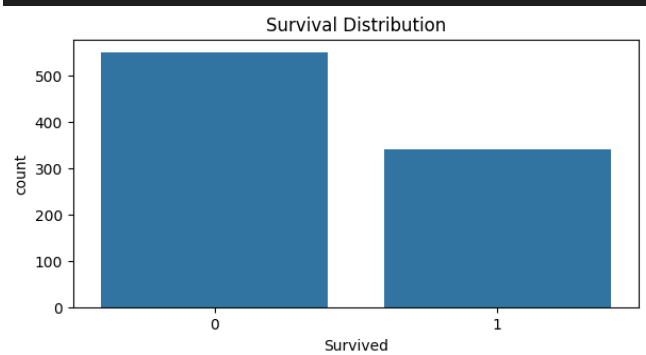
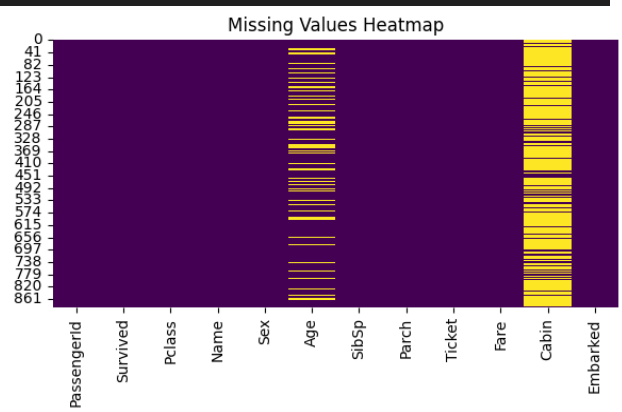
Assignment-01
Assignment Name: Problem solving on Classification

Task 01
Import a classification dataset (e.g., Iris, Titanic, or Breast Cancer). Perform an exploratory data analysis (EDA) to understand the features, target classes, and data distribution. Visualize key relationships using scatter plots, bar charts, or pair plots.

```
# Load Titanic dataset
titanic_data = pd.read_csv('titanic.csv')
# Display dataset information
print("\nTatanic dataset information")
print(titanic_data.info())

# Display first few rows
print("\n\nFirst few rows of the dataset")
print(titanic_data.head())

# Summary statistics
print("\n\nSummary statistics of the dataset")
print(titanic_data.describe())

# Survival distribution
fig, ax = plt.subplots(1, 2, figsize=(12, 4))

# Survival distribution
sns.countplot(data=titanic_data, x='Survived', ax=ax[0])
ax[0].set_title('Survival Distribution')

# Missing values heatmap
sns.heatmap(titanic_data.isnull(), cbar=False, cmap='viridis', ax=ax[1])
ax[1].set_title('Missing Values Heatmap')
plt.tight_layout()
plt.show()
```

```
Tatanic dataset information
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   PassengerId   891 non-null    int64
 1   Survived      891 non-null    int64
 2   Pclass        891 non-null    int64
 3   Name          891 non-null    object
 4   Sex           891 non-null    object
 5   Age           714 non-null    float64
 6   SibSp         891 non-null    int64
 7   Parch         891 non-null    int64
 8   Ticket        891 non-null    object
 9   Fare          891 non-null    float64
 10  Cabin         204 non-null    object
 11  Embarked      889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None


First few rows of the dataset
...
25%       0.000000     7.910400
50%       0.000000    14.454200
75%       0.000000    31.000000
max       6.000000   512.329200
```





Task 2:

Handling Missing Values in a Classification Dataset: Use a dataset with missing values (e.g., Titanic dataset). Demonstrate different methods to handle them, such as imputation (mean, mode) or deletion. Compare the impact of preprocessing on model performance.

```
from sklearn.impute import SimpleImputer

# Display missing values
print('Missing Values:')
print(titanic_data.isnull().sum())

# Impute missing 'Age' with mean
imputer = SimpleImputer(strategy='mean')
titanic_data['Age'] = imputer.fit_transform(titanic_data[['Age']])

# Impute missing 'Cabin' with mode (most frequent)
imputer = SimpleImputer(strategy='most_frequent')
titanic_data['Cabin'] = imputer.fit_transform(titanic_data[['Cabin']]).ravel()
```

```
# Impute missing 'Embarked' with mode (most frequent)
titanic_data['Embarked'] = imputer.fit_transform(titanic_data[['Embarked']]).ravel()
print('\nMissing Values After Handling:')
print(titanic_data.isnull().sum())
```

```
Missing Values:
PassengerId        0
Survived           0
Pclass             0
Name               0
Sex                0
Age              177
SibSp              0
Parch              0
Ticket             0
Fare               0
Cabin            687
Embarked           2
dtype: int64

Missing Values After Handling:
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age              0
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin            0
Embarked         0
dtype: int64
```

Task 3:

Scaling and Normalization: Apply standardization (z-score) or normalization (Min-Max scaling) to numeric features of a dataset (e.g., Wine dataset). Train a classification model before and after scaling, and evaluate the effect on accuracy.

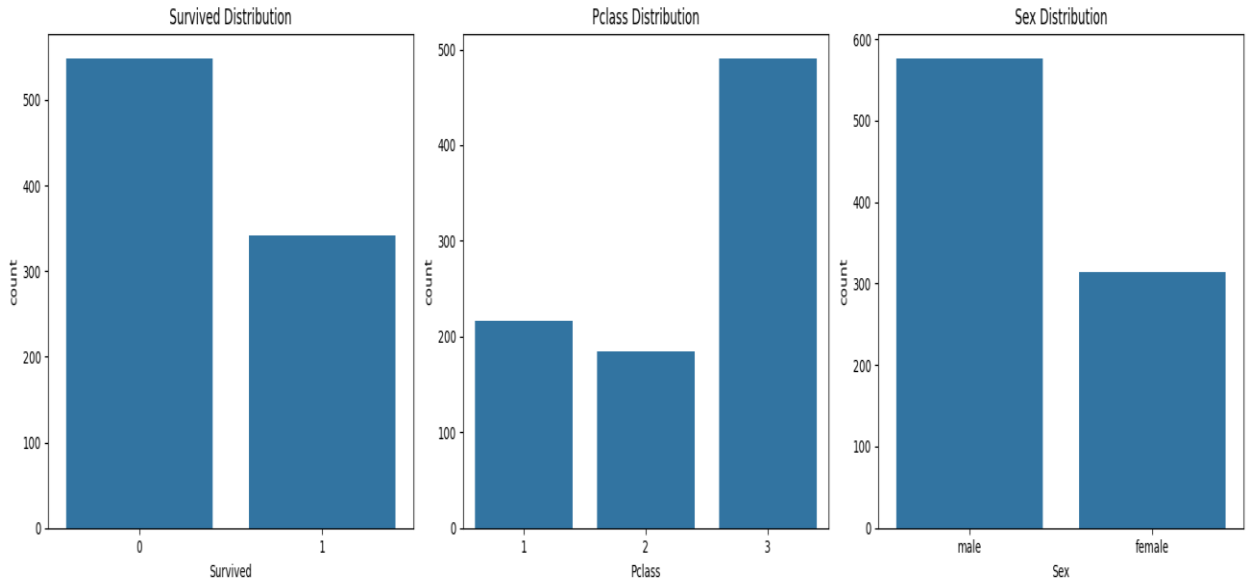
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Select numerical columns for scaling
numerical_cols = ['Age', 'Fare']

# Standardization (Z-score)
scaler = StandardScaler()
titanic_data_standardized = titanic_data.copy()
titanic_data_standardized[numerical_cols] = scaler.fit_transform(titanic_data[numerical_cols])

# Normalization (Min-Max Scaling)
min_max_scaler = MinMaxScaler()
titanic_data_normalized = titanic_data.copy()
titanic_data_normalized[numerical_cols] = min_max_scaler.fit_transform(titanic_data[numerical_cols])

# Display examples
print("Standardized Data (First 5 Rows):")
print(titanic_data_standardized[numerical_cols].head())

print("Normalized Data (First 5 Rows):")
```

print(titanic_data_normalized[numerical_cols].head())

```
Standardized Data (First 5 Rows):
         Age       Fare
0 -0.592481 -0.502445
1  0.638789  0.786845
2 -0.284663 -0.488854
3  0.407926  0.420730
4  0.407926 -0.486337
Normalized Data (First 5 Rows):
         Age       Fare
0  0.271174  0.014151
1  0.472229  0.139136
2  0.321438  0.015469
3  0.434531  0.103644
4  0.434531  0.015713
```

Task 4:
Handling Imbalanced Datasets: Import an imbalanced dataset (e.g., Credit Card Fraud Detection). Apply techniques such as oversampling (SMOTE), undersampling. Train a classifier and evaluate its performance using metrics like F1-score.

