1. FIND-S algorithm.py

```
def find_s_algorithm(training_data):
    hypothesis = training_data[0][:-1]


    for example in training_data:
        if example[-1] == 'Yes':
            for i in range(len(hypothesis)):


                if example[i] != hypothesis[i]:
                    hypothesis[i] = '?'


    return hypothesis
training_data = [
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Rainy', 'Cold', 'High', 'Weak', 'Cool', 'Change', 'No'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
]
result_hypothesis = find_s_algorithm(training_data)
print("Result Hypothesis:", result_hypothesis)
```

2. Candidate elimination.py

```
import copy


def initialize_hypotheses(n):
    hypotheses = []
    specific_hypothesis = ['0'] * n
    general_hypothesis = ['?'] * n
    hypotheses.append(specific_hypothesis)
```

```
    hypotheses.append(general_hypothesis)
    return hypotheses

def candidate_elimination(training_data):
    num_attributes = len(training_data[0]) - 1
    hypotheses = initialize_hypotheses(num_attributes)
    for example in training_data:
        if example[-1] == 'Yes':
            for i in range(num_attributes):
                if hypotheses[0][i] != '0' and hypotheses[0][i] != example[i]:
                    hypotheses[0][i] = '?'
                for h in hypotheses[1:]:
                    if h[i] != '?' and h[i] != example[i]:
                        hypotheses.remove(h)
        else:
            temp_hypotheses = copy.deepcopy(hypotheses)
            for h in temp_hypotheses:
                if h[:-1] != example[:-1] + ['?']:
                    hypotheses.remove(h)
                for i in range(num_attributes):
                    if example[i] != h[i] and h[i] != '?':
                        new_hypothesis = copy.deepcopy(h)
                        new_hypothesis[i] = '?'
                        if new_hypothesis not in hypotheses:
                            hypotheses.append(new_hypothesis)

    return hypotheses
training_data = [
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Sunny', 'Warm', 'High', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Rainy', 'Cold', 'High', 'Weak', 'Cool', 'Change', 'No'],
```

```
    ['Sunny', 'Warm', 'High', 'Strong', 'Cool', 'Change', 'Yes']
]
result_hypotheses = candidate_elimination(training_data)
print("Result Hypotheses:")
for h in result_hypotheses:
    print(h)
```

3. ID3 decision tree.py

```
class Node:
    def __init__(self, attribute=None, value=None, results=None, true_branch=None, false_branch=None):
        self.attribute, self.value, self.results, self.true_branch, self.false_branch = attribute, value, results, true_branch, false_branch

def build_tree(rows):
    if not rows:
        return Node()

    if len(set(row[-1] for row in rows)) == 1:
        return Node(results=rows[0][-1])

    num_attributes = len(rows[0]) - 1
    best_attribute = max(range(num_attributes), key=lambda col: information_gain(rows, col))

    true_rows = [row for row in rows if row[best_attribute] == 'Yes']
    false_rows = [row for row in rows if row[best_attribute] == 'No']

    true_branch = build_tree(true_rows)
    false_branch = build_tree(false_rows)
```

```
    return Node(attribute=best_attribute, value=rows[0][best_attribute], true_branch=true_branch,
false_branch=false_branch)


def information_gain(rows, col):
    total_entropy = entropy(rows)
    values = set(row[col] for row in rows)
    weighted_entropy = sum(len(list(filter(lambda row: row[col] == val, rows))) / len(rows) *
entropy(list(filter(lambda row: row[col] == val, rows))) for val in values)
    return total_entropy - weighted_entropy


def entropy(rows):
    from math import log2
    counts = class_counts(rows)
    return -sum(count / len(rows) * log2(count / len(rows)) for count in counts.values())


def class_counts(rows):
    return dict((row[-1], rows.count(row)) for row in rows)


# Example dataset (you can modify this as needed)
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']
]
```

```
# Build the decision tree
tree = build_tree(dataset)

# Print the decision tree
def print_tree(node, indent=""):
    if node is None:
        return
    if node.results is not None:
        print(indent + str(node.results))
    else:
        print(indent + f'Attribute {node.attribute} : {node.value}? ')
        print(indent + '--> True:')
        print_tree(node.true_branch, indent + '  ')
        print(indent + '--> False:')
        print_tree(node.false_branch, indent + '  ')

print_tree(tree)

# Classify a new sample
new_sample = ['Sunny', 'Cool', 'High', 'Strong']
current_node = tree
while current_node.results is None and current_node.attribute is not None:
    if new_sample[current_node.attribute] == current_node.value:
        current_node = current_node.true_branch
    else:
        current_node = current_node.false_branch

print(f"\nClassification result for {new_sample}: {current_node.results}")
```

