```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing

data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)

sns.set_style("whitegrid")

plt.figure(figsize=(12, 8))
df.hist(bins=30, figsize=(12, 8), edgecolor='black', grid=False)
plt.suptitle("Histograms of Numerical Features", fontsize=16, fontweight='bold')
plt.show()

plt.figure(figsize=(12, 8))
for i, col in enumerate(df.columns, 1):
    plt.subplot(3, 3, i)
    sns.boxplot(y=df[col], color='skyblue', width=0.5, flierprops={'marker': 'o', 'markerfacecolor':
'red',
    'markersize': 5})
    plt.title(col, fontsize=12, fontweight='bold')
    plt.tight_layout()
    plt.suptitle("Box Plots of Numerical Features", fontsize=16, fontweight='bold', y=1.02)
    plt.show()

print("Summary Statistics:\n")
print(df.describe())

print("Correlation Matrix:\n")
print(df.corr())
```

2

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_california_housing
# Load the California Housing dataset
california = fetch_california_housing(as_frame=True)
df = california.frame
# Compute the correlation matrix
correlation_matrix = df.corr()
# Visualize the correlation matrix using a heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.title('Correlation Matrix of California Housing Features')
plt.show()
# Create a pair plot to visualize pairwise relationships
sns.pairplot(df, diag_kind='kde') #kde or hist
plt.suptitle('Pair Plot of California Housing Features', y=1.02)
plt.show()
```

3

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
target_names = iris.target_names
print("Original shape of X:", X.shape)
print("Original shape of y:", y.shape)
print("Target names:", target_names)
X_mean = np.mean(X, axis=0)
X_std = np.std(X, axis=0)
X_centered = (X - X_mean) / X_std
cov_matrix = np.cov(X_centered.T)
print("Covariance Matrix shape:", cov_matrix.shape)
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors shape:", eigenvectors.shape)
sorted_indices = np.argsort(eigenvalues)[::-1]
sorted_eigenvectors = eigenvectors[:, sorted_indices]
sorted_eigenvalues = eigenvalues[sorted_indices]
print("Sorted Eigenvalues:", sorted_eigenvalues)
n_components = 2
eigenvector_subset = sorted_eigenvectors[:, :n_components]
print(f"Shape of top {n_components} eigenvectors:", eigenvector_subset.shape)
X_reduced = X_centered.dot(eigenvector_subset)
print("Shape of reduced X:", X_reduced.shape)
explained_variance_ratio = sorted_eigenvalues / np.sum(sorted_eigenvalues)
print("Explained variance ratio of each component:")
for i in range(n_components):
    print(f"PC{i+1}: {explained_variance_ratio[i]:.4f}")

total_variance_retained = np.sum(explained_variance_ratio[:n_components])
print(f"Total variance retained in {n_components} components: {total_variance_retained:.4f}")
plt.figure(figsize=(8, 6))
```

```
for target, color, label in zip([0, 1, 2], ['red', 'green', 'blue'], target_names):
    plt.scatter(X_reduced[y == target, 0],
                X_reduced[y == target, 1],
                alpha=0.7,
                color=color,
                label=label)

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA on Iris Dataset (4D reduced to 2D)')
plt.legend()
plt.grid(True)
plt.show()
```