```
# Check balance of the target variable
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Plot for 'Survived'
sns.countplot(data=titanic_data, x='Survived', ax=axes[0])
axes[0].set_title('Survived Distribution')

# Plot for 'Pclass'
sns.countplot(data=titanic_data, x='Pclass', ax=axes[1])
axes[1].set_title('Pclass Distribution')

# Plot for 'Sex'
sns.countplot(data=titanic_data, x='Sex', ax=axes[2])
axes[2].set_title('Sex Distribution')

plt.tight_layout()
plt.show()
```



Task 5:
Encoding Categorical Variables: Import a dataset with categorical features (e.g., Titanic or Heart Disease). Apply label encoding and one-hot encoding, then train a classifier to compare the effect of encoding techniques on performance.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# One-hot encoding for all categorical columns
categorical_cols = titanic_data.select_dtypes(include=['object']).columns
titanic_data_temp = titanic_data.copy()

for col in categorical_cols:
    one_hot = pd.get_dummies(titanic_data_temp[col], prefix=col)
    titanic_data_temp = pd.concat([titanic_data_temp, one_hot], axis=1)
    titanic_data_temp = titanic_data_temp.drop(columns=[col])

print("Data After One-Hot Encoding (First 5 Rows):")
print(titanic_data_temp.head())

# Label encoding for all categorical columns in the main dataframe
label_encoder = LabelEncoder()
for col in categorical_cols:
    titanic_data[col] = label_encoder.fit_transform(titanic_data[col])

print("\nData After Label Encoding (First 5 Rows):")
print(titanic_data.head())
```

```
Data After One-Hot Encoding (First 5 Rows):
   PassengerId  Survived  Pclass   Age  SibSp  Parch     Fare  \
0            1         0       3  22.0      1      0   7.2500
1            2         1       1  38.0      1      0  71.2833
2            3         1       3  26.0      0      0   7.9250
3            4         1       1  35.0      1      0  53.1000
4            5         0       3  35.0      0      0   8.0500

   Name_Abbing, Mr. Anthony  Name_Abbott, Mr. Rossmore Edward  \
0                     False                             False
1                     False                             False
2                     False                             False
3                     False                             False
4                     False                             False

   Name_Abbott, Mrs. Stanton (Rosa Hunt)  ...  Cabin_F G73  Cabin_F2  \
0                                  False  ...        False     False
1                                  False  ...        False     False
2                                  False  ...        False     False
3                                  False  ...        False     False
4                                  False  ...        False     False

   Cabin_F33  Cabin_F38  Cabin_F4  Cabin_G6  Cabin_T  Embarked_C  Embarked_Q  \
0      False      False     False     False    False       False       False
1      False      False     False     False    False        True       False
...
1  71.2833     81         0
2   7.9250     47         2
3  53.1000     55         2
4   8.0500     47         2
```

Task 6:
Feature Transformation: Apply feature transformation techniques like PCA to a classification dataset. Train a classifier and compare the performance with and without feature transformation.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report
from sklearn.decomposition import PCA

# Define numerical_features_temp
numerical_features_temp = titanic_data_temp.select_dtypes(include=['float64', 'int64']).columns
# Exclude 'Survived' from the features
features_for_pca = numerical_features_temp.drop('Survived')

# Step 1: Apply PCA to reduce dimensionality
pca = PCA(n_components=6)
titanic_data_pca = titanic_data_temp.copy()
titanic_data_pca_temp = titanic_data_temp.copy()

# Apply PCA on selected features
pca_components = pca.fit_transform(titanic_data_temp[features_for_pca])
```

```
# Assign PCA components to new columns
pca_columns = [f'PC{i+1}' for i in range(pca.n_components_)]
titanic_data_pca_temp[pca_columns] = pca_components

# Optionally, drop the original features used for PCA
titanic_data_pca_temp = titanic_data_pca_temp.drop(columns=features_for_pca)

# Step 2: Train a classifier on the original dataset
X = titanic_data_temp.drop(columns=['Survived'])
y = titanic_data_temp['Survived']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

clf_original = RandomForestClassifier(random_state=42)
clf_original.fit(X_train, y_train)
y_pred_original = clf_original.predict(X_test)

# Step 3: Train a classifier on the PCA-transformed dataset
X_pca = titanic_data_pca_temp.drop(columns=['Survived'])
y_pca = titanic_data_pca_temp['Survived']
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y_pca, test_size=0.2,
random_state=42)

clf_pca = RandomForestClassifier(random_state=42)
clf_pca.fit(X_train_pca, y_train_pca)
y_pred_pca = clf_pca.predict(X_test_pca)

# Step 4: Compare the performance of both classifiers
print("Performance on Original Dataset:")
print("Accuracy:", accuracy_score(y_test, y_pred_original))
print("Classification Report:\n", classification_report(y_test, y_pred_original))

print("\nPerformance on PCA-Transformed Dataset:")
print("Accuracy:", accuracy_score(y_test_pca, y_pred_pca))
print("Classification Report:\n", classification_report(y_test_pca, y_pred_pca))
```

```
Performance on Original Dataset:
Accuracy: 0.8100558659217877
Classification Report:
              precision    recall  f1-score   support

           0       0.81      0.89      0.85       105
           1       0.81      0.70      0.75        74

    accuracy                           0.81       179
   macro avg       0.81      0.79      0.80       179
weighted avg       0.81      0.81      0.81       179


Performance on PCA-Transformed Dataset:
Accuracy: 0.8156424581005587
Classification Report:
              precision    recall  f1-score   support

           0       0.82      0.89      0.85       105
           1       0.82      0.72      0.76        74

    accuracy                           0.82       179
   macro avg       0.82      0.80      0.81       179
weighted avg       0.82      0.82      0.81       179
```

Task 7:
Train a Binary Classification Model: Use a dataset like Titanic or Heart Disease to train a binary classifier (e.g., Logistic Regression or Random Forest). Evaluate the model using accuracy, precision, recall, and F1-score.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Select features and target
titanic_data_encoded = titanic_data.copy()
X = titanic_data_encoded.drop(columns=['Survived'])
y = titanic_data_encoded['Survived']

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a random forest classifier
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)
# Predictions
y_pred = rf_model.predict(X_test)

# Evaluation
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred))
```

Random Forest Accuracy: 0.8435754189944135

Task 8:
Train a Multi-Class Classification Model: Use a dataset like Iris or MNIST to train a multi-class classification model (e.g., Decision Tree or SVM). Evaluate the model using metrics like accuracy and per-class F1-scores.

```
# For multi-class classification, use Iris dataset
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X_iris = iris.data
y_iris = iris.target

# Split data
X_train_iris, X_test_iris, y_train_iris, y_test_iris = train_test_split(X_iris, y_iris, test_size=0.2,
random_state=42)

# Train a Decision Tree
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train_iris, y_train_iris)

# Predictions
y_pred_iris = dt_model.predict(X_test_iris)

# Evaluation
print("Decision Tree Accuracy on Iris:", accuracy_score(y_test_iris, y_pred_iris))
```

Decision Tree Accuracy on Iris: 1.0

Task 9:
Evaluation Metrics and Cross-Validation:

a. Use a dataset to train a classifier and evaluate it with k-fold cross-validation. Report metrics such as:

i. Confusion Matrix
ii. Accuracy, Precision, Recall, F1-Score
iii. ROC-AUC Score

```
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import cross_val_score
# Confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Cross-validation
cv_scores = cross_val_score(rf_model, X, y, cv=5)
```

```
print("Cross-Validation Scores:", cv_scores)
print("Mean CV Score:", cv_scores.mean())
```

```
Confusion Matrix:
[[94 11]
 [17 57]]
Classification Report:
              precision    recall  f1-score   support

           0       0.85      0.90      0.87       105
           1       0.84      0.77      0.80        74

    accuracy                           0.84       179
   macro avg       0.84      0.83      0.84       179
weighted avg       0.84      0.84      0.84       179

Cross-Validation Scores: [0.83798883 0.82022472 0.87640449 0.83707865 0.87078652]
Mean CV Score: 0.8484966417676227
```

Task 10:

Comparing Models: Train multiple classifiers (e.g., Logistic Regression, Decision Tree, k-NN, SVM, Naïve Bayes and Random Forest) on the same dataset and compare their evaluation metrics.

```
# Train multiple classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

models = {
    'Random Forest': RandomForestClassifier(random_state=42),
    'Logistic Regression': LogisticRegression(),
    'SVM': SVC(),
    'Naive Bayes': GaussianNB(),
    'K-Nearest Neighbors': KNeighborsClassifier()
}

results = {}
for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    results[name] = accuracy_score(y_test, y_pred)

print("Model Comparison:")
print(results)
```

Model Comparison:
{'Random Forest': 0.8435754189944135, 'Logistic Regression': 0.770949720670391, 'SVM': 0.6815642458100558, 'Naive Bayes': 0.7988826815642458, 'K-Nearest Neighbors': 0.6703910614525139}

Assignment-02
Assignment Name: Problem solving on Clustering

Task 01:
Dataset Import and Initial Analysis

a. Load a clustering dataset (e.g., Iris, Mall Customers, or Wine dataset). Perform an exploratory data analysis (EDA) to understand the features and the distribution of data.

b. Visualize the data distribution using pair plots, histograms, and scatter plots.

