```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Generate a random dataset
np.random.seed(42)
data = {
    'make': np.random.choice(['Ford', 'Toyota', 'BMW', 'Honda'], 100),
    'model': np.random.choice(['Model1', 'Model2', 'Model3', 'Model4'], 100),
    'year': np.random.randint(2000, 2023, 100),
    'engine_size': np.random.uniform(1.0, 5.0, 100),
    'num_doors': np.random.choice([2, 4], 100),
    'price': np.random.uniform(15000, 45000, 100)
}

# Create a DataFrame
df = pd.DataFrame(data)

# a) Read the dataset using the Pandas module (already created here)

# b) Print the 1st five rows
print("First five rows of the dataset:")
print(df.head())

# c) Basic statistical computations on the data set or distribution of data
print("\nBasic statistical computations:")
print(df.describe())

# d) The columns and their data types
print("\nColumns and their data types:")
print(df.dtypes)

# e) Detect null values in the dataset. If there are any null values, replace them with mode value
print("\nDetecting null values:")
print(df.isnull().sum())

# Replace null values with mode (although there shouldn't be any in this synthetic data)
df = df.fillna(df.mode().iloc[0])
print("\nNull values after replacement (if any):")
print(df.isnull().sum())


# g) Split the data into test and train
# Encoding categorical features
df_encoded = pd.get_dummies(df, columns=['make', 'model'], drop_first=True)

X = df_encoded.drop('price', axis=1)
y = df_encoded['price']

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

# h) Fit into the model Linear Regression (since Naive Bayes is not suitable for regression)
model = LinearRegression()
model.fit(X_train, y_train)

# i) Predict the model
y_pred = model.predict(X_test)

# j) Find the accuracy of the model (using regression metrics)
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print("\nModel Evaluation Metrics:")
print(f"Mean Absolute Error (MAE): {mae}")
print(f"Mean Squared Error (MSE): {mse}")
print(f"Root Mean Squared Error (RMSE): {rmse}")
print(f"R-squared (R2): {r2}")
```