4. **ANN using backpropagation.py**

```
import numpy as np


def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):
    return x * (1 - x)


class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.biases_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.biases_output = np.zeros((1, output_size))

    def forward(self, inputs):
        self.hidden = sigmoid(np.dot(inputs, self.weights_input_hidden) + self.biases_hidden)
        self.output = sigmoid(np.dot(self.hidden, self.weights_hidden_output) + self.biases_output)
        return self.output

    def backward(self, inputs, targets, learning_rate):
        output_error = targets - self.output
        output_delta = output_error * sigmoid_derivative(self.output)
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.hidden)

        self.weights_hidden_output += self.hidden.T.dot(output_delta) * learning_rate
        self.biases_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += inputs.T.dot(hidden_delta) * learning_rate
```

```
        self.biases_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate

    def train(self, inputs, targets, epochs, learning_rate):
        for _ in range(epochs):
            for i in range(len(inputs)):
                self.forward(inputs[i:i+1])
                self.backward(inputs[i:i+1], targets[i:i+1], learning_rate)

    def predict(self, inputs):
        return self.forward(inputs)

# Example dataset (you can modify this as needed)
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [0]])

# Create and train the neural network
neural_network = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
neural_network.train(inputs, targets, epochs=10000, learning_rate=0.1)

# Test the neural network
for i in range(len(inputs)):
    prediction = neural_network.predict(inputs[i:i+1])
    print(f"Input: {inputs[i]}, Target: {targets[i]}, Prediction: {prediction}")
```

5. **KNN.py**

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import numpy as np
```

```
iris = load_iris()
X, y = iris.data, iris.target

np.random.seed(42)
noise = np.random.choice([0, 1], size=len(y), p=[0.1, 0.9])
y_noisy = (y + noise) % 3

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

knn_classifier = KNeighborsClassifier(n_neighbors=3)

knn_classifier.fit(X_train, y_train)

y_pred = knn_classifier.predict(X_test)

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

6. **Naive bayes.py**

```
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

X, y = load_iris(return_X_y=True)

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

naive_bayes_classifier = GaussianNB()
```

```
naive_bayes_classifier.fit(X_train, y_train)

y_pred = naive_bayes_classifier.predict(X_test)

conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)

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

7. **Logistic Regression.py**

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import numpy as np

# Load the iris dataset
X, y = load_iris(return_X_y=True)

# Add some random noise to labels
np.random.seed(42)
y_noisy = (y + np.random.choice([1, -1], size=len(y), p=[0.1, 0.9])) % 3

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_noisy, test_size=0.2, random_state=42)

# Initialize and train the Logistic Regression model with increased max_iter
logistic_regression_model = LogisticRegression(max_iter=1000).fit(X_train, y_train)
```

```
# Make predictions on the test set
y_pred = logistic_regression_model.predict(X_test)

# Display accuracy
print(f"Accuracy: {accuracy_score(y_test, y_pred) * 100:.2f}%")
```

8. **Linear regression.py**

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import accuracy_score
import numpy as np

# Load the iris dataset
X, y = load_iris(return_X_y=True)

# Convert the problem into a binary classification task
y_binary = (y == 0).astype(int)  # 1 if class 0 (setosa), 0 otherwise