c. Investigate any potential outliers or missing values.

```
# Load the dataset
iris = pd.read_csv('iris.csv')

# Display basic information about the dataset
print(iris.info())
print(iris.describe())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal.length  150 non-null    float64
 1   sepal.width   150 non-null    float64
 2   petal.length  150 non-null    float64
 3   petal.width   150 non-null    float64
 4   variety       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
None
       sepal.length  sepal.width  petal.length  petal.width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

 # Visualize data distributions

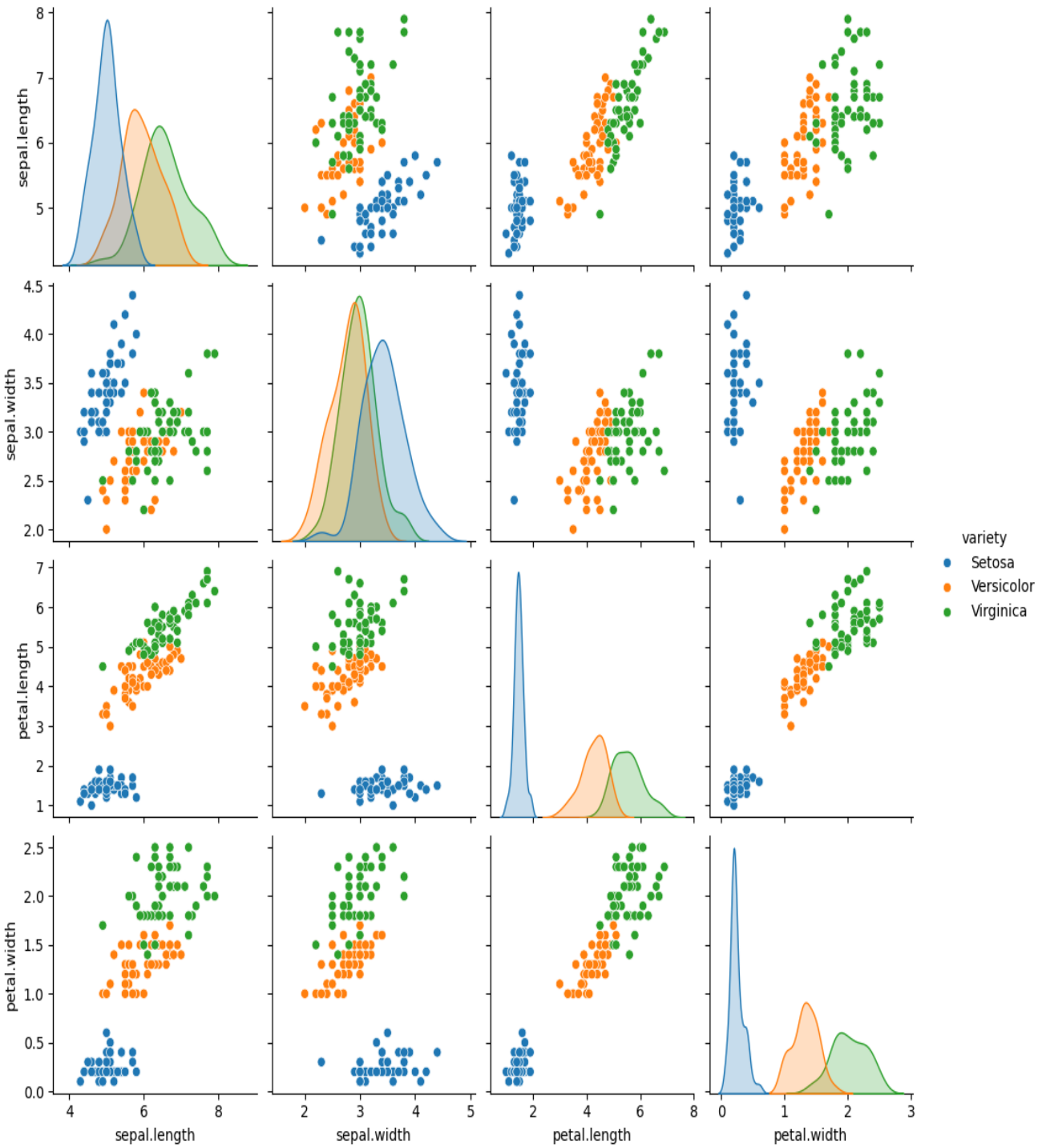
```
sns.pairplot(iris, hue='variety')
plt.show()
```

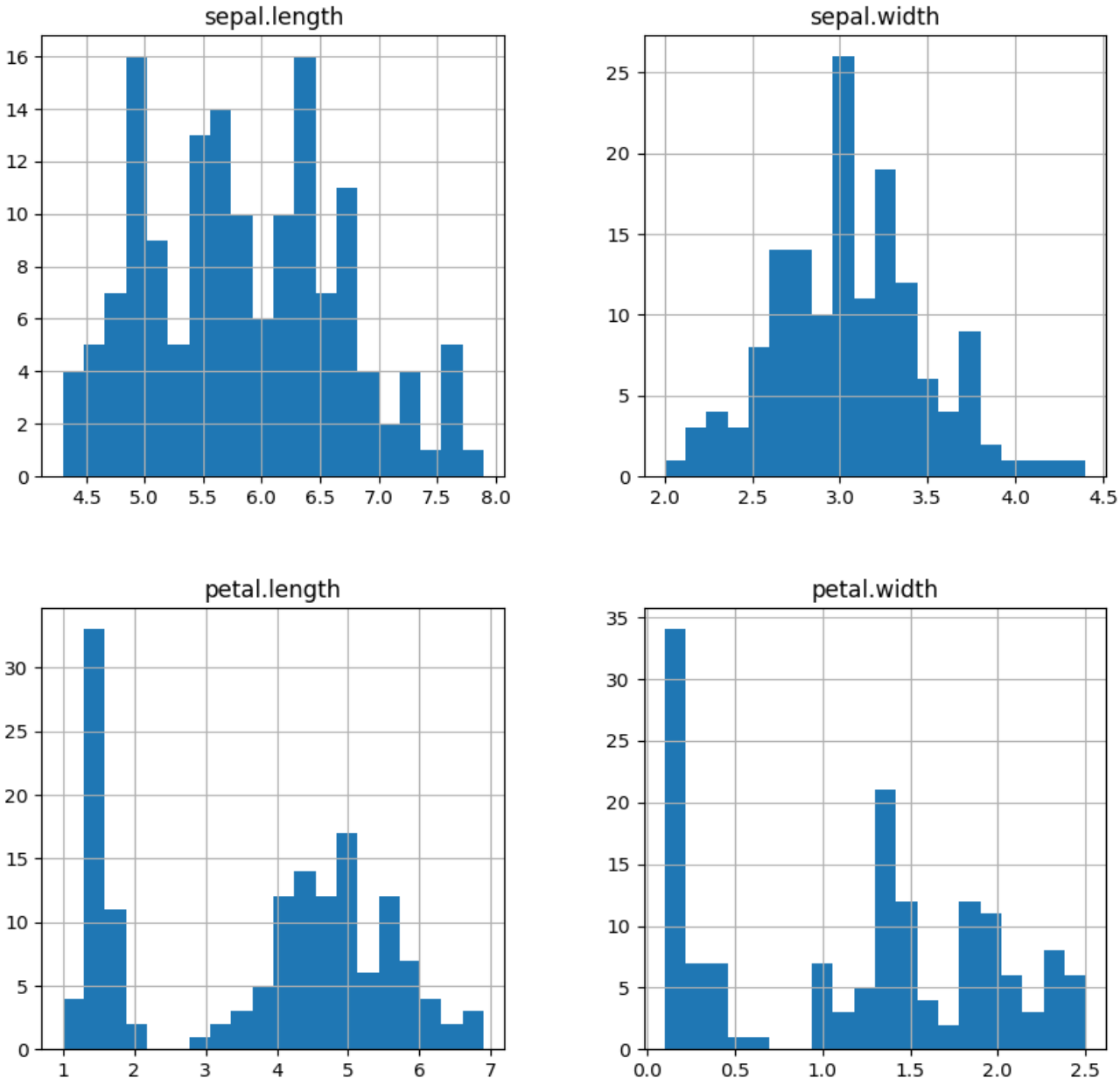
 # Visualize data distributions using histograms

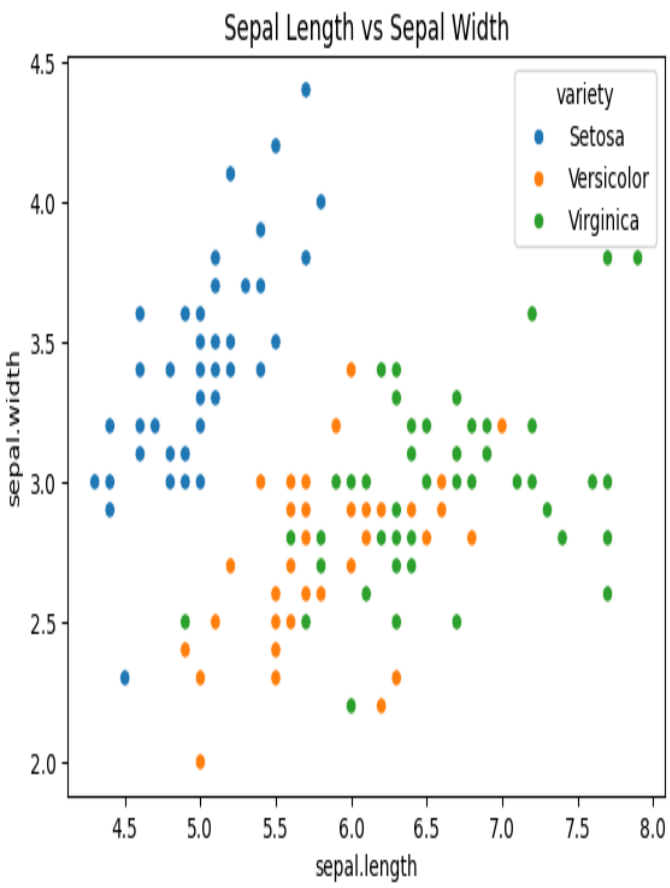
```
iris.hist(bins=20, figsize=(10, 10))
plt.show()
```

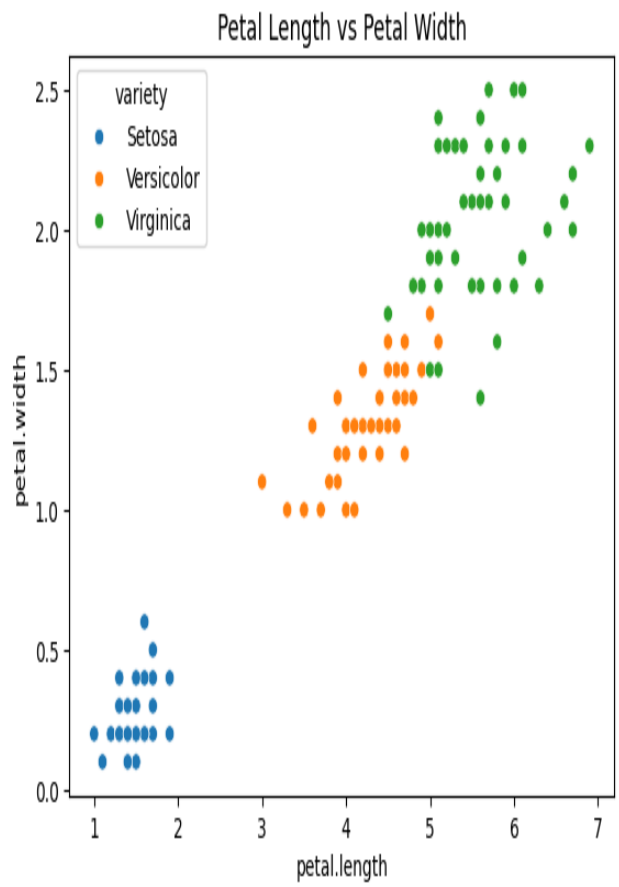
 # Visualize data distributions using scatter plots