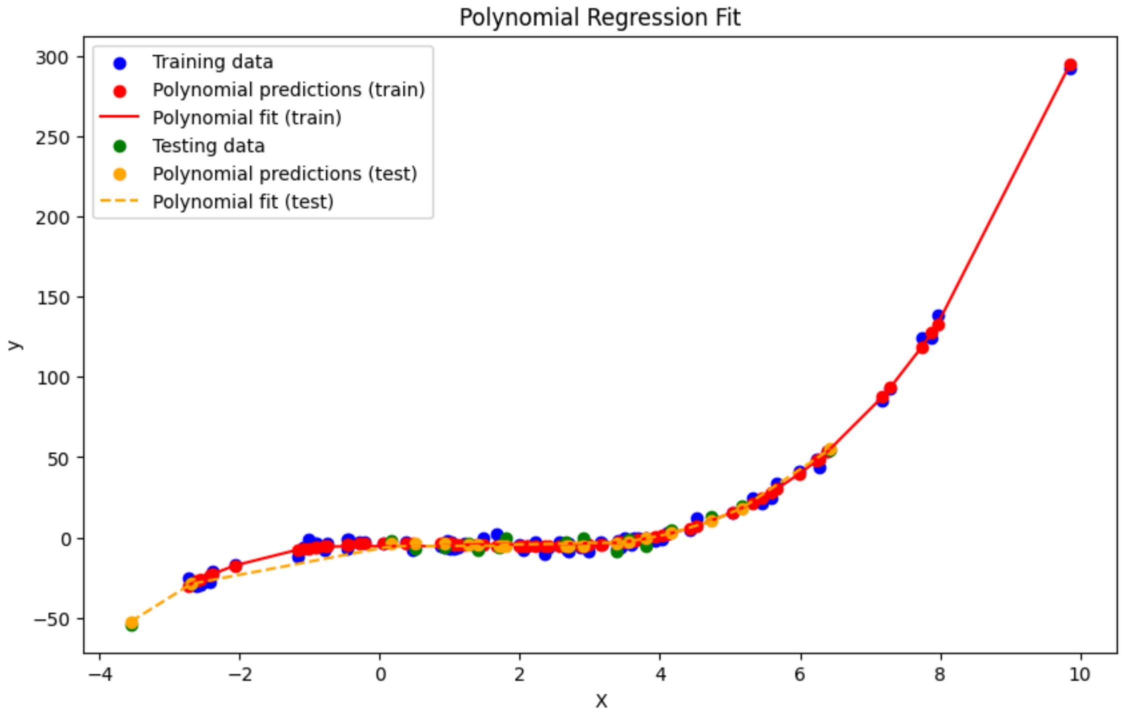
```
First five rows of the dataset:
    make   model  year  engine_size  num_doors         price
0    BMW  Model3  2006     4.746920          4  27588.001873
1  Honda  Model2  2016     1.550084          4  22431.929685
2   Ford  Model2  2019     2.364265          4  25679.180360
3    BMW  Model4  2003     1.453894          4  37735.383314
4    BMW  Model2  2004     4.698774          4  15431.804659

Basic statistical computations:
              year  engine_size  num_doors         price
count   100.000000   100.000000      100.0    100.000000
mean   2010.350000     3.214630        3.1  29889.227910
std       7.101536     1.173796        1.0   8998.292300
min    2000.000000     1.020246        2.0  15325.129544
25%    2004.000000     2.176939        2.0  23199.714854
50%    2010.000000     3.442873        4.0  29790.542697
75%    2016.500000     4.247562        4.0  37661.060495
max    2022.000000     4.960215        4.0  44715.154260

Columns and their data types:
make            object
model           object
year             int64
engine_size    float64
num_doors        int64
price          float64
dtype: object

Detecting null values:
make           0
model          0
year           0
engine_size    0
num_doors      0
price          0
dtype: int64

Null values after replacement (if any):
make           0
model          0
year           0
engine_size    0
num_doors      0
price          0
dtype: int64

Model Evaluation Metrics:
Mean Absolute Error (MAE): 8088.274486557004
Mean Squared Error (MSE): 91855251.51655649
Root Mean Squared Error (RMSE): 9584.11454003741
R-squared (R2): -0.47185899029023926
```

```
import numpy as np

# Define the dataset
data = [
    ['Japan', 'Honda', 'Blue', '1980', 'Economy', 'Positive'],
    ['Japan', 'Toyota', 'Green', '1970', 'Sports', 'Negative'],
    ['Japan', 'Toyota', 'Blue', '1990', 'Economy', 'Positive'],
    ['USA', 'Chrysler', 'Red', '1980', 'Economy', 'Negative'],
    ['Japan', 'Honda', 'White', '1980', 'Economy', 'Positive']
]

# Convert dataset to a numpy array for easier manipulation
data = np.array(data)

# Function to implement Find-S algorithm
def find_s_algorithm(data):
    # Initialize the most specific hypothesis
    hypothesis = ['0'] * (data.shape[1] - 1)

    # Iterate through the dataset
    for instance in data:
        if instance[-1] == 'Positive':
            # Update hypothesis for each positive example
            for i in range(len(hypothesis)):
                if hypothesis[i] == '0':
                    hypothesis[i] = instance[i]
                elif hypothesis[i] != instance[i]:
                    hypothesis[i] = '?'

    return hypothesis

# Apply Find-S algorithm on the dataset
most_specific_hypothesis = find_s_algorithm(data)

print("The most specific hypothesis is:", most_specific_hypothesis)
```

```
The most specific hypothesis is: ['Japan', '?', '?', '?', 'Economy']
```

