model_selection import GridSearchCV  # For grid search
from sklearn.metrics import silhouette_score      # For metrics and scores
from sklearn.preprocessing import StandardScaler  # For standardizing data
```

b. Load and prepare data: Following steps are used to prepare the data:

i. Load the `penguins` dataset in variable `df`
ii. Remove the `island`, `year`, and `sex` variables
iii. Rename the class variable `species` as `y`
iv. Drop all rows with `NaN`
v. Display the first 5 rows of `df`

```
# Loads the penguins dataset
df = load_penguins()
# Drop variables and NaN cases, rename variable
df = df.drop(['island', 'year', 'sex'], axis=1) \
    .dropna() \
    .rename(columns={'species': 'y'})
# Displays the first 5 rows of data
df.head()
```

c. Explore the data: Visualize various aspects of penguins dataset.

i. Bar Plot of Class Variable: `sns.countplot(x='y', data=df)`
ii. Scatter Plots and Density Plots for Feature Pairs:

```
# Creates a grid using Seaborn's PairGrid()
g = sns.PairGrid(
    df,
    vars=['bill_length_mm', 'bill_depth_mm', 'flipper_length_mm', 'body_mass_g']
,
    hue='y',
    diag_sharey=False,
    palette=["red", "green", "blue"])

# Adds histograms on the diagonal
g.map_diag(plt.hist)
# Adds density plots above the diagonal
g.map_upper(sns.kdeplot)
# Adds scatterplots below the diagonal
g.map_lower(sns.scatterplot)
# Adds a legend
```

```
g.add_legend()
```

d. Prepare data:

```
# Separates the class variable in y
y = df.y
# Removes the y column from df
df = df.drop('y', axis=1)
# Standardizes df
df = pd.DataFrame(
    StandardScaler().fit_transform(df),
    columns=df.columns)
# Displays the first 5 rows of df
df.head()
```

e. **k-MEANS: Train the Model**

We'll set up a KMeans object with the following parameters:

- `n_clusters`: Total number of clusters to make.
- `random_state`: Set to one to reproduce these results.
- `init`: How to initialize the k-means centers; we'll use k-means++.
- `n_init`: Number of times k-means would be run; the model returned would have the minimum value of inertia.

A few other attributes of the KMeans object:

- `cluster_centers_`: Stores the discovered cluster centers.
- `labels_`: Label of each instance.
- `inertia`: Sum of square of distances of each instance from its corresponding center.
- `n_iter`: Number of iterations run to find the centers.

```
# Sets up the kMeans object
km = KMeans(
    n_clusters=3,
    random_state=1,
    init='k-means++',
    n_init=10)

# Fits the model to the data
km.fit(df)

# Displays the parameters of the fitted model
km.get_params()
```

f. k-Means: Visualize the Clusters

```
# Creates a scatter plot
sns.scatterplot(
    x='bill_length_mm',
    y='bill_depth_mm',
    data=df,
    hue=y,
    style=km.labels_,
    palette=["orange", "green", "blue"])
```

```
# Adds cluster centers to the same plot
plt.scatter(
    km.cluster_centers_[:,0],
    km.cluster_centers_[:,1],
    marker='x',
    s=200,
    c='red')
```

g. **k-means: optimize via silhouette scores**

The main challenge in k-means is to find the optimal number of clusters. We can set up a `GridSearchCV` object to search for the optimal parameters. For `k-Mmeans`, we require a custom scorer that computes the silhouette value for different number of clusters specified by n_clusters. The custom scorer is called s2() in the code below, where it uses silhouette_score() from the `sklearn.metrics` library to compute a score for an instance X.

A silhouette score is a value in [-1,+1]. It is a means for comparing how similar an instance is to its corresponding cluster compared to its similarity with other clusters. Formally, it takes into account cohesion and separation to compute a silhouette value. A +1 or close to this score value indicates better clusters.

```
# Sets up the custom scorer
def s2(estimator,X):
    return silhouette_score(X, estimator.predict(X))
# List of values for the parameter `n_clusters`
param = range(2,10)
# KMeans object
km = KMeans(random_state=0, init='k-means++')
# Sets up GridSearchCV object and stores in grid variable
grid = GridSearchCV(
    km,
    {'n_clusters': param},
    scoring=s2,
    cv=2)
# Fits the grid object to data
grid.fit(df)
# Accesses the optimum model
best_km = grid.best_estimator_
# Displays the optimum model
best_km.get_params()
```

h. Plot of Scores for Different Number of Clusters

```
# Plot mean_test_scores vs. n_clusters
plt.plot(
    param,
    grid.cv_results_['mean_test_score'])
# Draw a vertical line, where the best model is
plt.axvline(
    x=best_km.n_clusters,
    color='red',
    ls='--')

# Adds labels to the plot
```

```
plt.xlabel('Total Centers')
plt.ylabel('Silhouette Score')
```

i. Visualize the Best Model

```
# Creates a scatter plot
sns.scatterplot(
    x='bill_length_mm',
    y='bill_depth_mm',
    data=df,
    hue=y,
    style=best_km.labels_,
    palette=['orange', 'green', 'blue'])