```
sns.scatterplot(x='sepal.length', y='sepal.width', hue='variety', data=iris)
plt.title('Sepal Length vs Sepal Width')
plt.show()

sns.scatterplot(x='petal.length', y='petal.width', hue='variety', data=iris)
plt.title('Petal Length vs Petal Width')
plt.show()
```

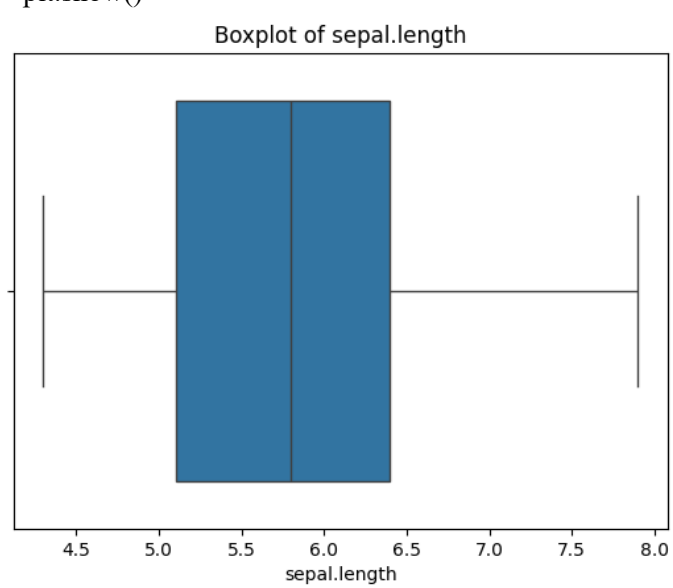


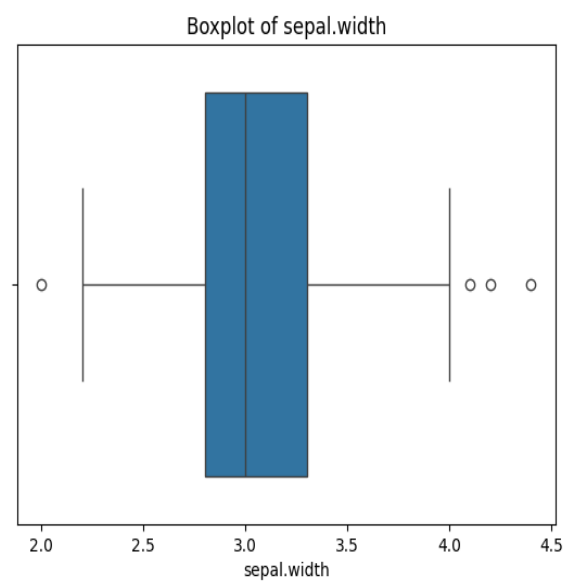

sepal.length
sepal.width
petal.length
petal.width

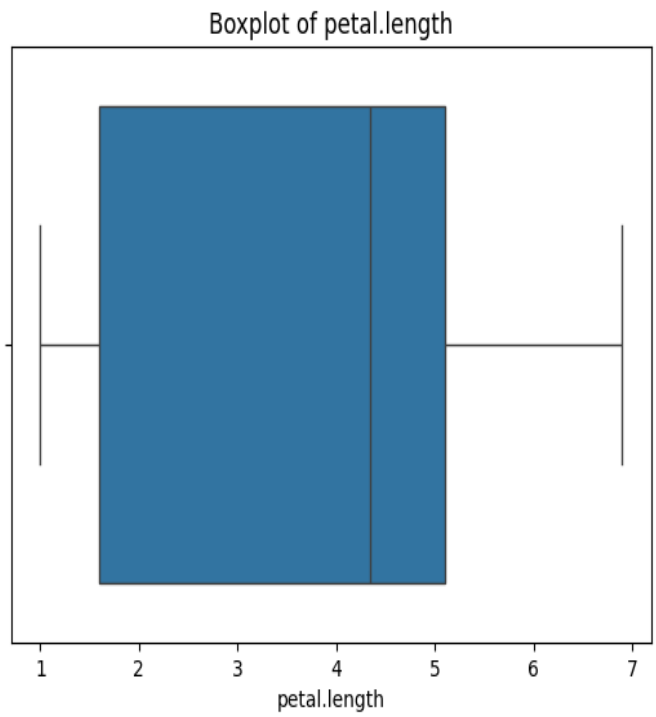


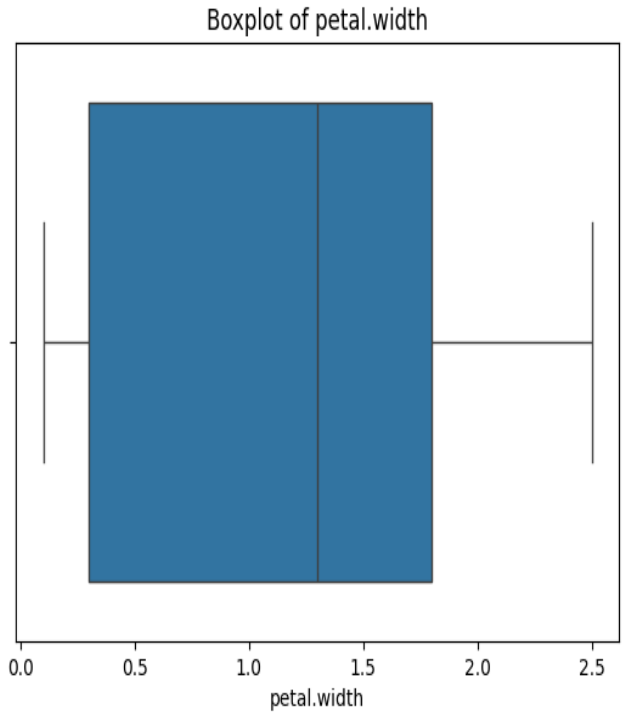



```
# Check for outliers using boxplots
for column in iris.select_dtypes(include=np.number).columns:
    sns.boxplot(x=iris[column])
    plt.title(f"Boxplot of {column}")
    plt.show()
```









Task 02:
Handling Missing Values: Import a dataset with missing values. Demonstrate how to handle missing data using imputation methods (e.g., mean, median, or mode imputation) and evaluate how this affects the clustering results.

iris.isnull().sum()

```
sepal.length    0
sepal.width     0
petal.length    0
petal.width     0
variety         0
dtype: int64
```

There is no missing value in this data set so no need to handle

Task 03:
Scaling and Normalization: Import a dataset with features that have different scales (e.g., Wine dataset). Perform feature scaling using techniques like Standardization (z-score) or Min-Max Normalization. Evaluate how this affects the clustering results

like k-means.

```
# Standardization (z-score)
scaler_standard = StandardScaler()
iris_standard = iris.copy()
iris_standard.iloc[:, :-1] = scaler_standard.fit_transform(iris_standard.iloc[:, :-1])

# Min-Max Normalization
scaler_minmax = MinMaxScaler()
iris_minmax = iris.copy()
iris_minmax.iloc[:, :-1] = scaler_minmax.fit_transform(iris_minmax.iloc[:, :-1])

# Apply KMeans clustering
kmeans_standard = KMeans(n_clusters=4)
iris_standard['cluster'] = kmeans_standard.fit_predict(iris_standard.iloc[:, :-1])

kmeans_minmax = KMeans(n_clusters=4)
iris_minmax['cluster'] = kmeans_minmax.fit_predict(iris_minmax.iloc[:, :-1])

# Plot clustering results for Standardization
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.scatterplot(x='petal.length', y='petal.width', hue='cluster', data=iris_standard, palette='viridis')
plt.title('KMeans Clustering (Standardization)')

# Plot clustering results for Min-Max Normalization
plt.subplot(1, 2, 2)
sns.scatterplot(x='petal.length', y='petal.width', hue='cluster', data=iris_minmax, palette='viridis')
plt.title('KMeans Clustering (Min-Max Normalization)')

plt.show()
```

Task 04:
Dealing with Categorical Data: Use a dataset with categorical features (e.g., Mall Customers dataset). Perform encoding techniques (e.g., one-hot encoding) to convert categorical variables into numerical values. Train a clustering model and compare the impact of these transformations on clustering results.

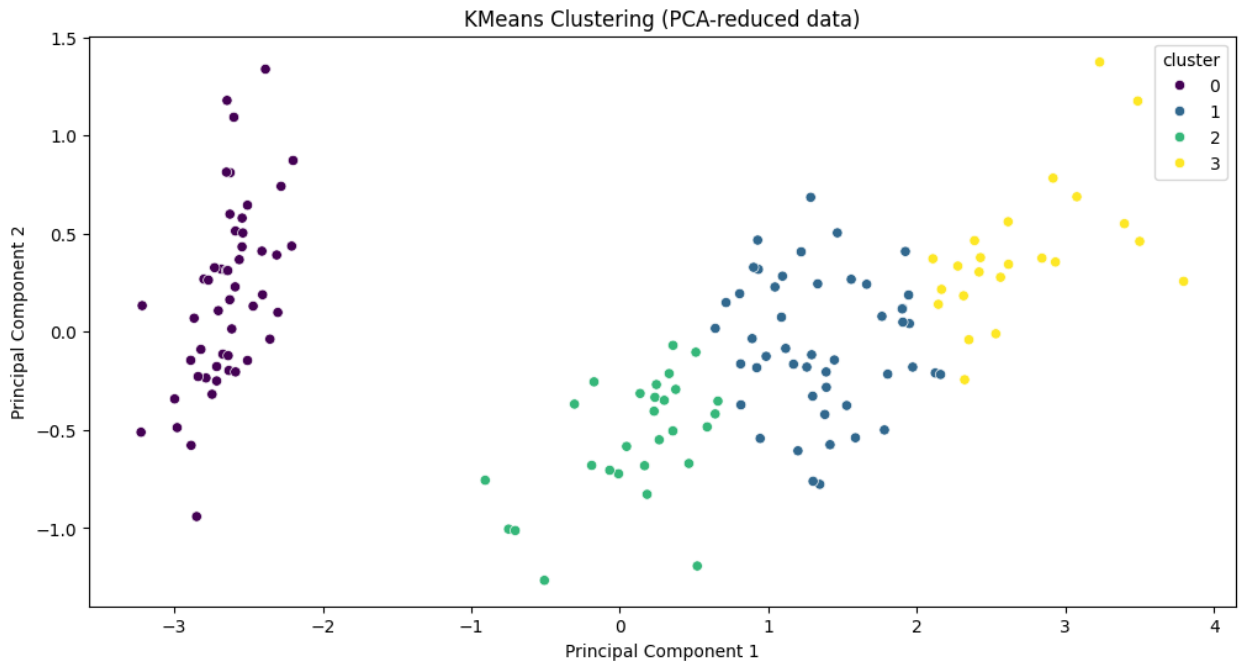
```
from sklearn.preprocessing import LabelEncoder