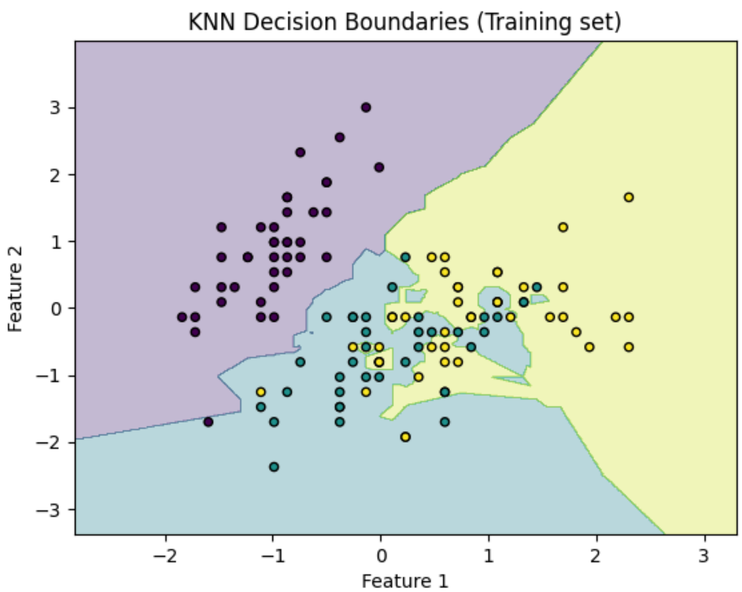
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Generate a synthetic dataset
np.random.seed(42)
X = 2 - 3 * np.random.normal(0, 1, 100)
y = X - 2 * (X ** 2) + 0.5 * (X ** 3) + np.random.normal(-3, 3, 100)

X = X[:, np.newaxis]  # Reshape X to a 2D array

# 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)

# Preprocess the data to include polynomial features
degree = 3  # Degree of the polynomial
poly_features = PolynomialFeatures(degree=degree)
X_train_poly = poly_features.fit_transform(X_train)
X_test_poly = poly_features.transform(X_test)

# Fit a polynomial regression model
model = LinearRegression()
model.fit(X_train_poly, y_train)

# Make predictions
y_train_pred = model.predict(X_train_poly)
y_test_pred = model.predict(X_test_poly)

# Evaluate the model's performance
mse_train = mean_squared_error(y_train, y_train_pred)
mse_test = mean_squared_error(y_test, y_test_pred)
r2_train = r2_score(y_train, y_train_pred)
r2_test = r2_score(y_test, y_test_pred)

print("Training set performance:")
print(f"Mean Squared Error (MSE): {mse_train}")
print(f"R-squared (R2): {r2_train}")

print("\nTesting set performance:")
print(f"Mean Squared Error (MSE): {mse_test}")
print(f"R-squared (R2): {r2_test}")

# Plot the results
plt.figure(figsize=(10, 6))

# Plot training data and predictions
plt.scatter(X_train, y_train, color='blue', label='Training data')
plt.scatter(X_train, y_train_pred, color='red', label='Polynomial predictions (train)')
plt.plot(np.sort(X_train[:, 0]), np.sort(y_train_pred), color='red', label='Polynomial fit (train)')

# Plot testing data and predictions
plt.scatter(X_test, y_test, color='green', label='Testing data')
plt.scatter(X_test, y_test_pred, color='orange', label='Polynomial predictions (test)')
plt.plot(np.sort(X_test[:, 0]), np.sort(y_test_pred), color='orange', linestyle='dashed', label='Polynomial fit (test)')

plt.title('Polynomial Regression Fit')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()
```

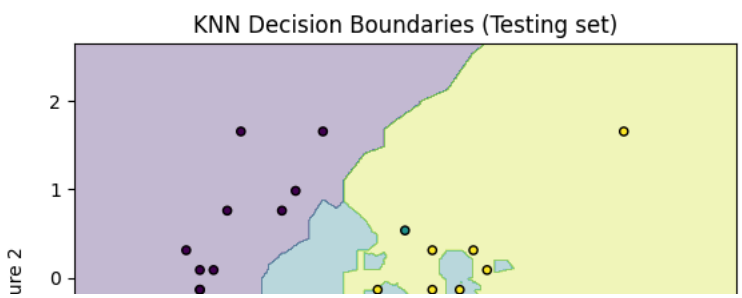
```
Training set performance:
Mean Squared Error (MSE): 7.546412010296407
R-squared (R2): 0.9962534104874832