4

```
import csv

def find_s(csv_filepath):
    """
    Implements the Find-S algorithm.

    Args:
        csv_filepath (str): The path to the CSV file containing training data.
                            The last column is assumed to be the target concept
                            ('Yes'/'No' or similar positive/negative labels).

    Returns:
        list: The most specific hypothesis found, represented as a list
              of attribute constraints ('?', 'specific_value', or initial 'âˆ…').
        list: The list of attribute names (header).
    """
    hypothesis = None
    attributes = []
    positive_label = 'Yes' # Assume 'Yes' indicates a positive example

    try:
        with open(csv_filepath, 'r') as f:
            reader = csv.reader(f)

            # Read header to get attribute names and count
            header = next(reader)
            attributes = header[:-1] # All columns except the last one
            num_attributes = len(attributes)

            print(f"Attributes: {attributes}")
            print("-" * 30)

            # Initialize hypothesis
            # Using 'âˆ…' (empty set symbol) for initial specific hypothesis
            hypothesis = ['âˆ…'] * num_attributes
            print(f"Initial Hypothesis: {hypothesis}")
```

```
        # Process training examples
        for i, row in enumerate(reader):
            if not row: # Skip empty rows if any
                continue

            instance = row[:-1]
            target_concept = row[-1]

            print(f"\nProcessing Instance {i+1}: {instance}")
            print(f"Target: {target_concept}")

            # Only process positive examples
            if target_concept.strip().lower() == positive_label.strip().lower():
                print("Instance is Positive. Updating hypothesis...")
                for j in range(num_attributes):
                    # If hypothesis attribute is initial 'âˆ…', set it
                    if hypothesis[j] == 'âˆ…':
                        hypothesis[j] = instance[j]
                    # If hypothesis attribute doesn't match instance, generalize
                    elif hypothesis[j] != instance[j]:
                        hypothesis[j] = '?'
                    # Otherwise (it matches), keep it specific
                    # else: pass
                print(f"Updated Hypothesis: {hypothesis}")
            else:
                print("Instance is Negative. Ignoring.")

    except FileNotFoundError:
        print(f"Error: File not found at {csv_filepath}")
        return None, None
    except Exception as e:
        print(f"An error occurred: {e}")
        return None, None

    # Check if any positive examples were found
    if all(h == 'âˆ…' for h in hypothesis):
        print("\nWarning: No positive examples found in the dataset. Hypothesis remains initial.")
```

```
    return hypothesis, attributes

# --- Demonstration ---
if __name__ == "__main__":
    csv_file = 'training_data.csv'
    print(f"Running Find-S algorithm on '{csv_file}'...")

    final_hypothesis, attribute_names = find_s(csv_file)

    if final_hypothesis is not None:
        print("\n" + "=" * 30)
        print("Find-S Algorithm Finished.")
        print(f"Attribute Names: {attribute_names}")
        print(f"Final Hypothesis: {final_hypothesis}")
        print("=" * 30)

        # Interpretation of the final hypothesis
        if attribute_names:
            print("\nInterpretation:")
            desc = []
            valid_hypothesis = False
            for i in range(len(final_hypothesis)):
                if final_hypothesis[i] != 'âˆ…' and final_hypothesis[i] != '?':
                    desc.append(f"{attribute_names[i]} = {final_hypothesis[i]}")
                    valid_hypothesis = True
                elif final_hypothesis[i] == '?':
                    desc.append(f"{attribute_names[i]} = Any")
                    valid_hypothesis = True
                # If 'âˆ…', it means no positive examples influenced this attribute

            if valid_hypothesis:
                print("The most specific hypothesis consistent with the positive examples is:")
                print(" -> If " + " AND ".join(desc) + ", then EnjoySport = Yes")
            elif all(h == 'âˆ…' for h in final_hypothesis):
                 print("No positive examples were found to form a hypothesis.")
            else:
                 print("Hypothesis could not be fully determined (check for 'âˆ…').")
```

```
import numpy as np
from collections import Counter
np.random.seed(42)
X = np.random.rand(100)
labels = []
for i in range(50):
    if X[i] <= 0.5:
        labels.append('Class1')
    else:
        labels.append('Class2')

X_test = X[50:]
true_labels_test = []
for i in range(50, 100):
    if X[i] <= 0.5:
        true_labels_test.append('Class1')
    else:
        true_labels_test.append('Class2')

def knn_classify(train_points, train_labels, test_point, k):
    distances = np.abs(train_points - test_point)
    k_indices = distances.argsort()[:k]
    k_labels = [train_labels[idx] for idx in k_indices]
    vote_counts = Counter(k_labels)
    return vote_counts.most_common(1)[0][0]
k_values = [1, 2, 3, 4, 5, 20, 30]
results = {}

for k in k_values:
    predicted_labels = []
    for test_point in X_test:
        predicted_label = knn_classify(X[:50], labels, test_point, k)
        predicted_labels.append(predicted_label)
    results[k] = predicted_labels
```

```
for k in k_values:
    print(f"\nk = {k}")
    print("Test Point | Predicted Label | True Label")
    for i, (test_point, pred_label, true_label) in enumerate(zip(X_test, results[k],
true_labels_test), start=51):
        print(f"{test_point:.3f}       | {pred_label}            | {true_label}")
```

6

```
import numpy as np
import matplotlib.pyplot as plt


def gaussian_kernel(x, xi, tau):
    return np.exp(-np.sum((x - xi) ** 2) / (2 * tau ** 2))

def locally_weighted_regression(x, X, y, tau):
    m = X.shape[0]
    weights = np.array([gaussian_kernel(x, X[i], tau) for i in range(m)])
    W = np.diag(weights)
    X_transpose_W = X.T @ W
    theta = np.linalg.inv(X_transpose_W @ X) @ X_transpose_W @ y
    return x @ theta

np.random.seed(42)
X = np.linspace(0, 2 * np.pi, 100)
y = np.sin(X) + 0.1 * np.random.randn(100)
X_bias = np.c_[np.ones(X.shape), X]

x_test = np.linspace(0, 2 * np.pi, 200)
x_test_bias = np.c_[np.ones(x_test.shape), x_test]
tau = 0.5
y_pred = np.array([locally_weighted_regression(xi, X_bias, y, tau) for xi in x_test_bias])

plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='red', label='Training Data', alpha=0.7)
plt.plot(x_test, y_pred, color='blue', label=f'LWR Fit (tau={tau})', linewidth=2)
plt.xlabel('X', fontsize=12)
plt.ylabel('y', fontsize=12)
plt.title('Locally Weighted Regression', fontsize=14)
plt.legend(fontsize=10)
plt.grid(alpha=0.3)
plt.show()
```

7

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

def linear_regression_california():
    housing = fetch_california_housing(as_frame=True)
    X = housing.data[["AveRooms"]]
    y = housing.target