# Initialize the LabelEncoder
label_encoder = LabelEncoder()

# Fit and transform the 'variety' column
iris['variety'] = label_encoder.fit_transform(iris['variety'])

# Display the first few rows to verify the encoding
print(iris.head())
```

```
   sepal.length  sepal.width  petal.length  petal.width  variety
0           5.1          3.5           1.4          0.2        0
1           4.9          3.0           1.4          0.2        0
2           4.7          3.2           1.3          0.2        0
3           4.6          3.1           1.5          0.2        0
4           5.0          3.6           1.4          0.2        0
```

Task 05:
Feature Selection and Dimensionality Reduction: Apply Principal Component Analysis (PCA) to reduce dimensionality in a high-dimensional clustering dataset (e.g., Wine dataset). Visualize the clustering results in 2D/3D using the reduced features and compare the performance before and after dimensionality reduction.

```
# Apply PCA to reduce dimensionality to 2 components
pca = PCA(n_components=2)
iris_pca = iris.copy()
iris_pca_transformed = pca.fit_transform(iris_pca.iloc[:, :-1])
iris_pca = pd.DataFrame(iris_pca_transformed, columns=['PC1', 'PC2'])
iris_pca['variety'] = iris['variety']

# Apply KMeans clustering on the PCA-reduced data
kmeans_pca = KMeans(n_clusters=4)
iris_pca['cluster'] = kmeans_pca.fit_predict(iris_pca[['PC1', 'PC2']])

# Plot clustering results for PCA-reduced data
plt.figure(figsize=(12, 6))
sns.scatterplot(x='PC1', y='PC2', hue='cluster', data=iris_pca, palette='viridis')
plt.title('KMeans Clustering (PCA-reduced data)')
plt.xlabel('Principal Component 1')
```

```
plt.ylabel('Principal Component 2')
plt.show()

# Compare performance using silhouette score
silhouette_original = silhouette_score(iris.iloc[:, :-1], iris['variety'])
silhouette_pca = silhouette_score(iris_pca[['PC1', 'PC2']], iris_pca['cluster'])

print(f'Silhouette Score (Original Data): {silhouette_original}')
print(f'Silhouette Score (PCA-reduced Data): {silhouette_pca}')
```



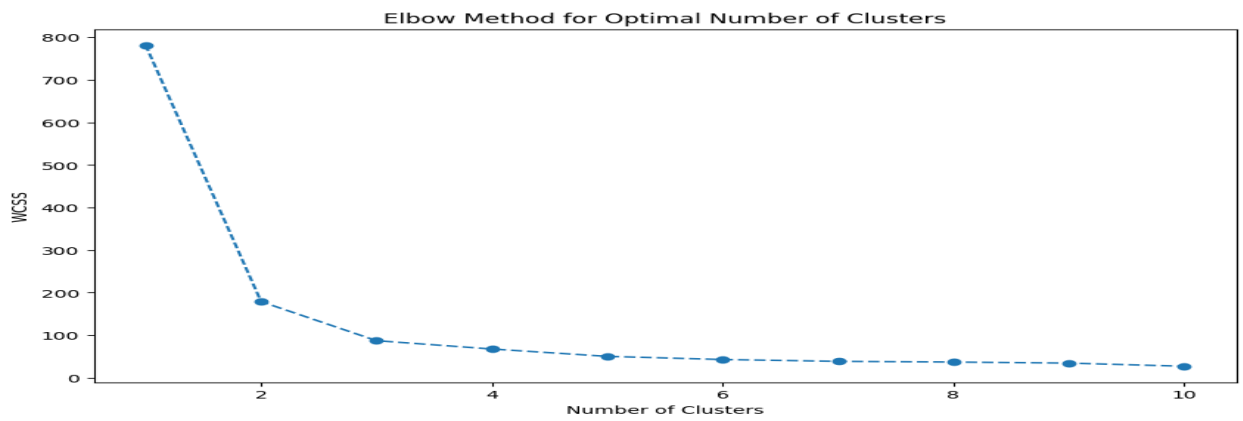

```
Silhouette Score (Original Data): 0.5034774406932966
Silhouette Score (PCA-reduced Data): 0.5591106064519759
```

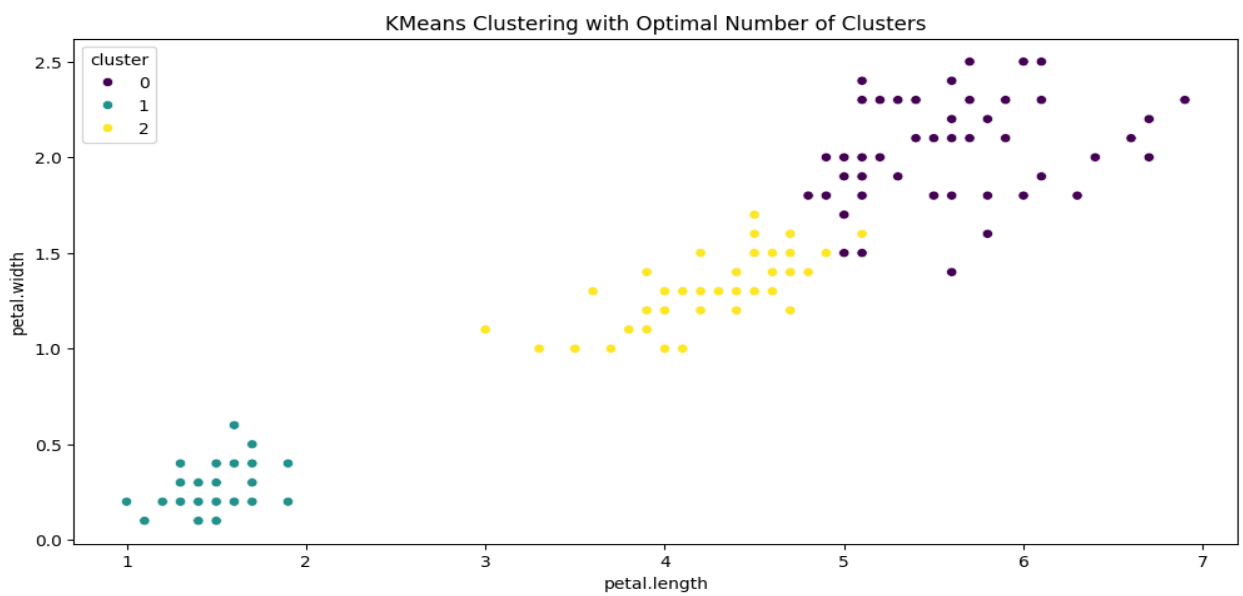
Task 06:
K-means Clustering:

a. Apply **K-means clustering** on a dataset (e.g., Iris or Mall Customers).
b. Use the **Elbow Method** to determine the optimal number of clusters.
c. Visualize the resulting clusters and analyze the cluster centroids.

```
# Apply K-means clustering with different numbers of clusters
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, random_state=42)
    kmeans.fit(iris.iloc[:, :-1])
    wcss.append(kmeans.inertia_)

# Plot the WCSS values to visualize the Elbow Method
plt.figure(figsize=(10, 6))
```

```
plt.plot(range(1, 11), wcss, marker='o', linestyle='--')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.show()

# From the Elbow Method plot, choose the optimal number of clusters (e.g., 3)
optimal_clusters = 3
kmeans_optimal = KMeans(n_clusters=optimal_clusters, random_state=42)
iris['cluster'] = kmeans_optimal.fit_predict(iris.iloc[:, :-1])

# Visualize the resulting clusters
plt.figure(figsize=(12, 6))
sns.scatterplot(x='petal.length', y='petal.width', hue='cluster', data=iris, palette='viridis')
plt.title('KMeans Clustering with Optimal Number of Clusters')
plt.show()

