**ADS\_Phase5 PROJECT SUBMISSION**

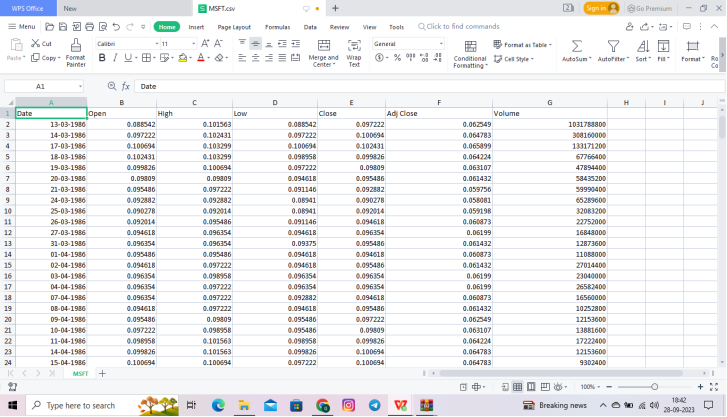
**PROBLEM STATEMENT:**

The problem is to build a predictive model that forecasts stock prices based on historical market data. The goal is to create a tool that assists investors in making well-informed decisions and optimizing their investment strategies. This project involves data collection, design thinking data preprocessing, feature engineering, model selection, training, and evaluation.

**DESIGN THINKING:**

1. Data Collection
2. Data Preprocessing
3. Feature Engineering
4. Model Selection
5. Model Training
6. Evaluation

**Dataset Link:** [**https://www.kaggle.com/datasets/prasoonkottarathil/microsoft-lifetime-stocks-dataset**](https://www.kaggle.com/datasets/prasoonkottarathil/microsoft-lifetime-stocks-dataset%E2%80%AF)



**Data Collection :**

Regardless of the field of study or preference for defining data (quantitative, qualitative), accurate data collection is essential to maintaining the integrity of research. Both the selection of appropriate data collection instruments (existing, modified, or newly developed) and clearly delineated instructions for their correct use reduce the likelihood of errors occurring.

**Data Preprocessing :**

Data Preprocessing can be defined as a process of converting raw data into a format that is understandable and usable for further analysis. It is an important step in the Data Preparation stage. It ensures that the outcome of the analysis is accurate, complete, and consistent. The main objective of Data Understanding is to gather general insights about the input dataset that will help to perform further steps to preprocess data. Let’s review two of the most common ways to understand input datasets

**Feature Engineering :**

Feature engineering refers to manipulation — addition, deletion, combination, mutation — of your data set to improve machine learning model training, leading to better performance and greater accuracy. Effective feature engineering is based on sound knowledge of the business problem and the available data sources.

**Model Selection :**

Variable selection is the process of selecting the best subset of predictors for a given problem and predictive model, while model selection is done to select one specific model from the list of available predictive models for a given business problem.

**Model Training :**

Model training is at the heart of the data science development lifecycle where the data science team works to fit the best weights and biases to an algorithm to minimize the loss function over prediction range. Loss functions define how to optimize the ML algorithms. A data science team may use different types of loss functions depending on the project objectives, the type of data used and the type of algorithm.

**Evaluation :**

Model evaluation is the process of using different evaluation metrics to understand a machine learning model’s performance, as well as its strengths and weaknesses. Model evaluation is important to assess the efficacy of a model during initial research phases, and it also plays a role in model monitoring.

**Feature engineering**

Feature engineering refers to manipulation — addition, deletion, combination, mutation — of your data set to improve machine learning model training, leading to better performance and greater accuracy.

**Model training**

Model training is the phase in the data science development lifecycle where practitioners try to fit the best combination of weights and bias to a machine learning algorithm to minimize a loss function over the prediction range

**PROGRAM:**

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

import numpy as np

# Create a sample dataset (replace with your own data)

data = np.random.rand(100, 5)

# Standardize the data

scaler = StandardScaler()

data\_std = scaler.fit\_transform(data)

# Initialize PCA with the desired number of components (k)

pca = PCA(n\_components=2)

# Fit and transform the data

data\_pca = pca.fit\_transform(data\_std)

# Explained variance ratio: the proportion of variance explained by each principal component

explained\_variance\_ratio = pca.explained\_variance\_ratio\_

# Print the results

print("Original Data Shape:", data.shape)

print("Reduced Data Shape:", data\_pca.shape)

print("Explained Variance Ratio:", explained\_variance\_ratio)

OUTPUT:

Original Data Shape: (100, 5)  
Reduced Data Shape: (100, 2)  
Explained Variance Ratio: [0.25962537 0.2206742 ]

**PROGRAM:**

import numpy as np

from sklearn.decomposition import PCA

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

# Load a sample dataset (Iris dataset)

data = load\_iris()

X = data.data

y = data.target

# Standardize the features (optional but recommended)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Create a PCA instance and specify the number of components (dimensions) you want to retain

n\_components = 2 # You can change this based on your needs

pca = PCA(n\_components=n\_components)

