```
# Hierarchical Clustering

'''
# `CRISP-ML(Q)` process model describes six phases:

# 1. Business and Data Understanding
# 2. Data Preparation
# 3. Model Building
# 4. Evaluation
# 5. Deployment
# 6. Monitoring and Maintenance

# Business Problem:
Students have to evaluate a lot of factors before taking a decision
to join a university for their higher education requirements.

# High Level Solution:
Logically grouping the available universities will allow understanding the characteristics of each group.

# Objective(s): Maximize the convenience of the admission process
# Constraint(s): Minimize the brain drain

Success Criteria

# Business Success Criteria: Reduce the application process time from anywhere between 20% to 40%
# ML Success Criteria: Achieve Silhouette coefficient of at least 0.6
# Economic Success Criteria: US Higher education department will see an increase in revenues by at least 30%

# HLD - DAR - DLD (Data Pipeline & Model Pipeline)

'''
# Data Understanding:
# Data Sources - Data Collection - Data Storage - EDA

# Data:
# The university details are obtained from the US Higher Education Body and is publicly available for students to access.
#
# Data Dictionary:
# - Dataset contains 25 university details
# - 7 features are recorded for each university
#
# Meta Data Description: (Features, Description of features, Units of measure, Values within each feature)
# - Univ - University Name
# - State - Location (state) of the university
# - SAT - Cutoff SAT score for eligibility
# - Top10 - % of students who ranked in the top 10 in their previous academics
# - Accept - % of students admitted to the universities
# - SFRatio - Student to Faculty ratio
# - Expenses - Overall cost in USD
# - GradRate - % of students who graduate
```

```
# Code modularity

# Install the required packages if not present already
# pip install sweetviz
# pip install py-AutoClean
# pip install clusteval
# pip install sqlalchemy
# pip install pymysql

# Importing required packages

import pandas as pd # Importing Pandas library for data manipulation
import numpy as np # Importing NumPy library for numerical computations
import matplotlib.pyplot as plt # Importing Matplotlib library for plotting

import sweetviz # Importing Sweetviz library for automated EDA (Exploratory Data Analysis)
from AutoClean import AutoClean # Importing AutoClean library for automated data cleaning

from sklearn.preprocessing import MinMaxScaler # Importing MinMaxScaler for feature scaling
from sklearn.pipeline import make_pipeline # Importing make_pipeline for creating a pipeline of preprocessing steps

from scipy.cluster.hierarchy import linkage, dendrogram # Importing functions for hierarchical clustering
from sklearn.cluster import AgglomerativeClustering # Importing AgglomerativeClustering for hierarchical clustering most of the ml model come under sklearn

from sklearn import metrics # Importing metrics module from scikit-learn for evaluating clustering performance like (sillhout technique)
from clusteval import clusteval # Importing clusteval library for cluster evaluation

from sqlalchemy import create_engine, text # Importing create_engine and text from sqlalchemy for database interaction
from urllib.parse import quote
# Reading the dataset from an Excel file into a Pandas DataFrame
uni = pd.read_excel(r"C:/Users/Bharani Kumar/Desktop/Data Analytics/Clustering/University_Clustering.xlsx")

# Credentials to connect to Database
user = 'root' # user name
pw = 'cutelucky@575' # password
db = 'univ_db' # database name
engine = create_engine(f"mysql+pymysql://{user}:{pw}@localhost/{db}")

# to_sql() - function to push the dataframe onto a SQL table.
uni.to_sql('univ_tbl', con = engine, if_exists = 'replace', chunksize = 1000, index = False)


###### To read the data from MySQL Database
sql = 'select * from univ_tbl;'
df = pd.read_sql_query(text(sql), engine.connect())

# Data types
df.info()
```

```python
# EXPLORATORY DATA ANALYSIS (EDA) / DESCRIPTIVE STATISTICS
df.describe() # Generating descriptive statistics of the DataFrame 'df', including count, mean, std, min, max, etc.