# Introduce more random noise to target values
np.random.seed(42)
y_noisy = y_binary + np.random.normal(scale=0.4, size=len(y_binary))

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, (y_noisy >= 0.5).astype(int), test_size=0.2,
random_state=42)

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

```
# Make predictions on the test set
y_pred = linear_regression_model.predict(X_test)
y_pred_binary = (y_pred >= 0.5).astype(int)  # Convert predicted probabilities to binary

# Display accuracy
accuracy = accuracy_score(y_test, y_pred_binary) * 100
print(f"Accuracy: {accuracy:.2f}%")
```

9. **linear and polynomial.py**

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

# Generate some random data for demonstration
np.random.seed(42)
X = np.sort(5 * np.random.rand(80, 1), axis=0)
y = np.sin(X).ravel() + np.random.normal(0, 0.1, X.shape[0])

# Reshape X to a column vector
X = X.reshape(-1, 1)

# Fit Linear Regression
linear_model = LinearRegression()
linear_model.fit(X, y)
y_linear_pred = linear_model.predict(X)

# Fit Polynomial Regression
```

```
poly_features = PolynomialFeatures(degree=3)
X_poly = poly_features.fit_transform(X)
poly_model = LinearRegression()
poly_model.fit(X_poly, y)
y_poly_pred = poly_model.predict(X_poly)

# Plot the results
plt.scatter(X, y, s=10, label='Data')
plt.plot(X, y_linear_pred, label='Linear Regression', color='red')
plt.plot(X, y_poly_pred, label='Polynomial Regression', color='green')
plt.xlabel('X')
plt.ylabel('y')
plt.legend()
plt.show()

# Calculate Mean Squared Error for both models
mse_linear = mean_squared_error(y, y_linear_pred)
mse_poly = mean_squared_error(y, y_poly_pred)

print(f"Linear Regression MSE: {mse_linear:.4f}")
print(f"Polynomial Regression MSE: {mse_poly:.4f}")
```

10. Expectation & Maximization Algorithm.py

```
import numpy as np
from scipy.stats import norm

# Generate some sample data
np.random.seed(42)
data = np.concatenate([np.random.normal(0, 1, 100), np.random.normal(5, 1, 100)])
```

```
# Initialize parameters
mu1, mu2 = np.random.rand(2) * 10
sigma1, sigma2 = np.random.rand(2) * 5
pi = 0.5

# EM algorithm
for _ in range(100):
    # Expectation step
    likelihood1 = norm.pdf(data, mu1, sigma1)
    likelihood2 = norm.pdf(data, mu2, sigma2)
    weight1 = pi * likelihood1 / (pi * likelihood1 + (1 - pi) * likelihood2)
    weight2 = 1 - weight1

    # Maximization step
    mu1 = np.sum(weight1 * data) / np.sum(weight1)
    mu2 = np.sum(weight2 * data) / np.sum(weight2)
    sigma1 = np.sqrt(np.sum(weight1 * (data - mu1)**2) / np.sum(weight1))
    sigma2 = np.sqrt(np.sum(weight2 * (data - mu2)**2) / np.sum(weight2))
    pi = np.mean(weight1)

# Print the final parameters
print("Final Parameters:")
print(f"Cluster 1 - Mean: {mu1:.2f}, Standard Deviation: {sigma1:.2f}")
print(f"Cluster 2 - Mean: {mu2:.2f}, Standard Deviation: {sigma2:.2f}")
print(f"Cluster Weights - Cluster 1: {pi:.2f}, Cluster 2: {1 - pi:.2f}")
```

11. Credit Score Classification.py

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score, classification_report

from sklearn.datasets import make_classification


X, y = make_classification(n_samples=1000, n_features=5, n_informative=3, n_redundant=1,
random_state=42)


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


decision_tree_classifier = DecisionTreeClassifier()

decision_tree_classifier.fit(X_train, y_train)


y_pred = decision_tree_classifier.predict(X_test)


accuracy = accuracy_score(y_test, y_pred)

print(f"Accuracy: {accuracy:.2f}")

print("Classification Report:")

print(classification_report(y_test, y_pred))
```