# Adds cluster centers to the same plot
plt.scatter(
    best_km.cluster_centers_[:, 0],
    best_km.cluster_centers_[:, 1],
    marker='x',
    s=200,
    c='red')
```

2. Classification

a. Import libraries

```
import pandas as pd                                    # For dataframes
import matplotlib.pyplot as plt                        # For plotting data
import seaborn as sns                                  # For plotting data
from sklearn.model_selection import train_test_split   # For train/test splits
from sklearn.model_selection import GridSearchCV      # For parameter optimization
from sklearn.neighbors import KNeighborsClassifier    # For kNN classification
from sklearn.metrics import plot_confusion_matrix     # Evaluation measure
```

b. Load and prepare data

i. Import Data

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spambase.data', header=None)

df.head()
```

ii. Rename Variables

- Assign a name to all attributes as X0, X1, ..., X56.
- Assign y to the class variable (the last column of df).
- Display the first 5 rows.

```
# Sequentially renames all attribute columns and renames the last column to 'y'
df.columns = ['X' + str(i) for i in range(0, len(df.columns) - 1)] + ['y']

# Shows the first few lines of the data
df.head()
```

iii. Split Data

- To prepare the dataset for classification, we have to split it into train and test sets.
- train_test_split() splits the data into train and test.
- In the arguments list, the data matrix consists of all attribute columns. Extract columns X0, X1, ..., X56 with df.filter(regex='\d'). The filter keeps only the names that have a numeric character in them.
- Specify the target variable as df.y.
- Set up trn and tst dataframes.

```
# Specifies X by filtering all columns with a number in name
X_trn, X_tst, y_trn, y_tst = train_test_split(
    df.filter(regex='\d'),
    df.y,
    test_size=0.30,
    random_state=1)
# Creates the training dataset, trn
trn = X_trn
trn['y'] = y_trn
# Creates the testing dataset, tst
tst = X_tst
tst['y'] = y_tst
```

iv. Explore training data

- Bar Plot of Class Variable

```
sns.countplot(x='y', data=trn)
```

- Explore Attribute Variables : Select four arbitrary features and get paired plots

```
# Creates a grid using Seaborn's PairGrid()
g = sns.PairGrid(
    trn,
    vars=['X5', 'X20', 'X25', 'X53'],
    hue='y',
    diag_sharey=False,
    palette=['red', 'green'])
# Adds histograms on the diagonal
g.map_diag(plt.hist)
# Adds density plots above the diagonal
g.map_upper(sns.kdeplot)
# Adds scatterplots below the diagonal
g.map_lower(sns.scatterplot)
# Adds a legend
g.add_legend(title='Spam')
```

v. Prepare data

```
# Separates the attributes X0-X56 into X_trn
X_trn = trn.filter(regex='\d')
# Separates the class variable into y_trn
y_trn = trn.y
# Separates the attributes X0-X56 into X_tst
X_tst = tst.filter(regex='\d')
# Separates the class variable into y_tst
y_tst = tst.y
# Class labels
spam = ['Not Spam','Spam']
trn.head()
```

c. kNN: train model

```
# Sets up a kNN model and fits it to data

knn = KNeighborsClassifier(n_neighbors=5) \
    .fit(X_trn, y_trn)
```

d. Mean Accuracy on Training Data

```
print('Accuracy on training data: '  + str("{:.2%}".format(knn.score(X_trn, y_trn))))
```

e. Optimize the kNN Model: The challenge in training a kNN model is to determine **the optimal number of neighbors.** To find the optimal parameters, GridSearchCV object can be used.

```
# Sets up the kNN classifier object
knn = KNeighborsClassifier()
# Search parameters
param = range(3, 15, 2)
# Sets up GridSearchCV object and stores it in grid variable
grid = GridSearchCV(
    knn,{'n_neighbors': param})
# Fits the grid object and gets the best model
best_knn = grid \
    .fit(X_trn,y_trn) \
    .best_estimator_
# Displays the optimum model
best_knn.get_params()
```

f. Plot the Accuracy by Neighbors Parameter

```
# Plots mean_test_scores vs. total neighbors
plt.plot(
    param,
    grid.cv_results_['mean_test_score'])

# Adds labels to the plot
plt.xticks(param)
plt.ylabel('Mean CV Score')
plt.xlabel('n_neighbors')

# Draws a vertical line where the best model is
plt.axvline(
    x=best_knn.n_neighbors,
    color='red',
    ls='--')
```

g. Test model: we'll evaluate the accuracy of the trained kNN model on the test set. A good evaluation measure is the confusion matrix that gives the fraction of true positives, true negatives, false positives, and false negatives.

```
plot_confusion_matrix(
    best_knn, X_tst, y_tst,
    display_labels=spam,
    normalize='true')
```

h. Calculate Mean Accuracy on Testing Data

```
print( 'Accuracy on testing data: '
+ str("{:.2%}".format(best_knn.score(X_tst, y_tst))))
```

Your Turn!

- Do all the step that we discussed above for Clustering and Classification

1. Use Iris flower data and cluster them using K-Means
    - Download the Iris flower data: https://archive.ics.uci.edu/ml/datasets/iris

2. Use the breast cancer data and use the KNN determine whether they are benign or malignant.
    - Download the breast cancer data: https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)