Testing set performance:
Mean Squared Error (MSE): 7.720695475663606
R-squared (R2): 0.9795784516066509
```



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.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)

# Standardize the features (important for KNN)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the KNN model
k = 3  # Number of neighbors
knn = KNeighborsClassifier(n_neighbors=k)
knn.fit(X_train, y_train)

# Make predictions
y_train_pred = knn.predict(X_train)
y_test_pred = knn.predict(X_test)

# Evaluate the model's performance
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)
conf_matrix = confusion_matrix(y_test, y_test_pred)
class_report = classification_report(y_test, y_test_pred)

print(f"Training set accuracy: {train_accuracy}")
print(f"Testing set accuracy: {test_accuracy}")
print("\nConfusion Matrix:")
print(conf_matrix)
print("\nClassification Report:")
print(class_report)

# Visualize the decision boundaries (for 2D feature space)
def plot_decision_boundaries(X, y, model, title):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                         np.arange(y_min, y_max, 0.01))
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, alpha=0.3)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolor='k', s=20)
    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.show()

# Reduce to 2 features for visualization
X_train_2D = X_train[:, :2]
X_test_2D = X_test[:, :2]

# Train and plot with 2D features
knn_2D = KNeighborsClassifier(n_neighbors=k)
knn_2D.fit(X_train_2D, y_train)

plot_decision_boundaries(X_train_2D, y_train, knn_2D, "KNN Decision Boundaries (Training set)")
plot_decision_boundaries(X_test_2D, y_test, knn_2D, "KNN Decision Boundaries (Testing set)")
```

```
Training set accuracy: 0.9416666666666667
Testing set accuracy: 1.0

Confusion Matrix:
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

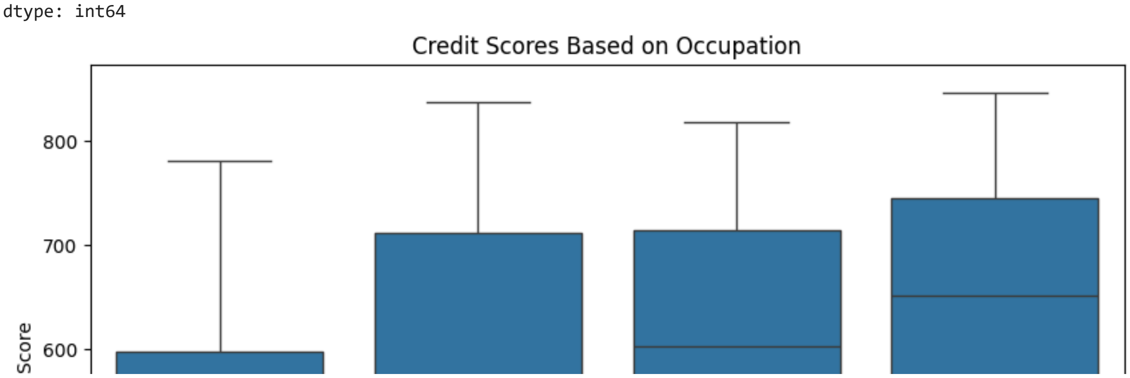




```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report

# Generate a random dataset
np.random.seed(42)

ages = np.random.randint(20, 70, 100)
incomes = np.random.randint(30000, 120000, 100)
occupations = np.random.choice(['Engineer', 'Teacher', 'Doctor', 'Artist'], 100)
credit_scores = np.random.randint(300, 850, 100)
approved = np.random.choice(['Yes', 'No'], 100)

# Create a DataFrame
data = {
    'Age': ages,
    'Income': incomes,
    'Occupation': occupations,
    'CreditScore': credit_scores,
    'Approved': approved
}

df = pd.DataFrame(data)

# a) Print the 1st five rows
print("First five rows of the dataset:")
print(df.head())

# b) Basic statistical computations on the dataset
print("\nBasic statistical computations:")
print(df.describe())

# c) Print the columns and their data types
print("\nColumns and their data types:")
print(df.dtypes)

# d) Detect null values in the dataset and replace them with the mode value
print("\nDetecting null values:")
print(df.isnull().sum())

# Replace null values with mode
for column in df.columns:
    mode_value = df[column].mode()[0]
    df[column].fillna(mode_value, inplace=True)

print("\nNull values after replacement (if any):")
print(df.isnull().sum())

# e) Explore the dataset using a box plot of credit scores based on occupation
plt.figure(figsize=(10, 6))
sns.boxplot(x='Occupation', y='CreditScore', data=df)
plt.title('Credit Scores Based on Occupation')
plt.show()

# f) Split the dataset into training and testing sets
# Assuming 'CreditScore' is the target variable
X = df.drop('CreditScore', axis=1)
y = df['CreditScore']

