Develop a program to create histograms for all numerical features and analyze the distribution of each feature. Generate box plots for all numerical features and identify any outliers. Use California Housing dataset.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing
from scipy.stats import zscore

# Load the California Housing dataset
data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)

# Add target variable
df['MedHouseVal'] = data.target

df.info()

df.describe()

df.shape
```

```
# Create histograms for all numerical features
plt.figure(figsize=(12, 8))
df.hist(bins=30, figsize=(12, 8), edgecolor='black')
plt.suptitle('Histograms of Numerical Features', fontsize=16)
plt.tight_layout()
plt.show()

# Generate box plots for all numerical features
plt.figure(figsize=(12, 8))
for i, col in enumerate(df.columns, 1):
    plt.subplot(3, 4, i)
    sns.boxplot(y=df[col])
    plt.title(col)
plt.suptitle('Box Plots of Numerical Features', fontsize=16)
plt.tight_layout()
plt.show()

# Method 1 (Identfiying Outliers)
# Identify outliers using IQR method
outliers = {}
for col in df.columns:
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
```

```
    upper_bound = Q3 + 1.5 * IQR
    outliers[col] = df[(df[col] < lower_bound) | (df[col] >
upper_bound)][col].count()

# Display the number of outliers per feature
outliers_df = pd.DataFrame.from_dict(outliers, orient='index',
columns=['Outlier Count'])
print("Outliers detected in each feature:")
print(outliers_df)

# Method 2 (Identfiying Outliers)
# Identify outliers using Z-score method
z_scores = np.abs(zscore(df))
outliers = (z_scores > 3).sum(axis=0)  # Count outliers per feature

# Display the number of outliers per feature
outliers_df = pd.DataFrame(outliers, index=df.columns, columns=['Outlier
Count'])
print("Outliers detected in each feature (Z-score method):")
print(outliers_df)
```

Develop a program to Compute the correlation matrix to understand the relationships between pairs of features. Visualize the correlation matrix using a heatmap to know which variables have strong positive/negative correlations. Create a pair plot to visualize pairwise relationships between features. Use California Housing dataset.

```
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing
from scipy.stats import zscore

# Load the California Housing dataset
data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)

# Add target variable
df['MedHouseVal'] = data.target

df.info()

df.describe()

df.shape

# Compute the correlation matrix
correlation_matrix = df.corr()
print("\nCorrelation Matrix:")
print(correlation_matrix)

# Visualize the correlation matrix using a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f',
```

```
linewidths=0.5)
plt.title("Heatmap of Feature Correlations")
plt.show()

# Create a pair plot to visualize pairwise relationships
sns.pairplot(df.sample(500))  # Sampling to reduce computation time
plt.suptitle("Pairwise Relationships Between Features", y=1.02)
plt.show()
```

Develop a program to implement Principal Component Analysis (PCA) for reducing the dimensionality of the Iris dataset from 4 features to 2.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# Load the Iris dataset
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)

df['Species'] = data.target  # Add target labels
species_names = dict(enumerate(data.target_names))  # Map target values to species names
df['Species'] = df['Species'].map(species_names)
```

```
# Standardize the features
scaler = StandardScaler()
df_features = df.drop(columns=['Species'])  # Exclude target column
scaled_features = scaler.fit_transform(df_features)

# Apply PCA to reduce dimensions from 4 to 2
pca = PCA(n_components=2)
principal_components = pca.fit_transform(scaled_features)

# Create a DataFrame with the principal components
pca_df = pd.DataFrame(principal_components, columns=['PC1', 'PC2'])
pca_df['Species'] = df['Species']

pca_df

# Plot the PCA results
plt.figure(figsize=(8, 6))
sns.scatterplot(x='PC1', y='PC2', hue='Species', data=pca_df,
palette='viridis', s=100, edgecolor='black')
plt.title('PCA of Iris Dataset (2 Components)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title='Species')
plt.grid(True)
```

```
plt.show()


# Explained variance ratio

explained_variance = pca.explained_variance_ratio_

print("Explained variance by each principal component:", explained_variance)
```

