```python
from tabulate import tabulate

def train_find_s(examples):
    # Initialize hypothesis to the most specific
    hypothesis = examples[0][:-1]  # Exclude the target attribute (last
column)

    for example in examples:
        if example[-1] == "Yes":  # Only consider positive examples
            for i in range(len(hypothesis)):
                if hypothesis[i] != example[i]:
                    hypothesis[i] = "?"  # Generalize if values differ

    return hypothesis

# Sample training data
training_data = [
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Rainy', 'Cold', 'High', 'Strong', 'Warm', 'Change', 'No'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
]

# Column headers
headers = ['Sky', 'AirTemp', 'Humidity', 'Wind', 'Water', 'Forecast',
'EnjoySport']

# Print training data in tabular form
print("Training Data:\n")
print(tabulate(training_data, headers=headers, tablefmt="grid"))

# Train the algorithm
final_hypothesis = train_find_s(training_data)

# Output the final hypothesis
print("\nThe most specific hypothesis is:", final_hypothesis)
```

```python
import csv
from tabulate import tabulate

def load_data(filename):
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        all_rows = list(reader)
        header = all_rows[0]
        data = all_rows[1:]
    return header, data

def more_general(h1, h2):
    """Returns True if h1 is more general than h2."""
    for x, y in zip(h1, h2):
        if x != '?' and (x != y and y != '?'):
            return False
    return True

def candidate_elimination(data):
    n_attr = len(data[0]) - 1
    S = data[0][:-1]
    G = [['?'] * n_attr]

    for example in data:
        instance, label = example[:-1], example[-1]
        if label == 'Yes':
            # Remove inconsistent hypotheses from G
            G = [g for g in G if more_general(g, instance)]

            for i in range(len(S)):
                if S[i] != instance[i]:
                    S[i] = '?'
        else:
            G_temp = []
            for g in G:
                for i in range(len(g)):
                    if g[i] == '?':
                        if S[i] != '?':
                            g_new = g.copy()
                            g_new[i] = S[i]
                            if g_new not in G_temp:
                                G_temp.append(g_new)
            G = G_temp

    return S, G

# Load training data from CSV
header, training_data = load_data("training_data.csv")

# Print table of training data
print("\nTraining Data:\n")
print(tabulate(training_data, headers=header, tablefmt="grid"))

# Run Candidate Elimination
```

```python
S_final, G_final = candidate_elimination(training_data)

# Output
print("\nFinal Specific Hypothesis (S):", S_final)
print("\nFinal General Hypotheses (G):")
for g in G_final:
    print(g)
```

```
import math
from collections import Counter

# Sample dataset
dataset = [
    ['Sunny', 'Hot', 'High', 'Weak', 'No'],
    ['Sunny', 'Hot', 'High', 'Strong', 'No'],
    ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Cool', 'Normal', 'Strong', 'No'],
    ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
    ['Sunny', 'Mild', 'High', 'Weak', 'No'],
    ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'Normal', 'Weak', 'Yes'],
    ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
    ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
    ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
    ['Rain', 'Mild', 'High', 'Strong', 'No']
]

attributes = ['Outlook', 'Temperature', 'Humidity', 'Wind']

# Helper functions
def entropy(examples):
    total = len(examples)
    label_counts = Counter(row[-1] for row in examples)
    return -sum((count / total) * math.log2(count / total) for count in
label_counts.values())

def info_gain(examples, attr_index):
    total_entropy = entropy(examples)
    subsets = {}
    for row in examples:
        key = row[attr_index]
        subsets.setdefault(key, []).append(row)
    subset_entropy = sum((len(subset) / len(examples)) * entropy(subset)
for subset in subsets.values())
    return total_entropy - subset_entropy

def majority_class(examples):
    labels = [row[-1] for row in examples]
    return Counter(labels).most_common(1)[0][0]

def id3(examples, attrs):
    labels = [row[-1] for row in examples]
    if labels.count(labels[0]) == len(labels):
        return labels[0]
    if not attrs:
        return majority_class(examples)

    gains = [info_gain(examples, attributes.index(attr)) for attr in
attrs]
    best_attr = attrs[gains.index(max(gains))]
```

```
    tree = {best_attr: {}}
    attr_index = attributes.index(best_attr)
    attr_values = set(row[attr_index] for row in examples)

    for val in attr_values:
        subset = [row for row in examples if row[attr_index] == val]
        if not subset:
            tree[best_attr][val] = majority_class(examples)
        else:
            new_attrs = [a for a in attrs if a != best_attr]
            tree[best_attr][val] = id3(subset, new_attrs)

    return tree

# Build the tree
decision_tree = id3(dataset, attributes)

# Function to classify a sample
def classify(tree, sample):
    if isinstance(tree, str):
        return tree
    attr = next(iter(tree))
    attr_index = attributes.index(attr)
    attr_value = sample[attr_index]
    subtree = tree[attr].get(attr_value)
    if not subtree:
        return "Unknown"
    return classify(subtree, sample)

# Print the tree
import pprint
print("Decision Tree:")
pprint.pprint(decision_tree)

# Classify a new sample
sample = ['Sunny', 'Cool', 'High', 'Strong']
print("\nClassifying sample:", sample)
result = classify(decision_tree, sample)
print("Prediction:", result)
```