# One-hot encode categorical variables
X = pd.get_dummies(X, drop_first=True)

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

# g) Fit a Naive Bayes Classifier model
model = GaussianNB()
model.fit(X_train, y_train)

# h) Predict using the model
y_pred = model.predict(X_test)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
```

```
report = classification_report(y_test, y_pred)

print("\nModel Evaluation Metrics:")
print(f"Accuracy: {accuracy}")
print("\nClassification Report:")
print(report)
```

```
First five rows of the dataset:
   Age  Income Occupation  CreditScore Approved
0   58   32568   Engineer          308      Yes
1   48   92592     Artist          532       No
2   34   97563   Engineer          398       No
3   62   32695     Doctor          507       No
4   27   78190   Engineer          430       No

Basic statistical computations:
              Age         Income  CreditScore
count  100.000000     100.00000   100.000000
mean    44.070000   75837.40000   562.410000
std     14.447575   26342.49392   163.560516
min     20.000000   30206.00000   308.000000
25%     33.000000   53229.75000   425.000000
50%     43.000000   79547.50000   548.500000
75%     58.000000   96924.50000   700.750000
max     69.000000  119474.00000   846.000000

Columns and their data types:
Age             int64
Income          int64
Occupation     object
CreditScore     int64
Approved       object
dtype: object

Detecting null values:
Age            0
Income         0
Occupation     0
CreditScore    0
Approved       0
dtype: int64

Null values after replacement (if any):
Age            0
Income         0
Occupation     0
CreditScore    0
Approved       0
dtype: int64
```



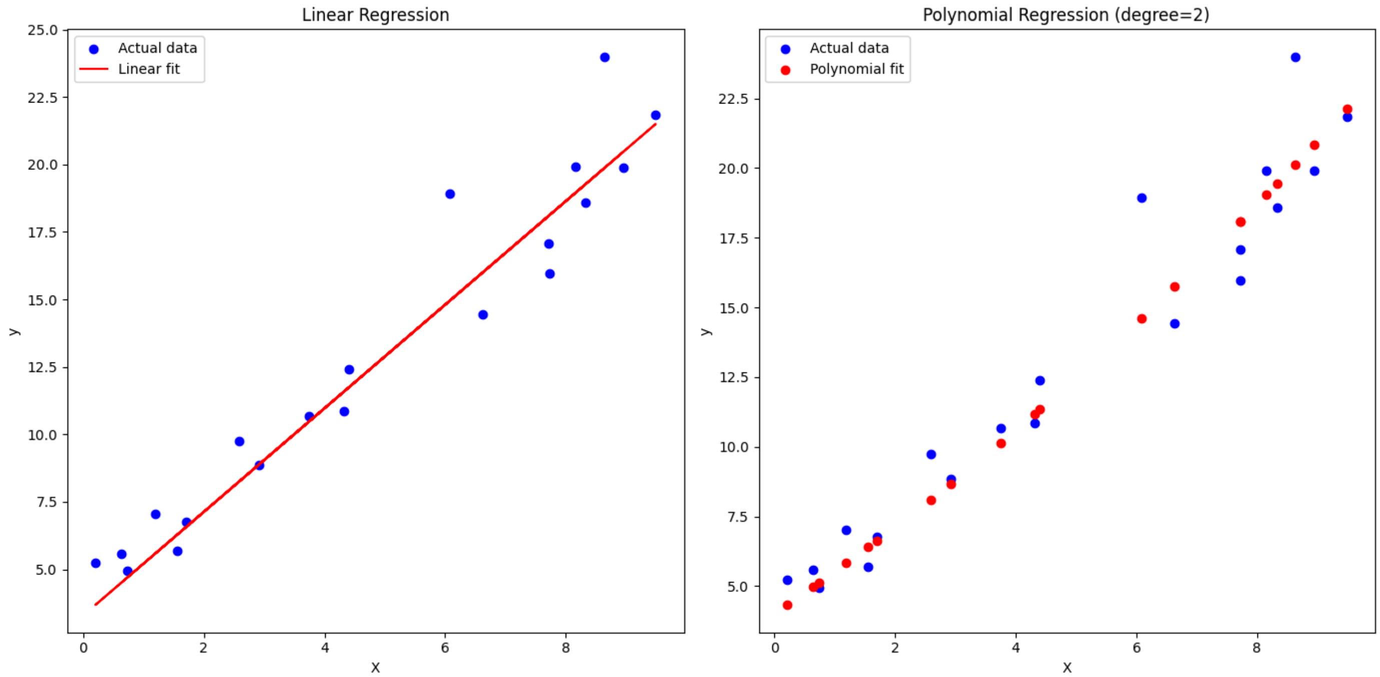
```
import pandas as pd
import numpy as np