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

    model = LinearRegression()
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)

    plt.scatter(X_test, y_test, color="blue", label="Actual")
    plt.plot(X_test, y_pred, color="red", label="Predicted")
    plt.xlabel("Average number of rooms (AveRooms)")
    plt.ylabel("Median value of homes ($100,000)")
    plt.title("Linear Regression - California Housing Dataset")
    plt.legend()
    plt.show()

    print("Linear Regression - California Housing Dataset")
    print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
    print("R^2 Score:", r2_score(y_test, y_pred))


def polynomial_regression_auto_mpg():
```

```
    url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data"
    column_names = ["mpg", "cylinders", "displacement", "horsepower", "weight",
"acceleration", "model_year", "origin"]
    data = pd.read_csv(url, sep='\s+', names=column_names, na_values="?")
    data = data.dropna()

    X = data["displacement"].values.reshape(-1, 1)
    y = data["mpg"].values

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

    poly_model = make_pipeline(PolynomialFeatures(degree=2), StandardScaler(),
LinearRegression())
    poly_model.fit(X_train, y_train)

    y_pred = poly_model.predict(X_test)

    plt.scatter(X_test, y_test, color="blue", label="Actual")
    plt.scatter(X_test, y_pred, color="red", label="Predicted")
    plt.xlabel("Displacement")
    plt.ylabel("Miles per gallon (mpg)")
    plt.title("Polynomial Regression - Auto MPG Dataset")
    plt.legend()
    plt.show()

    print("Polynomial Regression - Auto MPG Dataset")
    print("Mean Squared Error:", mean_squared_error(y_test, y_pred))
    print("R^2 Score:", r2_score(y_test, y_pred))


if __name__ == "__main__":
    print("Demonstrating Linear Regression and Polynomial Regression\n")
    linear_regression_california()
    polynomial_regression_auto_mpg()
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn import tree

data = load_breast_cancer()
X = data.data
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
clf = DecisionTreeClassifier(random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy * 100:.2f}%")
new_sample = np.array([X_test[0]])
prediction = clf.predict(new_sample)

prediction_class = "Benign" if prediction == 1 else "Malignant"
print(f"Predicted Class for the new sample: {prediction_class}")

plt.figure(figsize=(12,8))
tree.plot_tree(clf, filled=True, feature_names=data.feature_names,
class_names=data.target_names)
plt.title("Decision Tree - Breast Cancer Dataset")
plt.show()
```

9

```
import numpy as np
from sklearn.datasets import fetch_olivetti_faces
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import matplotlib.pyplot as plt

data = fetch_olivetti_faces(shuffle=True, random_state=42)
X = data.data
y = data.target

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

gnb = GaussianNB()
gnb.fit(X_train, y_train)
y_pred = gnb.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

print("\nClassification Report:")
print(classification_report(y_test, y_pred, zero_division=1))

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

cross_val_accuracy = cross_val_score(gnb, X, y, cv=5, scoring='accuracy')
print(f'\nCross-validation accuracy: {cross_val_accuracy.mean() * 100:.2f}%')

fig, axes = plt.subplots(3, 5, figsize=(12, 8))
for ax, image, label, prediction in zip(axes.ravel(), X_test, y_test, y_pred):
    ax.imshow(image.reshape(64, 64), cmap=plt.cm.gray)
    ax.set_title(f"True: {label}, Pred: {prediction}")
    ax.axis('off')

plt.show()
```

10

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
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 confusion_matrix, classification_report

data = load_breast_cancer()
X = data.data
y = data.target

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

kmeans = KMeans(n_clusters=2, random_state=42)
y_kmeans = kmeans.fit_predict(X_scaled)

print("Confusion Matrix:")
print(confusion_matrix(y, y_kmeans))
print("\nClassification Report:")
print(classification_report(y, y_kmeans))

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

df = pd.DataFrame(X_pca, columns=['PC1', 'PC2'])
df['Cluster'] = y_kmeans
df['True Label'] = y

plt.figure(figsize=(8, 6))
sns.scatterplot(data=df, x='PC1', y='PC2', hue='Cluster', palette='Set1', s=100,
edgecolor='black', alpha=0.7)
plt.title('K-Means Clustering of Breast Cancer Dataset')
```

```
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title="Cluster")
plt.show()

plt.figure(figsize=(8, 6))
sns.scatterplot(data=df, x='PC1', y='PC2', hue='True Label', palette='coolwarm', s=100,
edgecolor='black', alpha=0.7)
plt.title('True Labels of Breast Cancer Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title="True Label")
plt.show()

plt.figure(figsize=(8, 6))
sns.scatterplot(data=df, x='PC1', y='PC2', hue='Cluster', palette='Set1', s=100,
edgecolor='black', alpha=0.7)
centers = pca.transform(kmeans.cluster_centers_)
plt.scatter(centers[:, 0], centers[:, 1], s=200, c='red', marker='X', label='Centroids')
plt.title('K-Means Clustering with Centroids')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend(title="Cluster")
plt.show()
```