For a given set of training data examples stored in a .CSV file, implement and demonstrate the Find-S algorithm to output a description of the set of all hypotheses consistent with the training examples.

```
import pandas as pd

import numpy as np


data = pd.read_csv('DS1.csv')


data.columns


print("Training Data:")

print(data)


def find_s_algorithm(data):

    """Implements the Find-S algorithm to find the most specific hypothesis."""

    attributes = data.columns[:-1]  # Excluding the target column

    target = data.columns[-1]
```

```
    # Initialize hypothesis with the first positive example
    for i in range(len(data)):
        if data.iloc[I, -1] == "Yes":  # Assuming 'Yes' is the positive class
            hypothesis = np.array(data.iloc[i][:-1])
            break

    # Iterate through the data and generalize the hypothesis
    for i in range(len(data)):
        if data.iloc[I, -1] == "Yes":
            for j in range(len(hypothesis)):
                if data.iloc[i, j] != hypothesis[j]:
                    hypothesis[j] = '?'  # Generalize differing attributes

    return hypothesis

hypothesis = find_s_algorithm(data)
print("\nMost Specific Hypothesis:")
print(hypothesis)
```

Develop a program to implement k-Nearest Neighbour algorithm to classify the randomly generated 100 values of x in the range of [0,1]. Perform the following based on dataset generated.

a. Label the first 50 points $\{x_1,\ldots\ldots,x_{50}\}$ as follows: if ($x_i \leq 0.5$), then $x_i \in$ Class1, else $x_i \in$ Class1

b. Classify the remaining points, $x_{51},\ldots\ldots,x_{100}$ using KNN. Perform this for k=1,2,3,4,5,20,30

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Step 1: Generate Random Data
np.random.seed(42)
x = np.random.rand(100).reshape(-1, 1)  # 100 values in the range [0,1]

# Step 2: Label the first 50 points
y = np.array([1 if xi <= 0.5 else 2 for xi in x[:50]])  # Class 1 for xi <= 0.5, Class 2 otherwise

y = np.concatenate([y, np.full(50, -1)])  # Unknown labels for points x51 to x100

# Step 3: Train KNN Classifier and Predict Labels for Remaining Points
accuracies = []

plt.figure(figsize=(10, 6))
plt.scatter(x[:50], y[:50], color='blue', label='Labeled Data (Class 1 & 2)')

for k in [1, 2, 3, 4, 5, 20, 30]:
    knn = KNeighborsClassifier(n_neighbors=k)
```

```
    knn.fit(x[:50], y[:50])
    y_pred = knn.predict(x[50:])

    # Store accuracy for comparison (true labels are known from the original condition)
    true_labels = np.array([1 if xi <= 0.5 else 2 for xi in x[50:]])
    accuracy = accuracy_score(true_labels, y_pred)
    accuracies.append((k, accuracy))

    # Visualization for each k value
    plt.scatter(x[50:], y_pred, label=f'k={k}')

plt.title('KNN Classification Results for Various k Values')
plt.xlabel('x values')
plt.ylabel('Class Labels')
plt.legend()
plt.show()

# Step 4: Display Accuracy Results
print("Accuracy Results for Different k values:")
for k, acc in accuracies:
    print(f'k = {k}: Accuracy = {acc:.2f}')
```

Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select appropriate data set for your experiment and draw graphs

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler

# Load dataset
boston = pd.read_csv('HousingData.csv')

# Extract RM (average number of rooms) and target (MEDV or last column)
X = boston[['RM']].values  # double brackets to keep it 2D
y = boston.iloc[:, -1].values  # get the last column as target

# Feature scaling
scaler_X = StandardScaler()
X_scaled = scaler_X.fit_transform(X)

# Add bias term
def add_bias(X):
    return np.c_[np.ones(X.shape[0]), X]

# Gaussian kernel
def gaussian_kernel(xi, x, tau):
    return np.exp(-np.sum((xi - x)**2, axis=1) / (2 * tau**2))
```

```
# Locally Weighted Regression
def locally_weighted_regression(X_train, y_train, tau, x_query):
    X_bias = add_bias(X_train)
    x_query_bias = add_bias(x_query)
    y_pred = []

    for xi in x_query_bias:
        weights = np.diag(gaussian_kernel(xi[1:], X_train, tau))
        theta = np.linalg.pinv(X_bias.T @ weights @ X_bias) @ (X_bias.T @ weights @ y_train)
        y_hat = xi @ theta
        y_pred.append(y_hat)

    return np.array(y_pred)

# Query points for smooth curve
x_query = np.linspace(X_scaled.min(), X_scaled.max(), 300).reshape(-1, 1)

# LWR Prediction
tau = 0.5  # bandwidth
y_pred = locally_weighted_regression(X_scaled, y, tau, x_query)

# Inverse transform query points for plotting
```

```
x_query_orig = scaler_X.inverse_transform(x_query)


# Plotting
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='blue', alpha=0.5, label='Data (RM vs MEDV)')
plt.plot(x_query_orig, y_pred, color='red', label=f'LWR fit (tau={tau})')
plt.xlabel('Average Number of Rooms (RM)')
plt.ylabel('Median House Value (MEDV)')
plt.title('Locally Weighted Regression on Boston Housing Data')
plt.legend()
plt.grid(True)
plt.show()
```