12. **Iris Flower Classification using KNN.py**

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy_score

import numpy as np


X, y = load_iris(return_X_y=True)


np.random.seed(42)

y_noisy = y.copy()
```

```
mask = np.random.choice([True, False], size=len(y), p=[0.1, 0.9])
y_noisy[mask] = np.random.randint(0, 3, size=np.sum(mask))

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

knn_classifier = KNeighborsClassifier(n_neighbors=3)
knn_classifier.fit(X_train, y_train)
y_pred = knn_classifier.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)*100
print(f"Accuracy: {accuracy:.2f}")
```

13. **Car Price Prediction Model.py**

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Sample car dataset (for demonstration purposes)
# You should replace this with a more extensive and relevant dataset
car_features = np.array([[2000, 150000, 4], [2010, 80000, 2], [2015, 50000, 1]])
car_prices = np.array([5000, 15000, 25000])

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(car_features, car_prices, test_size=0.2,
random_state=42)

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

```
# Make predictions on the test set
y_pred = linear_regression_model.predict(X_test)

# Display mean squared error (for simplicity, not accuracy)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")

# Example: Predict the price for a new car
new_car_features = np.array([[2022, 10000, 2]])
predicted_price = linear_regression_model.predict(new_car_features)
print(f"Predicted Price for the New Car: ${predicted_price[0]:,.2f}")
```

14. House price Prediction.py

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Sample house dataset (for demonstration purposes)
# You should replace this with a more extensive and relevant dataset
house_features = np.array([[1500, 3, 20], [2000, 4, 15], [1200, 2, 25]])
house_prices = np.array([200000, 300000, 150000])

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(house_features, house_prices, test_size=0.2,
random_state=42)

# Initialize and train the Linear Regression model
linear_regression_model = LinearRegression()
```

```
linear_regression_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = linear_regression_model.predict(X_test)

# Display mean squared error (for simplicity, not accuracy)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")

# Example: Predict the price for a new house
new_house_features = np.array([[1800, 3, 18]])
predicted_price = linear_regression_model.predict(new_house_features)
print(f"Predicted Price for the New House: ${predicted_price[0]:,.2f}")
```

15. **Iris Flower Classification using Naive Bayes classifier.py**

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
import numpy as np

# Load the iris dataset
X, y = load_iris(return_X_y=True)

# Introduce some random noise to the labels
np.random.seed(42)
y_noisy = y.copy()
mask = np.random.choice([True, False], size=len(y), p=[0.1, 0.9])
y_noisy[mask] = np.random.randint(0, 3, size=np.sum(mask))
```

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_noisy, test_size=0.2, random_state=42)

# Initialize and train the Gaussian Naive Bayes classifier
naive_bayes_classifier = GaussianNB()
naive_bayes_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = naive_bayes_classifier.predict(X_test)

# Display accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

16. **Classification Algorithms and evaluate their performance..py**

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import numpy as np

# Load the iris dataset
X, y = load_iris(return_X_y=True)

# Introduce some random noise to the labels
np.random.seed(42)
y_noisy = y.copy()
mask = np.random.choice([True, False], size=len(y), p=[0.1, 0.9])
```

```
y_noisy[mask] = np.random.randint(0, 3, size=np.sum(mask))

# Split the noisy dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_noisy, test_size=0.2, random_state=42)

# Initialize and train Decision Tree classifier
decision_tree_classifier = DecisionTreeClassifier()
decision_tree_classifier.fit(X_train, y_train)
y_pred_dt = decision_tree_classifier.predict(X_test)
accuracy_dt = accuracy_score(y_test, y_pred_dt)

# Initialize and train Support Vector Machine (SVM) classifier
svm_classifier = SVC()
svm_classifier.fit(X_train, y_train)
y_pred_svm = svm_classifier.predict(X_test)
accuracy_svm = accuracy_score(y_test, y_pred_svm)

# Initialize and train K-Nearest Neighbors (KNN) classifier
knn_classifier = KNeighborsClassifier()
knn_classifier.fit(X_train, y_train)
y_pred_knn = knn_classifier.predict(X_test)
accuracy_knn = accuracy_score(y_test, y_pred_knn)

# Display accuracy for each classifier
print(f"Decision Tree Accuracy: {accuracy_dt:.2f}")
print(f"SVM Accuracy: {accuracy_svm:.2f}")
print(f"KNN Accuracy: {accuracy_knn:.2f}")
```