# AutoEDA
# D-Tale
# pip install dtale
import dtale

# Display the DataFrame using D-Tale
d = dtale.show(df, host = 'localhost', port = 8000)

# Open the browser to view the interactive D-Tale dashboard
d.open_browser()

# Data Preprocessing

# **Cleaning Unwanted columns**
# UnivID is the identity to each university.
# Analytically it does not have any value (Nominal data).
# We can safely ignore the UnivID column by dropping the column.

df.drop(['UnivID'], axis = 1, inplace = True) # Dropping the column 'UnivID' from the DataFrame 'df'
df.info() # Displaying concise summary of DataFrame 'df', including the number of non-null values and data types of each column

# EDA report highlights:
# ------------------------
# Missing Data: Identified Missing Data in columns: SAT, GradRate
# Outliers: Detected exceptional values in 4 columns: SAT, Top10, Accept, SFRatio
# Encoding: 'State' is categorical data that needs to be encoded into numeric values

# Data Preprocessing
# ----------------------------------------------------------------------------
# Auto Preprocessing and Cleaning
from AutoClean import AutoClean
# Creating an instance of AutoClean class and defining a cleaning pipeline
clean_pipeline = AutoClean(
df.iloc[:, 1:], # Selecting all rows and columns except the first one ('UnivID') for cleaning
mode = 'manual', # Setting the cleaning mode to 'manual'
missing_num = 'auto', # Specifying automatic handling of missing numerical values
outliers = 'winz', # Specifying Winsorization method for outlier handling
encode_categ = 'auto' # Specifying automatic encoding of categorical variables
)

help(AutoClean) # Displaying the documentation or help for the AutoClean class

# Missing values = 'auto': AutoClean first attempts to predict the missing values with Linear Regression
# outliers = 'winz': outliers are handled using winsorization
# encode_categ = 'auto': Label encoding performed (if more than 10 categories are present)

# Obtaining the cleaned DataFrame by applying the cleaning pipeline to the original DataFrame
df_clean = clean_pipeline.output

# Displaying the first few rows of the cleaned DataFrame to inspect the changes
```

```
df_clean.head()

# #### Drawback with this approach: If there are more than 10 categories, then Autoclean performs label encoding.

df_clean.drop(['State'], axis = 1, inplace = True) # Dropping the 'State' column from the cleaned DataFrame 'df_clean'
df_clean.head() # Displaying the first few rows of the updated cleaned DataFrame

# ----------------------------------------------------------------------

# Normalization/MinMax Scaler - To address the scale differences

# Python Pipelines
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import MinMaxScaler

df_clean.info() # Displaying concise summary of the cleaned DataFrame 'df_clean', including the number of non-null values and data types of each column

cols = list(df_clean.columns) # Creating a list of column names from the cleaned DataFrame 'df_clean'
print(cols) # Printing the list of column names to inspect them

# Creating a pipeline using make_pipeline to apply MinMaxScaler for feature scaling
pipe1 = make_pipeline(MinMaxScaler())

# Train the data preprocessing pipeline on data
# Applying the pipeline 'pipe1' to transform the cleaned DataFrame 'df_clean' and storing the transformed data in a new DataFrame 'df_pipelined'
df_pipelined = pd.DataFrame(pipe1.fit_transform(df_clean), columns = cols, index = df_clean.index)

# Displaying the first few rows of the transformed DataFrame 'df_pipelined' to inspect the changes
df_pipelined.head()

# Generating descriptive statistics of the transformed DataFrame 'df_pipelined'
# The scale of the data is normalized to have a minimum value of 0 and a maximum value of 1 due to MinMaxScaler
df_pipelined.describe()

###### End of Data Preprocessing ######
# ----------------------------------------------------------------------

# Save Preprocessed data into SQL Mandatory

user = 'user1' # user name
pw = 'user1' # password
db = 'univ_db' # database name
engine = create_engine(f"mysql+pymysql://{user}:{pw}@localhost/{db}")
df_pipelined.to_sql('processeddata', con = engine, if_exists = 'replace', chunksize = 1000, index = False)


######### Model Building #########
```

```
# CLUSTERING MODEL BUILDING

# Hierarchical Clustering - Agglomerative Clustering

# from scipy.cluster.hierarchy import linkage, dendrogram
# from sklearn.cluster import AgglomerativeClustering
# import matplotlib.pyplot as plt
# get_ipython().run_line_magic('matplotlib', 'inline') --- if running in jupyter notebook

plt.figure(1, figsize = (16, 8)) # Creating a new figure with specified size for the dendrogram plot