Develop a program to demonstrate the working of Linear Regression and Polynomial Regression. Use Boston Housing Dataset for Linear Regression and Auto MPG Dataset (for vehicle fuel efficiency prediction) for Polynomial Regression.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
```

```
# ---------------------- Linear Regression (Boston Housing) ----------------------
#

# Load Boston Housing Dataset

data = pd.read_csv('HousingData.csv')


data


# Features and target selection

X_boston = data[['RM']].values  # Number of rooms feature

y_boston = data['MEDV'].values  # Median value of owner-occupied homes




# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X_boston, y_boston,
test_size=0.2, random_state=42)


# Linear Regression Model

linear_reg = LinearRegression()

linear_reg.fit(X_train, y_train)

y_pred = linear_reg.predict(X_test)


# Evaluation

print("Linear Regression Results (Boston Housing)")

print(f"Mean Squared Error (MSE): {mean_squared_error(y_test,
y_pred):.2f}")

print(f"R-squared (R2): {r2_score(y_test, y_pred):.2f}")
```

 # Plotting Linear Regression

```
plt.scatter(X_test, y_test, color='blue', label='Actual Values')

plt.plot(X_test, y_pred, color='red', linewidth=2, label='Linear Regression Line')

plt.title('Linear Regression - Boston Housing (Rooms vs. Price)')

plt.xlabel('Number of Rooms')

plt.ylabel('Median Value of Owner-Occupied Homes (in $1000)')

plt.legend()

plt.show()
```

 # ---------- Polynomial Regression (Auto MPG) ---------- #

 # Load Auto MPG dataset

```
data1 = pd.read_csv('auto-mpg.csv')


data1


data1.columns
```

 # Data Cleaning

```
data1 = data1.loc[data1['horsepower'] != '?'].copy()  # Removing invalid data and creating a copy

data1.loc[:, 'horsepower'] = data1['horsepower'].astype(float)  # Safe modification using .loc[]
```

 # Features and target

```
X_auto = data1[['horsepower']].values  # Using 'horsepower' as feature

y_auto = data1['mpg'].values           # 'mpg' as target


# Polynomial Transformation

poly = PolynomialFeatures(degree=3)

X_poly = poly.fit_transform(X_auto)


# Train-test split

X_train, X_test, y_train, y_test = train_test_split(X_poly, y_auto,
test_size=0.2, random_state=42)


# Polynomial Regression Model

poly_reg = LinearRegression()

poly_reg.fit(X_train, y_train)

y_pred = poly_reg.predict(X_test)


# Evaluation

print("\nPolynomial Regression Results (Auto MPG)")

print(f"Mean Squared Error (MSE): {mean_squared_error(y_test,
y_pred):.2f}")

print(f"R-squared (R2): {r2_score(y_test, y_pred):.2f}")


# Plotting Polynomial Regression

plt.scatter(X_auto, y_auto, color='blue', label='Actual Values')

plt.scatter(X_auto, poly_reg.predict(poly.transform(X_auto)), color='red',
s=10, label='Predicted Values')
```

```
plt.title('Polynomial Regression - Auto MPG (Horsepower vs. MPG)')

plt.xlabel('Horsepower')

plt.ylabel('Miles Per Gallon (MPG)')

plt.legend()

plt.show()
```

Develop a program to demonstrate the working of the decision tree algorithm. Use Breast Cancer Data set for building the decision tree and apply this knowledge to classify a new sample.

```
import numpy as np

import pandas as pd

from sklearn.datasets import load_breast_cancer

from sklearn.tree import DecisionTreeClassifier, plot_tree

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDisplay, precision_score, recall_score, f1_score

# Load Breast Cancer Dataset

data = load_breast_cancer()

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

y = data.target  # 0 = Malignant, 1 = Benign


# Split data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Create and train Decision Tree model

model = DecisionTreeClassifier(criterion='gini', max_depth=4, random_state=42)

model.fit(X_train, y_train)


# Evaluate Model

y_pred = model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))

print("\nClassification Report:\n", classification_report(y_test, y_pred))



#Confusion Matrix


confusion_matrix = confusion_matrix(y_test, y_pred)

print(confusion_matrix)