# Analyze the cluster centroids
centroids = kmeans_optimal.cluster_centers_
print(centroids)
```

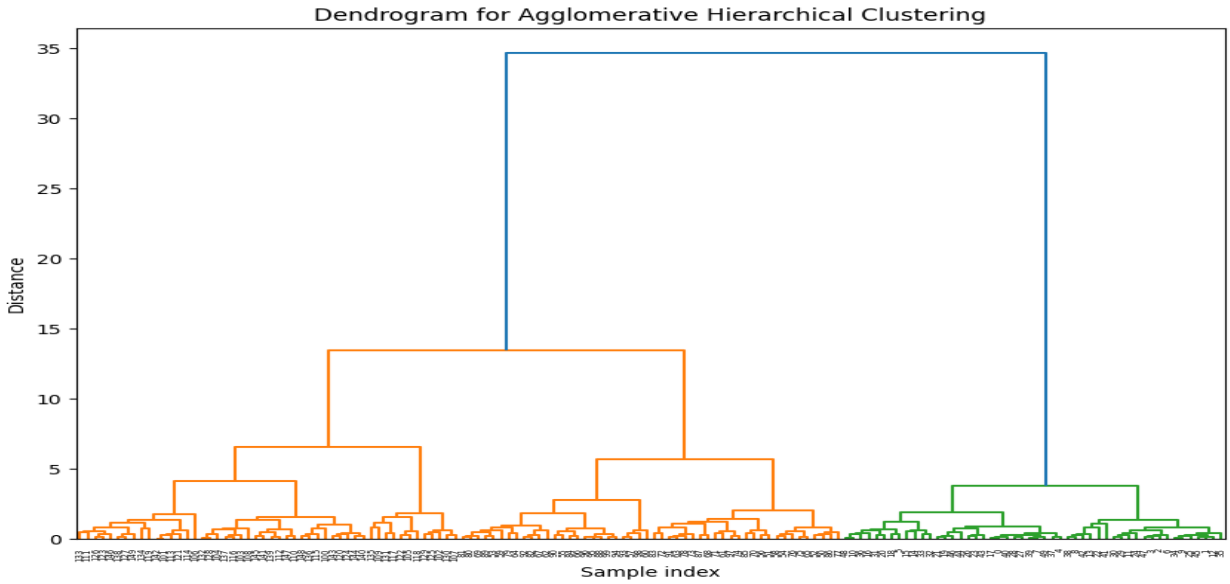


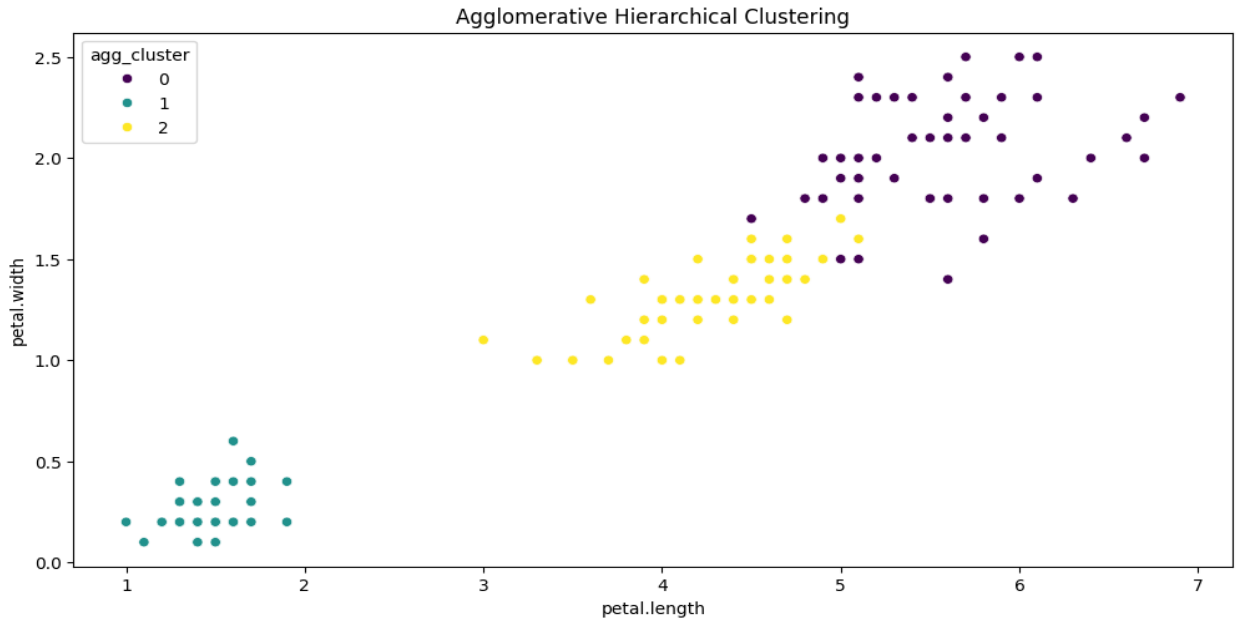


Task 07
Hierarchical Clustering:

a. Apply Agglomerative Hierarchical Clustering to a dataset (e.g., Wine or Iris).
b. Use a Dendrogram to visualize the hierarchy and determine the appropriate number of clusters.

```
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.cluster import AgglomerativeClustering

# Perform hierarchical/agglomerative clustering
linked = linkage(iris.iloc[:, :-1], method='ward')

# Plot the dendrogram
plt.figure(figsize=(10, 7))
dendrogram(linked, orientation='top', labels=iris.index, distance_sort='descending',
show_leaf_counts=True)
plt.title('Dendrogram for Agglomerative Hierarchical Clustering')
plt.xlabel('Sample index')
plt.ylabel('Distance')
plt.show()

# Determine the appropriate number of clusters from the dendrogram and apply Agglomerative Clustering
n_clusters = 3  # You can change this based on the dendrogram
agg_clustering = AgglomerativeClustering(n_clusters=n_clusters)
iris['agg_cluster'] = agg_clustering.fit_predict(iris.iloc[:, :-1])

# Visualize the resulting clusters
plt.figure(figsize=(12, 6))
sns.scatterplot(x='petal.length', y='petal.width', hue='agg_cluster', data=iris, palette='viridis')
plt.title('Agglomerative Hierarchical Clustering')
plt.show()
```





Task 08

Evaluation of Clustering Results: After applying clustering algorithms (e.g., K-means or Hierarchical Clustering), evaluate the clustering results using metrics like:

a. Silhouette Score: To measure how similar points within a cluster are, compared to points in other clusters.

b. Inertia (within-cluster sum of squares): To assess how compact the clusters are.

c. Adjusted Rand Index (ARI): To evaluate the similarity between the clusters and the true labels (if available).

d. Davies-Bouldin Index: To evaluate the average similarity ratio of each cluster with other clusters.

```
from sklearn.metrics import silhouette_score, adjusted_rand_score, davies_bouldin_score

# Silhouette Score
silhouette_kmeans = silhouette_score(iris.iloc[:, :-2], iris['cluster'])
silhouette_agg = silhouette_score(iris.iloc[:, :-2], iris['agg_cluster'])

# Adjusted Rand Index (ARI)
ari_kmeans = adjusted_rand_score(iris['variety'], iris['cluster'])
ari_agg = adjusted_rand_score(iris['variety'], iris['agg_cluster'])

# Davies-Bouldin Index
dbi_kmeans = davies_bouldin_score(iris.iloc[:, :-2], iris['cluster'])
dbi_agg = davies_bouldin_score(iris.iloc[:, :-2], iris['agg_cluster'])
```

```
# Inertia (within-cluster sum of squares) for KMeans
inertia_kmeans = kmeans_optimal.inertia_

print(f'Silhouette Score (KMeans): {silhouette_kmeans}')
print(f'Silhouette Score (Agglomerative Clustering): {silhouette_agg}')
print(f'Adjusted Rand Index (KMeans): {ari_kmeans}')
print(f'Adjusted Rand Index (Agglomerative Clustering): {ari_agg}')
print(f'Davies-Bouldin Index (KMeans): {dbi_kmeans}')
print(f'Davies-Bouldin Index (Agglomerative Clustering): {dbi_agg}')
print(f'Inertia (KMeans): {inertia_kmeans}')
```

```
Silhouette Score (KMeans): 0.5791983481276921
Silhouette Score (Agglomerative Clustering): 0.5782157637460459
Adjusted Rand Index (KMeans): 0.9602666666666667
Adjusted Rand Index (Agglomerative Clustering): 1.0
Davies-Bouldin Index (KMeans): 0.6443138947249966
Davies-Bouldin Index (Agglomerative Clustering): 0.6522266467240291
Inertia (KMeans): 87.2646
```

Assignemnet-03
Assignment Name: Problem solving on Regression

Task 01
Data Loading:

- Import a dataset of your choice (e.g., California Housing, Boston Housing, or custom CSV
- data).
- Identify the target and predictor variables.
- Check for missing values, data types, and outliers.

```
import pandas as pd
import numpy as np
# Load the dataset
data = pd.read_csv('housing.csv')

# Display basic info about the dataset
print(data.info())
print(data.describe())
print(data.head())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
None
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000     32.540000            1.000000      2.000000
25%     -121.800000     33.930000           18.000000   1447.750000
50%     -118.490000     34.260000           29.000000   2127.000000
...
1      2401.0      1138.0         8.3014            358500.0        NEAR BAY
2       496.0       177.0         7.2574            352100.0        NEAR BAY
3       558.0       219.0         5.6431            341300.0        NEAR BAY
4       565.0       259.0         3.8462            342200.0        NEAR BAY
```

Task 02
Handling Missing Values: Import a dataset with missing values. Demonstrate methods to handle them (e.g., mean/mode/median imputation, dropping rows/columns) and analyze how this affects the regression model's performance.