17. **Mobile Price Prediction.py**

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score

import numpy as np


# Sample mobile dataset (for demonstration purposes)

# You should replace this with a more extensive and relevant dataset

mobile_features = [

    [5.5, 32, 3, 13],

    [6.0, 64, 4, 15],

    [4.7, 16, 2, 10],

    [5.2, 32, 3, 12],

    [6.1, 128, 4, 16],

    [4.5, 16, 2, 9],

]


# Introduce more random noise to the labels

np.random.seed(42)

mobile_labels = np.array([1, 2, 1, 1, 3, 1])  # Assuming 1, 2, and 3 represent different price ranges

noise_mask = np.random.choice([True, False], size=len(mobile_labels), p=[0.7, 0.3])

mobile_labels[noise_mask] = np.random.randint(1, 4, size=np.sum(noise_mask))


# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(mobile_features, mobile_labels, test_size=0.2, 
random_state=42)


# Initialize and train the Decision Tree classifier

decision_tree_classifier = DecisionTreeClassifier()

decision_tree_classifier.fit(X_train, y_train)


# Make predictions on the test set

y_pred = decision_tree_classifier.predict(X_test)
```

```
# Display accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Example: Predict the price range for a new mobile
new_mobile_features = [[5.8, 64, 3, 14]]
predicted_price_range = decision_tree_classifier.predict(new_mobile_features)
print(f"Predicted Price Range for the New Mobile: {predicted_price_range[0]}")
```

18. **Perceptron based IRIS classification.py**

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score

# Load the iris dataset
X, y = load_iris(return_X_y=True)

# Split the dataset 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)

# Initialize and train the Perceptron classifier
perceptron_classifier = Perceptron()
perceptron_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = perceptron_classifier.predict(X_test)

# Display accuracy (for simplicity, not accuracy)
```

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

19.Naive Bayes classification for Bank Loan prediction.py

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

# Sample bank loan dataset (for demonstration purposes)
# You should replace this with a more extensive and relevant dataset
loan_features = [
    [25, 50000, 1],  # Age, Income, Education Level (1: Low, 2: Medium, 3: High)
    [35, 80000, 2],
    [45, 120000, 3],
    [30, 60000, 1],
    [40, 100000, 2],
]

loan_labels = [0, 1, 1, 0, 1]  # 0: Not Approved, 1: Approved

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(loan_features, loan_labels, test_size=0.2, random_state=42)

# Initialize and train the Gaussian Naive Bayes classifier
naive_bayes_classifier = GaussianNB()
naive_bayes_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = naive_bayes_classifier.predict(X_test)
```

```
# Display accuracy (for simplicity, not accuracy)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

20. **Future Sales Prediction.py**

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import numpy as np

# Sample future sales dataset (for demonstration purposes)
# You should replace this with a more extensive and relevant dataset
sales_features = np.array([[1], [2], [3], [4], [5]])
sales_targets = np.array([100, 150, 200, 250, 300])  # Sales for each corresponding feature

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(sales_features, sales_targets, test_size=0.2,
random_state=42)

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

# Make predictions on the test set
y_pred = linear_regression_model.predict(X_test)

# Display predicted future sales
print("Predicted Future Sales:")
```

```
for i, pred in enumerate(y_pred):
    print(f"Feature: {X_test[i][0]}, Predicted Sales: {pred:.2f}")
```