# Generating a dendrogram plot using hierarchical clustering with complete linkage method
tree_plot = dendrogram(linkage(df_pipelined, method = "complete"))

plt.title('Hierarchical Clustering Dendrogram') # Setting the title of the dendrogram plot
plt.xlabel('Index') # Setting the label for x-axis
plt.ylabel('Euclidean distance') # Setting the label for y-axis
plt.show() # Displaying the dendrogram plot


# Applying AgglomerativeClustering and grouping data into 3 clusters
# based on the above dendrogram as a reference
# Creating an instance of AgglomerativeClustering with parameters:
# - n_clusters: number of clusters set to 3
# - affinity: distance metric set to 'euclidean'
# - linkage: linkage criterion set to 'complete'
hc1 = AgglomerativeClustering(n_clusters = 3, metric = 'euclidean', linkage = 'complete')

# Fitting the AgglomerativeClustering model to the data and predicting the cluster labels for each
sample
y_hc1 = hc1.fit_predict(df_pipelined)

# Displaying the cluster labels assigned by the AgglomerativeClustering model
y_hc1

# Accessing the cluster labels directly from the AgglomerativeClustering model
hc1.labels_

# Converting the cluster labels into a Pandas Series for further analysis
cluster_labels = pd.Series(hc1.labels_)

# Combine the labels obtained with the data
# Concatenating the cluster labels (cluster_labels) with the cleaned DataFrame (df_clean) along
the columns axis
df_clust = pd.concat([cluster_labels, df], axis = 1)

# Displaying the first few rows of the DataFrame df_clust to inspect the concatenation result
df_clust.head()

# Displaying the column names of the DataFrame df_clust
df_clust.columns

# Renaming the first column (containing cluster labels) to 'cluster' for better clarity
df_clust = df_clust.rename(columns = {0: 'cluster'})

# Displaying the first few rows of the DataFrame df_clust after renaming the column
```

```
df_clust.head()

# Clusters Evaluation

# Silhouette coefficient:
# Silhouette coefficient is a Metric, which is used for calculating
# goodness of the clustering technique, and the value ranges between (-1 to +1).
# It tells how similar an object is to its own cluster (cohesion) compared to
# other clusters (separation).
# A score of 1 denotes the best meaning that the data point is very compact
# within the cluster to which it belongs and far away from the other clusters.
# Values near 0 denote overlapping clusters.

# from sklearn import metrics
metrics.silhouette_score(df_pipelined, cluster_labels)

'''Alternatively, we can use:'''
# **Calinski Harabasz:**
# Calculating the silhouette score to evaluate the quality of clustering
# The silhouette score measures how similar an object is to its own cluster (cohesion) compared to other clusters (separation)
# Higher silhouette score indicates better clustering results
# Parameters:
# - df_pipelined: the data points used for clustering, after preprocessing and feature scaling
# - cluster_labels: the cluster labels assigned to each data point by the clustering algorithm

from sklearn.metrics import calinski_harabasz_score
calinski_harabasz = calinski_harabasz_score(df_pipelined, cluster_labels)
print("Calinski-Harabasz Score:", calinski_harabasz) # A higher CH score indicates better clustering quality.
# There is no strict range like the Silhouette Score, but higher values suggest better-defined clusters.

# **Davies-Bouldin Index:**
# Calculating the Davies-Bouldin index to evaluate the quality of clustering
# The Davies-Bouldin index measures the average similarity between each cluster and its most similar cluster, taking into account both cluster separation and cohesion
# Lower Davies-Bouldin index indicates better clustering results
# Parameters:
# - df_pipelined: the data points used for clustering, after preprocessing and feature scaling
# - cluster_labels: the cluster labels assigned to each data point by the clustering algorithm
metrics.davies_bouldin_score(df_pipelined, cluster_labels) # Lower DB scores indicate better clustering (closer to 0 is ideal).

# A DB Score of 1.28 suggests moderate cluster separation, but not very strong.
# Since your Silhouette Score is low (0.249) and Calinski-Harabasz Score is 16.98, your clustering is not optimal and might have overlapping clusters.

'''
Try a different linkage method in hierarchical clustering (ward, complete, average, single).
Optimize the number of clusters by:
Analyzing the dendrogram.
Using the Elbow Method.
Trying agglomerative clustering with different n_clusters.
Use different distance metrics, such as:
Euclidean (default, good for dense clusters).
```

Manhattan (for high-dimensional data).
Cosine (for text or sparse data).