```
# Check for missing values
missing_values = data.isnull().sum()
print("Missing values before handling:\n", missing_values)
```

```
Missing values before handling:
 longitude                 0
latitude                  0
housing_median_age        0
total_rooms               0
total_bedrooms          207
population                0
households                0
median_income             0
median_house_value        0
ocean_proximity           0
dtype: int64
```

```
numeric_cols = data.select_dtypes(include=['float64', 'int64']).columns
# Handle missing values using mean imputation
data.loc[:, numeric_cols] = data[numeric_cols].fillna(data[numeric_cols].mean())

# Fill missing values for categorical fields with mode
data['ocean_proximity'].fillna(data['ocean_proximity'].mode()[0])

# Verify if missing values are handled
print("Missing values after handling:\n", data.isnull().sum())
```

```
Missing values after handling:
 longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms        0
population            0
households            0
median_income         0
median_house_value    0
ocean_proximity       0
dtype: int64
```

Task 03

Scaling and Normalization: Import a dataset with features on different scales. Perform normalization (e.g., Min-Max Scaling) and standardization (e.g., z-score). Compare the impact of these preprocessing steps on the performance of a linear regression model.

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler
# Min-Max Scaling
scaler = MinMaxScaler()
data_minmax_scaled = data.copy()
data_minmax_scaled[numeric_cols] = scaler.fit_transform(data[numeric_cols])

# Standard Scaling
standard_scaler = StandardScaler()
data_standard_scaled = data.copy()
data_standard_scaled[numeric_cols] = standard_scaler.fit_transform(data[numeric_cols])

print("Data after Min-Max Scaling:\n", data_minmax_scaled.head())
print("Data after Standard Scaling:\n", data_standard_scaled.head())
```

```
Data after Min-Max Scaling:
    longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0   0.211155  0.567481            0.784314     0.022331        0.019863
1   0.212151  0.565356            0.392157     0.180503        0.171477
2   0.210159  0.564293            1.000000     0.037260        0.029330
3   0.209163  0.564293            1.000000     0.032352        0.036313
4   0.209163  0.564293            1.000000     0.041330        0.043296

   population  households  median_income  median_house_value ocean_proximity
0    0.008941    0.020556       0.539668            0.902266        NEAR BAY
1    0.067210    0.186976       0.538027            0.708247        NEAR BAY
2    0.013818    0.028943       0.466028            0.695051        NEAR BAY
3    0.015555    0.035849       0.354699            0.672783        NEAR BAY
4    0.015752    0.042427       0.230776            0.674638        NEAR BAY
Data after Standard Scaling:
    longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0  -1.327835  1.052548            0.982143    -0.804819       -0.975228
1  -1.322844  1.043185           -0.607019     2.045890        1.355088
2  -1.332827  1.038503            1.856182    -0.535746       -0.829732
3  -1.337818  1.038503            1.856182    -0.624215       -0.722399
4  -1.337818  1.038503            1.856182    -0.462404       -0.615066

   population  households  median_income  median_house_value ocean_proximity
0   -0.974429   -0.977033       2.344766            2.129631        NEAR BAY
1    0.861439    1.669961       2.332238            1.314156        NEAR BAY
2   -0.820777   -0.843637       1.782699            1.258693        NEAR BAY
3   -0.766028   -0.733781       0.932968            1.165100        NEAR BAY
4   -0.759847   -0.629157      -0.012881            1.172900        NEAR BAY
```

Task 04

Categorical Data Encoding: Use a dataset with categorical features. Demonstrate how to apply label encoding and one hot encoding. Train a regression model and compare results using these encoding techniques.

```
# Identify categorical columns
categorical_cols = data.select_dtypes(include=['object']).columns

# Label Encoding
from sklearn.preprocessing import LabelEncoder

data_label_encoded = data.copy()
for col in categorical_cols:
    le = LabelEncoder()
    data_label_encoded[col] = le.fit_transform(data[col])

# One-Hot Encoding
data_one_hot_encoded = pd.get_dummies(data, columns=categorical_cols)

print("Data after Label Encoding:\n", data_label_encoded.head())
print("Data after One-Hot Encoding:\n", data_one_hot_encoded.head())
data = data_label_encoded
```

```
Data after Label Encoding:
    longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0   0.211155  0.567481            0.784314     0.022331        0.019863
1   0.212151  0.565356            0.392157     0.180503        0.171477
2   0.210159  0.564293            1.000000     0.037260        0.029330
3   0.209163  0.564293            1.000000     0.032352        0.036313
4   0.209163  0.564293            1.000000     0.041330        0.043296

   population  households  median_income  median_house_value  ocean_proximity
0    0.008941    0.020556       0.539668            0.902266                3
1    0.067210    0.186976       0.538027            0.708247                3
2    0.013818    0.028943       0.466028            0.695051                3
3    0.015555    0.035849       0.354699            0.672783                3
4    0.015752    0.042427       0.230776            0.674638                3
Data after One-Hot Encoding:
    longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
0   0.211155  0.567481            0.784314     0.022331        0.019863
1   0.212151  0.565356            0.392157     0.180503        0.171477
2   0.210159  0.564293            1.000000     0.037260        0.029330
3   0.209163  0.564293            1.000000     0.032352        0.036313
4   0.209163  0.564293            1.000000     0.041330        0.043296

   population  households  median_income  median_house_value  \
0    0.008941    0.020556       0.539668            0.902266
1    0.067210    0.186976       0.538027            0.708247
...
1                     True                       False
2                     True                       False
3                     True                       False
4                     True                       False
```

Task 05

Feature Selection using LASSO Regression: Use a given dataset to perform regression using LASSO (Least Absolute Shrinkage and Selection Operator). Identify the features selected by LASSO, train the regression model, and evaluate its performance using metrics like R^2, RMSE, and MAE.

```
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Define target and predictors
X = data.drop('median_house_value', axis=1)
y = data['median_house_value']

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply LASSO
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)

# Features selected
selected_features = X_train.columns[lasso.coef_ != 0]
print("Features selected by LASSO:\n", selected_features)

# Evaluate the model
y_pred = lasso.predict(X_test)
print("LASSO Regression Performance:")
print("R²:", r2_score(y_test, y_pred))
print("RMSE:", mean_squared_error(y_test, y_pred, squared=False))
print("MAE:", mean_absolute_error(y_test, y_pred))
```

```
Features selected by LASSO:
 Index([], dtype='object')
LASSO Regression Performance:
R²: -0.00021908714592466794
RMSE: 0.23605188314893 12
MAE: 0.18681748714453036
```

Task 06

Correlation-Based Feature Selection: Perform feature selection based on correlation analysis. Remove highly correlated features (e.g., correlation > 0.85) and train a regression model on the remaining features. Evaluate the model using R^2, RMSE, MAE, and compare its performance with a model trained on the original dataset.

```
# Compute correlation matrix
correlation_matrix = data.corr().abs()

# Find highly correlated features
upper_triangle = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape), k=1).astype(bool))
high_corr_features = [column for column in upper_triangle.columns if any(upper_triangle[column] >
0.85)]

# Drop highly correlated features
data_uncorrelated = data.drop(high_corr_features, axis=1)

# Train and evaluate a regression model
X = data_uncorrelated.drop('median_house_value', axis=1)
y = data_uncorrelated['median_house_value']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply LASSO
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)

# Evaluate the model
y_pred = lasso.predict(X_test)
print("Model Performance after Correlation-Based Feature Selection:")
print("R²:", r2_score(y_test, y_pred))
print("RMSE:", mean_squared_error(y_test, y_pred, squared=False))
print("MAE:", mean_absolute_error(y_test, y_pred))
```

```
Model Performance after Correlation-Based Feature Selection:
R²: -0.00021908714592466794
RMSE: 0.2360518831489312
MAE: 0.18681748714453036
```

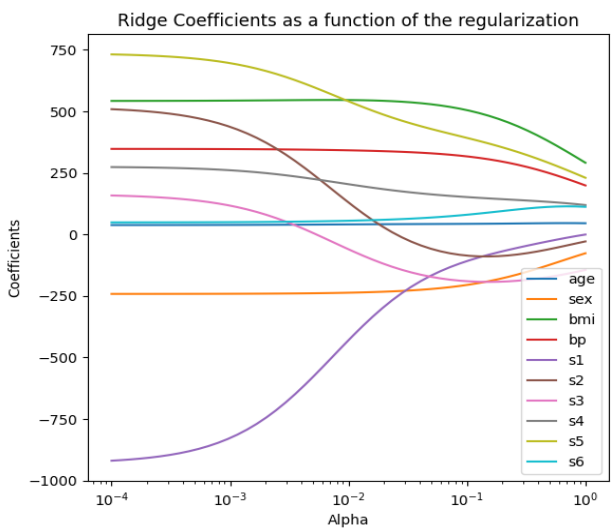
Task 07

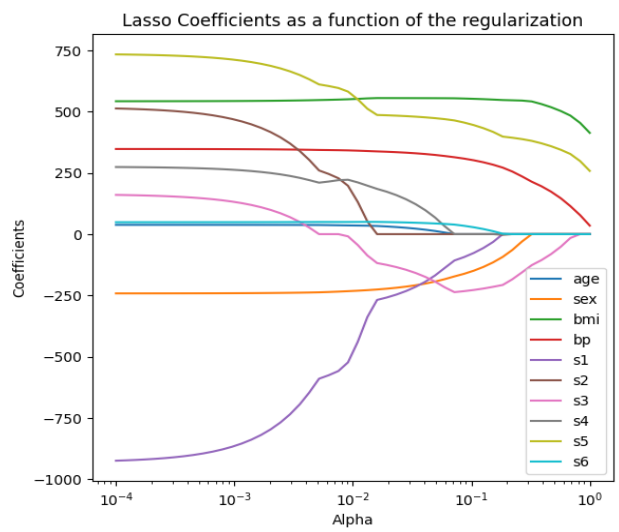
Regularization Techniques: Ridge and Lasso : Train Ridge and Lasso regression models on a dataset with multicollinearity (e.g., diabetes dataset). Compare the regularization effects by varying hyperparameters (α or λ) and observing their impact on model performance and feature coefficients.