# Fit and transform the data to the reduced feature space

X\_pca = pca.fit\_transform(X\_scaled)

# Percentage of variance explained by each component

explained\_variance = pca.explained\_variance\_ratio\_

eigenvectors = pca.components\_

# Plot the explained variance for each component

plt.bar(range(n\_components), explained\_variance, tick\_label=[f'PC{i+1}' for i in range(n\_components)])

plt.xlabel('Principal Component')

plt.ylabel('Explained Variance')

plt.title('Explained Variance of Principal Components')

plt.show()

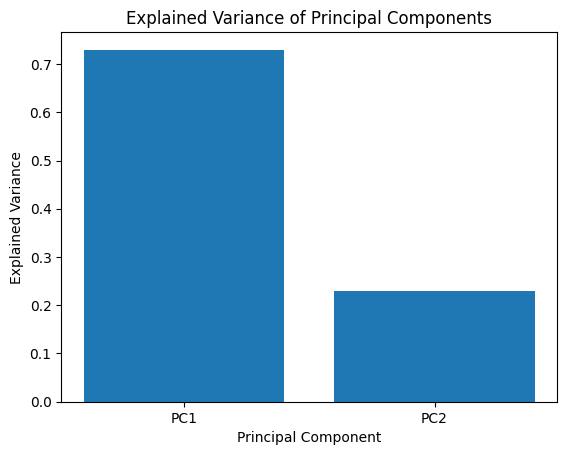
# Now X\_pca contains the data in the reduced feature space

print(f"Original data shape: {X\_scaled.shape}")

print(f"Reduced data shape: {X\_pca.shape}")

# You can use X\_pca for downstream machine learning tasks

**OUTPUT:**



Original data shape: (150, 4)  
Reduced data shape: (150, 2)

**PROGRAM:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_blobs

# Generate a synthetic dataset with 4 clusters

X, \_ = make\_blobs(n\_samples=30, centers=4, cluster\_std=1.0, random\_state=42)

# Define a divisive clustering function

def divisive\_clustering(data, num\_clusters, threshold):

if num\_clusters == 1:

return [data] # Return all data as one cluster

else:

# Perform divisive clustering recursively

# Split data into two subclusters

cluster\_1 = divisive\_clustering(data[: len(data) // 2], num\_clusters // 2, threshold)

cluster\_2 = divisive\_clustering(data[len(data) // 2 :], num\_clusters // 2, threshold)

return cluster\_1 + cluster\_2

# Perform divisive clustering with 4 clusters

num\_clusters = 4

clusters = divisive\_clustering(X, num\_clusters, threshold=0.5)

# Visualize the clusters

colors = ['r', 'g', 'b', 'y']

for i, cluster in enumerate(clusters):

cluster\_data = np.array(cluster)

plt.scatter(cluster\_data[:, 0], cluster\_data[:, 1], label=f'Cluster {i+1}', c=colors[i])

plt.title("Divisive Clustering")

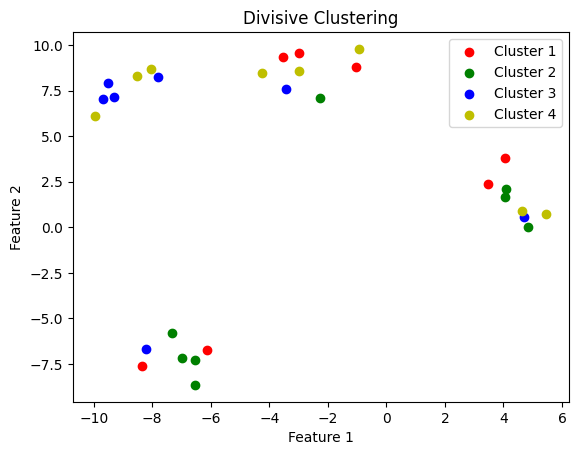
plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()

**OUTPUT:**



**PROGRAM:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_blobs

from sklearn.cluster import KMeans

# Generate a synthetic dataset with 4 clusters

X, \_ = make\_blobs(n\_samples=300, centers=5, cluster\_std=1.0, random\_state=42)

# Visualize the data

plt.scatter(X[:, 0], X[:, 1], s=50)

plt.title("Synthetic Data")

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.show()

# Perform K-means clustering with K=4 (number of clusters)

kmeans = KMeans(n\_clusters=5, random\_state=42)

kmeans.fit(X)

# Get the cluster assignments and centroids

cluster\_labels = kmeans.labels\_

print(cluster\_labels)

centroids = kmeans.cluster\_centers\_

print(centroids)

# Visualize the clusters and centroids

plt.scatter(X[:, 0], X[:, 1], c=cluster\_labels, s=50, cmap='viridis')

plt.scatter(centroids[:, 0], centroids[:, 1], c='red', marker='x', s=200, label='Centroids')

plt.title("K-means Clustering")