# Manually create the dataset
data = {
    'Origin': ['Japan', 'Japan', 'Japan', 'USA', 'Japan'],
    'Manufacturer': ['Honda', 'Toyota', 'Toyota', 'Chrysler', 'Honda'],
    'Color': ['Blue', 'Green', 'Blue', 'Red', 'White'],
    'Decade': ['1980', '1970', '1990', '1980', '1980'],
    'Type': ['Economy', 'Sports', 'Economy', 'Economy', 'Economy'],
    'ExampleType': ['Positive', 'Negative', 'Positive', 'Negative', 'Positive']
}

df = pd.DataFrame(data)

# Initialize the specific boundary (S0) and the general boundary (G0)
def initialize_boundaries(df):
    num_attributes = len(df.columns) - 1
    specific_boundary = ['0'] * num_attributes
    general_boundary = [['?'] * num_attributes]
    return specific_boundary, general_boundary

def is_more_general(h1, h2):
    more_general_parts = []
    for x, y in zip(h1, h2):
        mg = x == '?' or (x != '0' and (x == y or y == '0'))
        more_general_parts.append(mg)
    return all(more_general_parts)

def generalize_specific(h, instance):
    return [i if i == '?' else j if i == '0' else '?' if i != j else i for i, j in zip(h, instance)]

def specialize_general(h, domains, instance):
    specializations = []
    for i, val in enumerate(h):
        if val == '?':
            for domain in domains[i]:
                if domain != instance[i]:
                    new_h = h.copy()
                    new_h[i] = domain
                    specializations.append(new_h)
        elif val != '0' and val != instance[i]:
            new_h = h.copy()
            new_h[i] = '0'
            specializations.append(new_h)
    return specializations

def candidate_elimination(df):
    specific_boundary, general_boundary = initialize_boundaries(df)
    domains = [list(df[col].unique()) for col in df.columns[:-1]]

    for index, row in df.iterrows():
        instance = row[:-1]
        if row['ExampleType'] == 'Positive':
            for i, val in enumerate(instance):
                if specific_boundary[i] == '0':
                    specific_boundary[i] = val
                elif specific_boundary[i] != val:
                    specific_boundary[i] = '?'
            general_boundary = [g for g in general_boundary if is_more_general(g, specific_boundary)]
        elif row['ExampleType'] == 'Negative':
            new_general_boundary = []
            for g in general_boundary:
                if not is_more_general(g, instance):
                    new_general_boundary.append(g)
                else:
                    new_general_boundary.extend(specialize_general(g, domains, instance))
            general_boundary = [g for g in new_general_boundary if any(is_more_general(g, s) for s in [specific_boundary])]

    return specific_boundary, general_boundary

# Apply Candidate Elimination algorithm on the dataset
specific_boundary, general_boundary = candidate_elimination(df)

print("Final Specific Boundary:")
print(specific_boundary)
print("\nFinal General Boundary:")
```

```
for g in general_boundary:
    print(g)
```

```
Final Specific Boundary:
['Japan', '?', '?', '?', 'Economy']

Final General Boundary:
['Japan', '?', '?', '?', 'Economy']
```

```
import numpy as np
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

# Generate a sample dataset
np.random.seed(42)
X = np.random.rand(100, 1) * 10  # Random values between 0 and 10
y = 2 * X + 3 + np.random.randn(100, 1) * 2  # Linear relationship with noise

# 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)

# Linear Regression
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
y_pred_linear = linear_model.predict(X_test)

# Polynomial Regression (degree=2)
poly_features = PolynomialFeatures(degree=2)
X_poly_train = poly_features.fit_transform(X_train)
X_poly_test = poly_features.transform(X_test)

poly_model = LinearRegression()
poly_model.fit(X_poly_train, y_train)
y_pred_poly = poly_model.predict(X_poly_test)