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split into train/test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Standardize features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create ANN with 1 hidden layer (10 neurons), using backpropagation
mlp = MLPClassifier(hidden_layer_sizes=(10,), activation='relu',
solver='adam', max_iter=1000, random_state=1)

# Train model
mlp.fit(X_train, y_train)

# Predict
y_pred = mlp.predict(X_test)

# Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred,
target_names=iris.target_names))
```

```
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 classification_report, accuracy_score

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

# Split into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Normalize the feature data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Create and train the KNN model (k=3)
knn = KNeighborsClassifier(n_neighbors=3)
knn.fit(X_train, y_train)

# Predict on test data
y_pred = knn.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred,
target_names=iris.target_names))
```

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score,
classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Create and train the Naïve Bayes model
model = GaussianNB()
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred,
target_names=iris.target_names))

# Generate confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=iris.target_names,
            yticklabels=iris.target_names)
plt.title("Confusion Matrix - Naïve Bayes")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, accuracy_score,
confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Load dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

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

# Create and train Logistic Regression model
lr = LogisticRegression(max_iter=200)
lr.fit(X_train, y_train)

# Predict on test data
y_pred = lr.predict(X_test)

# Evaluate the model
print("Accuracy:", accuracy_score(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred,
target_names=iris.target_names))

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, cmap='Blues', fmt='d',
            xticklabels=iris.target_names,
            yticklabels=iris.target_names)
plt.title("Confusion Matrix - Logistic Regression")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

```
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Generate synthetic regression dataset
X, y = make_regression(n_samples=100, n_features=1, noise=10,
random_state=42)

# 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)

# Create and train the Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate the model
print("Mean Squared Error (MSE):", mean_squared_error(y_test, y_pred))
print("R^2 Score (Coefficient of Determination):", r2_score(y_test,
y_pred))

# Plotting the regression line
plt.scatter(X_test, y_test, color='blue', label='Actual')
plt.plot(X_test, y_pred, color='red', linewidth=2, label='Predicted')
plt.title("Linear Regression - Actual vs Predicted")
plt.xlabel("Feature")
plt.ylabel("Target")
plt.legend()
plt.show()
```

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

# Generate synthetic non-linear data
np.random.seed(0)
X = np.sort(5 * np.random.rand(100, 1), axis=0)
y = np.sin(X).ravel() + np.random.normal(0, 0.2, X.shape[0])  # Non-
linear function with noise

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

# Linear Regression
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
y_pred_lin = lin_reg.predict(X_test)

# Polynomial Regression (degree = 4)
poly = PolynomialFeatures(degree=4)
X_train_poly = poly.fit_transform(X_train)
X_test_poly = poly.transform(X_test)

poly_reg = LinearRegression()
poly_reg.fit(X_train_poly, y_train)
y_pred_poly = poly_reg.predict(X_test_poly)

# Evaluation
print("Linear Regression R² Score:", r2_score(y_test, y_pred_lin))
print("Polynomial Regression R² Score:", r2_score(y_test, y_pred_poly))

# Visualization
X_range = np.linspace(0, 5, 100).reshape(-1, 1)
y_range_lin = lin_reg.predict(X_range)
y_range_poly = poly_reg.predict(poly.transform(X_range))

plt.scatter(X, y, color='blue', label='Actual Data')
plt.plot(X_range, y_range_lin, color='red', label='Linear Regression')
plt.plot(X_range, y_range_poly, color='green', label='Polynomial
Regression (deg=4)')
plt.title("Comparison of Linear vs Polynomial Regression")
plt.xlabel("X")
plt.ylabel("y")
plt.legend()
plt.show()
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score

# Generate synthetic data with 3 clusters
X, y_true = make_blobs(n_samples=300, centers=3, cluster_std=0.60,
random_state=0)

# Fit a Gaussian Mixture Model using EM algorithm
gmm = GaussianMixture(n_components=3, random_state=0)
gmm.fit(X)

# Predict cluster labels
labels = gmm.predict(X)

# Evaluate clustering
print("Cluster Centers:\n", gmm.means_)
print("Silhouette Score:", silhouette_score(X, labels))

# Visualize clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis')
plt.scatter(gmm.means_[:, 0], gmm.means_[:, 1], s=100, c='red',
marker='X', label='Centroids')
plt.title("Clustering using EM (GMM)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()
```