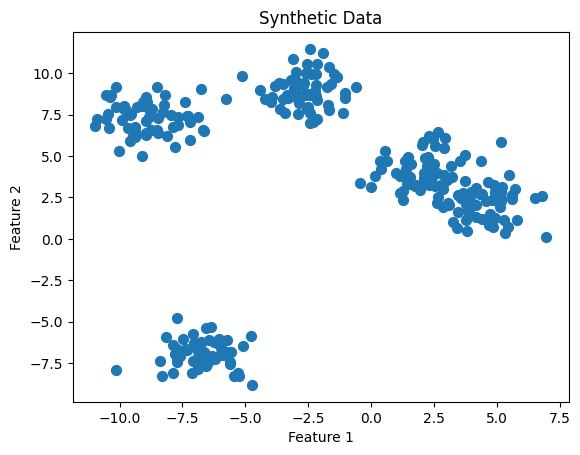
plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend()

plt.show()

**OUTPUT:**



**PROGRAM:**

import numpy as np

from sklearn.linear\_model import Lasso

from sklearn.datasets import make\_regression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

# Generate some random data for regression

X, y = make\_regression(n\_samples=100, n\_features=5, noise=0.1, random\_state=42)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a Lasso regression model with L1 regularization

alpha = 0.01 # Alpha is the regularization strength, higher values mean stronger regularization

lasso\_model = Lasso(alpha=alpha)

# Fit the model to the training data

lasso\_model.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = lasso\_model.predict(X\_test)

# Calculate the Mean Squared Error (MSE) to evaluate the model's performance

mse = mean\_squared\_error(y\_test, y\_pred)

print(f"Mean Squared Error (MSE): {mse}")

r\_square = r2\_score(y\_test, y\_pred)

print(r\_square)

# Print the learned coefficients after L1 regularization

print("Learned Coefficients:", lasso\_model.coef\_)

import matplotlib.pyplot as plt

# Visualize predicted vs. actual values

plt.scatter(y\_test, y\_pred)

plt.xlabel("Actual Values")

plt.ylabel("Predicted Values")

plt.title("Actual vs. Predicted Values")

plt.show()

**OUTPUT:**

[3 4 3 0 3 1 4 2 1 4 0 3 0 3 4 3 0 0 4 3 3 0 3 4 4 3 2 4 0 2 3 4 4 1 2 0 0  
 4 1 0 3 0 2 1 4 1 3 2 4 2 1 4 2 1 3 1 0 3 1 2 3 1 4 4 2 0 2 2 2 2 0 2 1 4  
 1 3 1 1 4 3 2 3 3 4 3 2 1 0 0 4 4 4 3 4 3 3 4 1 0 4 1 1 0 2 3 3 3 0 3 0 1  
 4 0 0 3 3 3 2 1 1 1 2 1 1 3 2 4 4 1 3 0 1 0 3 3 3 3 0 2 4 3 1 3 0 4 3 4 2  
 0 2 0 2 2 0 3 4 2 1 3 0 1 1 4 0 3 1 4 0 0 4 2 3 0 0 3 0 1 0 2 2 1 1 0 2 2  
 4 3 1 2 0 4 2 3 2 2 1 1 2 4 0 1 2 0 0 1 3 3 2 2 3 2 1 4 2 3 0 1 1 2 4 1 1  
 4 1 4 0 1 4 1 4 1 4 0 2 4 0 0 0 4 3 1 0 2 4 2 1 4 4 2 2 0 4 1 2 1 4 2 4 2  
 4 2 3 4 0 0 3 4 0 2 0 1 2 2 3 3 1 0 4 4 4 1 3 3 0 0 2 2 2 4 0 1 0 2 1 3 1  
 2 3 4 3]  
[[ 4.5017291 2.06024722]  
 [-8.89889839 7.28107273]  
 [-6.68626728 -6.96943511]  
 [-2.69693158 8.92945971]  
 [ 2.07811358 4.13347063]]

**PROGRAM:**

import numpy as np

from sklearn.linear\_model import Ridge

from sklearn.datasets import make\_regression

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

# Generate some random data for regression

X, y = make\_regression(n\_samples=100, n\_features=5, noise=0.1, random\_state=42)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a Ridge regression model with L2 regularization

alpha = 1.0 # Alpha is the regularization strength, higher values mean stronger regularization

ridge\_model = Ridge(alpha=alpha)

# Fit the model to the training data

ridge\_model.fit(X\_train, y\_train)

# Make predictions on the test data

y\_pred = ridge\_model.predict(X\_test)

# Calculate the Mean Squared Error (MSE) to evaluate the model's performance

mse = mean\_squared\_error(y\_test, y\_pred)

print(f"Mean Squared Error (MSE): {mse}")

# Print the learned coefficients after Ridge regularization

print("Learned Coefficients:", ridge\_model.coef\_)

**OUTPUT:**

Mean Squared Error (MSE): 4.114050771972589  
Learned Coefficients: [59.87954432 97.15091098 63.24364738 56.31999433 35.34591136]

**Thank you** .