#Confusion Matrix Visualization

cm_display = ConfusionMatrixDisplay(confusion_matrix = confusion_matrix, display_labels = [0, 1])


cm_display.plot()

plt.show()


#Performance Metrics

Accuracy = accuracy_score(y_test, y_pred)
```

```
Precision = precision_score(y_test, y_pred)

Sensitivity_recall = recall_score(y_test, y_pred)

Specificity = recall_score(y_test, y_pred, pos_label=0)

F1_score = f1_score(y_test, y_pred)



print({"Accuracy":Accuracy,"Precision":Precision,
"Sensitivity_recall":Sensitivity_recall,"Specificity":Specificity,
"F1_score":F1_score})

# Visualizing the Decision Tree

import matplotlib.pyplot as plt

plt.figure(figsize=(15, 8))

plot_tree(model,feature_names=data.feature_names,
class_names=data.target_names, filled=True)

plt.show()



# Convert new sample to DataFrame with feature names

new_sample = pd.DataFrame([[15.0, 20.0, 100.0, 800.0, 0.1, 0.2, 0.3, 0.1,
0.2, 0.1,

                    0.3, 0.4, 2.0, 30.0, 0.002, 0.004, 0.01, 0.002, 0.007, 0.001,

                    16.0, 25.0, 110.0, 900.0, 0.12, 0.22, 0.35, 0.12, 0.24, 0.15]],

                  columns=data.feature_names)



# Predict using the model

prediction = model.predict(new_sample)

print("\nNew Sample Classification:", 'Benign' if prediction[0] == 1 else
'Malignant')
```

Develop a program to implement the Naive Bayesian classifier considering Olivetti Face Data set for training. Compute the accuracy of the classifier, considering a few test data sets

```
import numpy as np
from sklearn.datasets import fetch_olivetti_faces
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import (
    accuracy_score,
    precision_recall_fscore_support,
    classification_report
)

# 1. Load Olivetti Faces Dataset
data = fetch_olivetti_faces()
X = data.data  # (400, 4096)
y = data.target  # 40 classes

# View the Contents
X
y
# 2. Split 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
)

# 3. Train the Naive Bayes Classifier
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# 4. Predict
y_pred = gnb.predict(X_test)

# 5. Compute Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy: {accuracy * 100:.2f}%")

# 6. Compute Precision, Recall, F1 Score (macro-averaged for multi-class)
precision, recall, f1, _ = precision_recall_fscore_support(
    y_test, y_pred, average='macro'
)
print(f"Macro-Averaged Precision: {precision * 100:.2f}%")
print(f"Macro-Averaged Recall:    {recall * 100:.2f}%")
print(f"Macro-Averaged F1 Score:  {f1 * 100:.2f}%")

# 7. Detailed Classification Report
print("\nClassification Report:")
```

```
print(classification_report(y_test, y_pred, zero_division=0))
```

Develop a program to implement k-means clustering using Wisconsin Breast Cancer data set and visualize the clustering result.

```
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load_breast_cancer

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

from sklearn.decomposition import PCA

from sklearn.metrics import silhouette_score


# Load the Breast Cancer Dataset

data = load_breast_cancer()

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

y = data.target  # Ground truth labels (for reference)


# Standardize the data for better clustering performance

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)


# Apply K-Means Clustering

kmeans = KMeans(n_clusters=2, random_state=42, n_init=10)

y_kmeans = kmeans.fit_predict(X_scaled)
```

```
# Visualizing the Clusters using PCA (2D plot)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)


plt.figure(figsize=(8, 6))
plt.scatter(X_pca[y_kmeans == 0, 0], X_pca[y_kmeans == 0, 1], c='red', label='Cluster 1')
plt.scatter(X_pca[y_kmeans == 1, 0], X_pca[y_kmeans == 1, 1], c='blue', label='Cluster 2')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
            s=200, c='green', marker='X', label='Centroids')
plt.title('K-Means Clustering on Breast Cancer Data')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.legend()
plt.grid(True)
plt.show()
# Evaluating Clustering Performance
silhouette_avg = silhouette_score(X_scaled, y_kmeans)
print(f"Silhouette Score: {silhouette_avg:.2f}")
# Comparing with Original Labels
from sklearn.metrics import accuracy_score
# Since K-Means doesn't label the clusters exactly as 'Malignant' and 'Benign',
# we align them with the ground truth
```

```
labels_corrected = np.where(y_kmeans == 0, 1, 0)

accuracy = accuracy_score(y, labels_corrected)

print(f"Accuracy (based on cluster alignment): {accuracy:.2f}")
```

Signature of the Faculty