'''

```
'''Hyperparameter Optimization for Hierarchical Clustering'''
# Experiment to obtain the best clusters by altering the parameters
# Define the parameter grid for hyperparameter tuning
param_grid = {
'n_clusters': [2, 3, 4], # Number of clusters to try
'metric': ['euclidean', 'manhattan', 'cosine'], # Distance metrics
'linkage': ['ward', 'complete', 'average', 'single'] # Linkage criteria
}

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import silhouette_score
# Custom scorer for GridSearchCV (using Silhouette Score)
def custom_scorer(estimator, X):
labels = estimator.fit_predict(X)
if len(set(labels)) == 1: # Handle case where all points are in one cluster
return -1 # Return a bad score
return silhouette_score(X, labels)

# Initialize Agglomerative Clustering
agg_clustering = AgglomerativeClustering()

# Initialize GridSearchCV
grid_search = GridSearchCV(
estimator = agg_clustering,
param_grid = param_grid,
scoring = custom_scorer,
cv = 3 # Cross-validation folds
)

# Fit the model
grid_search.fit(df_pipelined)

# Print the best parameters and best score
print("Best Parameters:", grid_search.best_params_)
print("Best Silhouette Score:", grid_search.best_score_) #we will get 0.4 this is bcz of very low
data

# Use the best model to predict clusters
best_model = grid_search.best_estimator_
labels = best_model.fit_predict(df_pipelined)

# Add cluster labels to the original DataFrame
df['Cluster'] = labels
print("\nData with Cluster Labels:")
print(df)


# ## Cluster Evaluation Library

# pip install clusteval
# Refer to link: https://pypi.org/project/clusteval
```

```
# from clusteval import clusteval
# import numpy as np

# Silhouette cluster evaluation.
# Creating an instance of clusteval for cluster evaluation using silhouette score
ce = clusteval(evaluate = 'silhouette')

# Converting the DataFrame of preprocessed and scaled data (df_pipelined) into a numpy array
df_array = np.array(df_pipelined)

# Fitting the clusteval instance to the data array to compute silhouette scores for different
numbers of clusters
ce.fit(df_array)

# Plotting the silhouette scores for different numbers of clusters
ce.plot()

## Using the report from clusteval library building 2 clusters
# Fit using agglomerativeClustering with metrics: euclidean, and linkage: ward

# Creating an instance of AgglomerativeClustering with parameters for 2 clusters, using
Euclidean distance and Ward linkage
hc_2clust = AgglomerativeClustering(n_clusters = 2, metric = 'euclidean', linkage = 'ward')

# Fitting the AgglomerativeClustering model to the data and predicting the cluster labels for each
sample
y_hc_2clust = hc_2clust.fit_predict(df_pipelined)

# Obtaining the cluster labels assigned by the AgglomerativeClustering model
hc_2clust.labels_

# Converting the cluster labels into a Pandas Series for further analysis
cluster_labels2 = pd.Series(hc_2clust.labels_)

# Concatenating the cluster labels with the cleaned DataFrame along the columns axis
df_2clust = pd.concat([cluster_labels2, df_clean], axis = 1)

# Renaming the first column containing cluster labels to 'cluster' for clarity
df_2clust = df_2clust.rename(columns = {0: 'cluster'})

# Displaying the first few rows of the DataFrame df_2clust after renaming the column
df_2clust.head()

# Calculating the mean of selected columns (columns 1 to 6) grouped by the cluster labels (0 or
1)
df_2clust.iloc[:, 1:7].groupby(df_2clust.cluster).mean()

# Concatenating the original 'Univ' column, cluster labels, and cleaned DataFrame along the
columns axis
df_3clust = pd.concat([df.Univ, cluster_labels2, df_clean], axis = 1)

# Renaming the first column containing cluster labels to 'cluster' for clarity
df_3clust = df_3clust.rename(columns = {0: 'cluster'})

# Saving the DataFrame df_3clust to a CSV file named 'University.csv' with UTF-8 encoding
df_3clust.to_csv('University.csv', encoding = 'utf-8')
```

```
# Getting the current working directory
import os
os.getcwd()

# Save final data into Database
user = 'user1' # user name
pw = 'user1' # password
db = 'univ_db' # database name
engine = create_engine(f"mysql+pymysql://{user}:{pw}@localhost/{db}")
df_3clust.to_sql('final', con = engine, if_exists = 'replace', chunksize = 1000, index = False)

# End of Hierarchical Clustering
```