# Performance Evaluation
mse_linear = mean_squared_error(y_test, y_pred_linear)
r2_linear = r2_score(y_test, y_pred_linear)

mse_poly = mean_squared_error(y_test, y_pred_poly)
r2_poly = r2_score(y_test, y_pred_poly)

print("Linear Regression Performance:")
print(f"Mean Squared Error: {mse_linear}")
print(f"R^2 Score: {r2_linear}")

print("\nPolynomial Regression Performance:")
print(f"Mean Squared Error: {mse_poly}")
print(f"R^2 Score: {r2_poly}")

# Plotting the results
plt.figure(figsize=(14, 7))

# Plot Linear Regression results
plt.subplot(1, 2, 1)
plt.scatter(X_test, y_test, color='blue', label='Actual data')
plt.plot(X_test, y_pred_linear, color='red', label='Linear fit')
plt.title('Linear Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()

# Plot Polynomial Regression results
plt.subplot(1, 2, 2)
plt.scatter(X_test, y_test, color='blue', label='Actual data')
plt.scatter(X_test, y_pred_poly, color='red', label='Polynomial fit')
plt.title('Polynomial Regression (degree=2)')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()

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

```
Linear Regression Performance:
Mean Squared Error: 2.614798054868011
R^2 Score: 0.9287298556395621

Polynomial Regression Performance:
Mean Squared Error: 2.5433624291283223
R^2 Score: 0.9306769380727418
```



```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Generate a synthetic dataset
np.random.seed(42)
X, _ = make_blobs(n_samples=300, centers=3, cluster_std=0.60, random_state=0)

# Number of clusters
K = 3

# Initialize the parameters
def initialize_parameters(X, K):
    n, d = X.shape
    pi = np.ones(K) / K
    means = X[np.random.choice(n, K, replace=False)]
    covariances = np.array([np.eye(d)] * K)
    return pi, means, covariances

# E-step: Calculate the responsibilities
def e_step(X, pi, means, covariances):
    n, d = X.shape
    responsibilities = np.zeros((n, K))
    for k in range(K):
        diff = X - means[k]
        exponent = np.einsum('ij, ij -> i', diff @ np.linalg.inv(covariances[k]), diff)
        responsibilities[:, k] = pi[k] * np.exp(-0.5 * exponent) / np.sqrt(np.linalg.det(covariances[k]))
    responsibilities /= responsibilities.sum(axis=1, keepdims=True)
    return responsibilities

# M-step: Update the parameters based on the responsibilities
def m_step(X, responsibilities):
    n, d = X.shape
    N_k = responsibilities.sum(axis=0)
    pi = N_k / n
    means = responsibilities.T @ X / N_k[:, np.newaxis]
    covariances = np.zeros((K, d, d))
    for k in range(K):
        diff = X - means[k]
        covariances[k] = (responsibilities[:, k, np.newaxis, np.newaxis] * np.einsum('ij, ik -> ijk', diff, diff)).sum(axis=0) / N_k[k]
    return pi, means, covariances

# Log-likelihood
def log_likelihood(X, pi, means, covariances):
    n, d = X.shape
    log_likelihood = 0
    for k in range(K):
        diff = X - means[k]
        exponent = np.einsum('ij, ij -> i', diff @ np.linalg.inv(covariances[k]), diff)
        log_likelihood += np.sum(np.log(pi[k] * np.exp(-0.5 * exponent) / np.sqrt(np.linalg.det(covariances[k]))))
    return log_likelihood

# EM Algorithm
def expectation_maximization(X, K, max_iter=100, tol=1e-4):
    pi, means, covariances = initialize_parameters(X, K)
    log_likelihoods = []

    for i in range(max_iter):
        responsibilities = e_step(X, pi, means, covariances)
        pi, means, covariances = m_step(X, responsibilities)
        log_likelihoods.append(log_likelihood(X, pi, means, covariances))

        if i > 0 and abs(log_likelihoods[-1] - log_likelihoods[-2]) < tol:
            break

    return pi, means, covariances, responsibilities, log_likelihoods
```