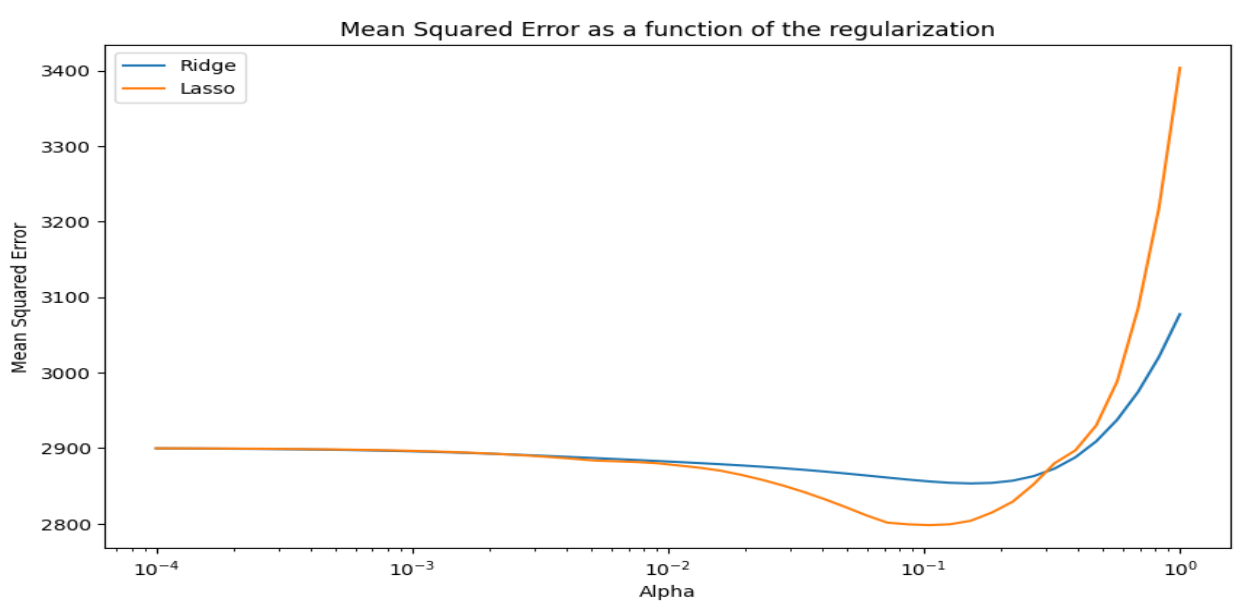
```
from sklearn.datasets import load_diabetes
from sklearn.linear_model import Ridge, Lasso
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np

import matplotlib.pyplot as plt
```

```
# Load the diabetes dataset
diabetes = load_diabetes()
X, y = diabetes.data, diabetes.target

# 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)

# Define a range of alpha values
alphas = np.logspace(-4, 0, 50)

# Initialize lists to store results
ridge_coefs = []
lasso_coefs = []
ridge_errors = []
lasso_errors = []

# Train Ridge and Lasso models for each alpha value
for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    lasso = Lasso(alpha=alpha)

    ridge.fit(X_train, y_train)
    lasso.fit(X_train, y_train)

    ridge_coefs.append(ridge.coef_)
    lasso_coefs.append(lasso.coef_)

    ridge_pred = ridge.predict(X_test)
    lasso_pred = lasso.predict(X_test)

    ridge_errors.append(mean_squared_error(y_test, ridge_pred))
    lasso_errors.append(mean_squared_error(y_test, lasso_pred))

# Plot the coefficients as a function of the regularization parameter
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.plot(alphas, ridge_coefs)
plt.xscale('log')
plt.xlabel('Alpha')
plt.ylabel('Coefficients')
plt.title('Ridge Coefficients as a function of the regularization')
plt.legend(diabetes.feature_names, loc='best')

plt.subplot(1, 2, 2)
plt.plot(alphas, lasso_coefs)
plt.xscale('log')
plt.xlabel('Alpha')
plt.ylabel('Coefficients')
plt.title('Lasso Coefficients as a function of the regularization')
```

```
plt.legend(diabetes.feature_names, loc='best')

plt.show()

# Plot the mean squared error as a function of the regularization parameter
plt.figure(figsize=(10, 6))
plt.plot(alphas, ridge_errors, label='Ridge')
plt.plot(alphas, lasso_errors, label='Lasso')
plt.xscale('log')
plt.xlabel('Alpha')
plt.ylabel('Mean Squared Error')
plt.title('Mean Squared Error as a function of the regularization')
plt.legend(loc='best')
plt.show()
```







Task 08

Model Evaluation with Cross-Validation: Use k-fold cross-validation to train and evaluate a regression model. Report metrics such as MSE, R^2, MAE, and RMSE for each fold. Compare performance with and without cross-validation.

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.metrics import make_scorer, mean_squared_error, r2_score, mean_absolute_error
import numpy as np

# Define the number of folds
kf = KFold(n_splits=5, shuffle=True, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Define custom scoring functions
mse_scorer = make_scorer(mean_squared_error)
r2_scorer = make_scorer(r2_score)
mae_scorer = make_scorer(mean_absolute_error)

# Perform cross-validation
mse_scores = cross_val_score(lasso, X, y, cv=kf, scoring=mse_scorer)
r2_scores = cross_val_score(lasso, X, y, cv=kf, scoring=r2_scorer)
mae_scores = cross_val_score(lasso, X, y, cv=kf, scoring=mae_scorer)
rmse_scores = np.sqrt(mse_scores)

# Report metrics for each fold
print("Cross-Validation Metrics:")
print("MSE for each fold:", mse_scores)
print("R² for each fold:", r2_scores)
print("MAE for each fold:", mae_scores)
print("RMSE for each fold:", rmse_scores)

# Compare performance with and without cross-validation
y_pred_train = lasso.predict(X_train)
y_pred_test = lasso.predict(X_test)

print("\nPerformance without Cross-Validation:")
print("Train MSE:", mean_squared_error(y_train, y_pred_train))
print("Test MSE:", mean_squared_error(y_test, y_pred_test))
print("Train R²:", r2_score(y_train, y_pred_train))
print("Test R²:", r2_score(y_test, y_pred_test))
print("Train MAE:", mean_absolute_error(y_train, y_pred_train))
print("Test MAE:", mean_absolute_error(y_test, y_pred_test))
print("Train RMSE:", np.sqrt(mean_squared_error(y_train, y_pred_train)))
print("Test RMSE:", np.sqrt(mean_squared_error(y_test, y_pred_test)))
```

```
Cross-Validation Metrics:
MSE for each fold: [3403.57572161 4091.01753878 3869.79768168 4061.82507058 4048.89044476]
R² for each fold: [0.35759188 0.34424209 0.28902288 0.39649174 0.27537681]
MAE for each fold: [49.73032754 53.81539502 51.52863472 55.78953202 54.40102956]
RMSE for each fold: [58.34017245 63.96106268 62.20769793 63.73244912 63.63089222]

Performance without Cross-Validation:
Train MSE: 3860.7549830123576
Test MSE: 3403.5757216070733
Train R²: 0.3646309911295581
Test R²: 0.3575918767219115
Train MAE: 52.95878032849505
Test MAE: 49.73032753662261
Train RMSE: 62.134973911737966
Test RMSE: 58.340172450954185
```

Task 09

Regression Model Comparison: Train multiple regression models (e.g., Linear Regression, Ridge, and Lasso) on the same dataset. Compare them based on evaluation metrics such as MAE, MSE, RMSE, and R^2. Discuss the scenarios where each model performs better.

```
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# Initialize the models
linear_reg = LinearRegression()
ridge_reg = Ridge(alpha=1.0)
lasso_reg = Lasso(alpha=0.1)

# Train the models
linear_reg.fit(X_train, y_train)
ridge_reg.fit(X_train, y_train)
lasso_reg.fit(X_train, y_train)

# Make predictions
y_pred_linear = linear_reg.predict(X_test)
y_pred_ridge = ridge_reg.predict(X_test)
y_pred_lasso = lasso_reg.predict(X_test)

# Evaluate the models
def evaluate_model(y_true, y_pred):
    mse = mean_squared_error(y_true, y_pred)
    mae = mean_absolute_error(y_true, y_pred)
    rmse = mean_squared_error(y_true, y_pred, squared=False)
    r2 = r2_score(y_true, y_pred)
    return mse, mae, rmse, r2

metrics_linear = evaluate_model(y_test, y_pred_linear)
metrics_ridge = evaluate_model(y_test, y_pred_ridge)
metrics_lasso = evaluate_model(y_test, y_pred_lasso)
```

```
# Print the evaluation metrics
print("Linear Regression Performance:")
print(f"MSE: {metrics_linear[0]}, MAE: {metrics_linear[1]}, RMSE: {metrics_linear[2]}, R²:
{metrics_linear[3]}")

print("\nRidge Regression Performance:")
print(f"MSE: {metrics_ridge[0]}, MAE: {metrics_ridge[1]}, RMSE: {metrics_ridge[2]}, R²:
{metrics_ridge[3]}")

print("\nLasso Regression Performance:")
print(f"MSE: {metrics_lasso[0]}, MAE: {metrics_lasso[1]}, RMSE: {metrics_lasso[2]}, R²:
{metrics_lasso[3]}")
```

```
Linear Regression Performance:
MSE: 2900.19362849348, MAE: 42.79409467959994, RMSE: 53.853445836765914, R²: 0.4526027629719197

Ridge Regression Performance:
MSE: 3077.41593882723, MAE: 46.13885766697452, RMSE: 55.47446204180109, R²: 0.41915292635986545

Lasso Regression Performance:
MSE: 2798.193485169719, MAE: 42.85442771664998, RMSE: 52.897953506442185, R²: 0.4718547867276227
```

Model Performance Comparison

Linear Regression Performance:
-Mean Squared Error (MSE): 2900.19362849348
- Mean Absolute Error (MAE): 42.79409467959994
- Root Mean Squared Error (RMSE): 53.853445836765914
- **R^2 Score: 0.4526027629719197

Ridge Regression Performance:
- Mean Squared Error (MSE): 3077.41593882723
- Mean Absolute Error (MAE): 46.13885766697452
- Root Mean Squared Error (RMSE): 55.47446204180109
- R^2 Score: 0.41915292635986545

Lasso Regression Performance:
- Mean Squared Error (MSE): 2798.193485169719
- Mean Absolute Error (MAE): 42.85442771664998
- Root Mean Squared Error (RMSE): 52.897953506442185
- R^2 Score: 0.4718547867276227

Discussion of Scenarios:

- Linear Regression:
- Performs better when the relationship between the independent and dependent variables is approximately linear and there is no multicollinearity in the data.
- Suitable for datasets with a large number of features where feature selection is not critical.

- Ridge Regression:
- Performs better in scenarios where multicollinearity is present among the features.
- Useful when you want to retain all features but reduce their impact by shrinking the coefficients.
- Suitable for datasets where overfitting is a concern and you want to regularize the model.

- Lasso Regression:
- Performs better when feature selection is important, as it can shrink some coefficients to zero, effectively performing variable selection.
- Suitable for datasets with many features, especially when some of them are not important.
- Useful when you want a simpler model that is easier to interpret.

Each model has its strengths and is suitable for different scenarios based on the nature of the data and the specific requirements of the analysis.