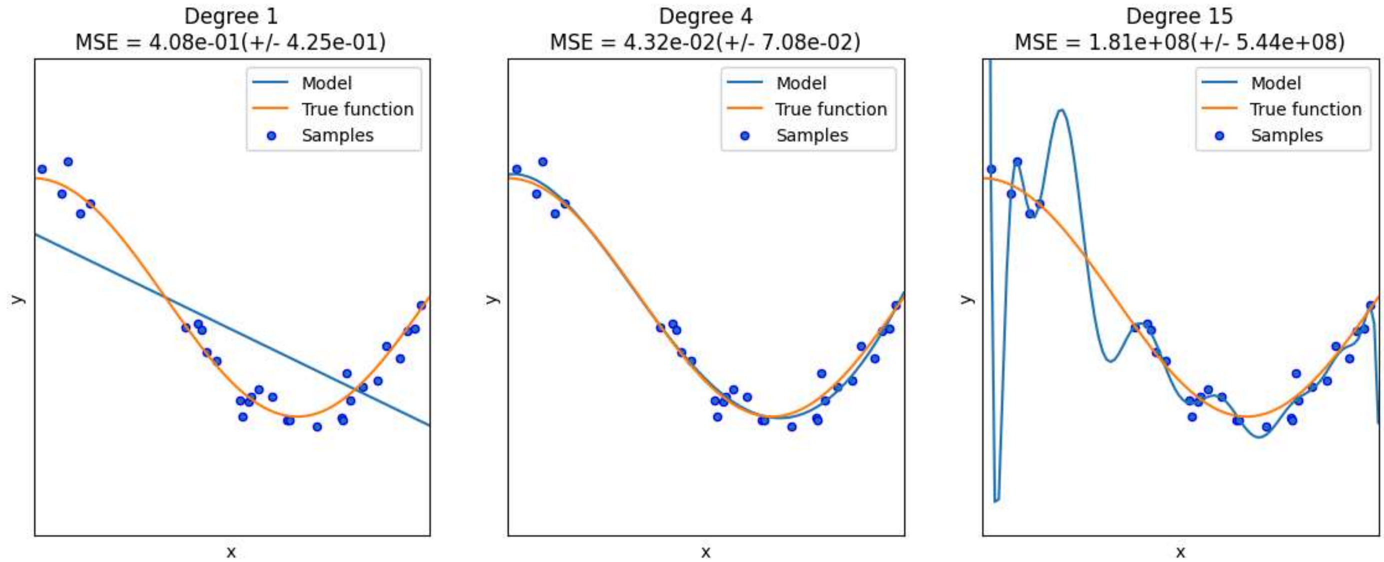
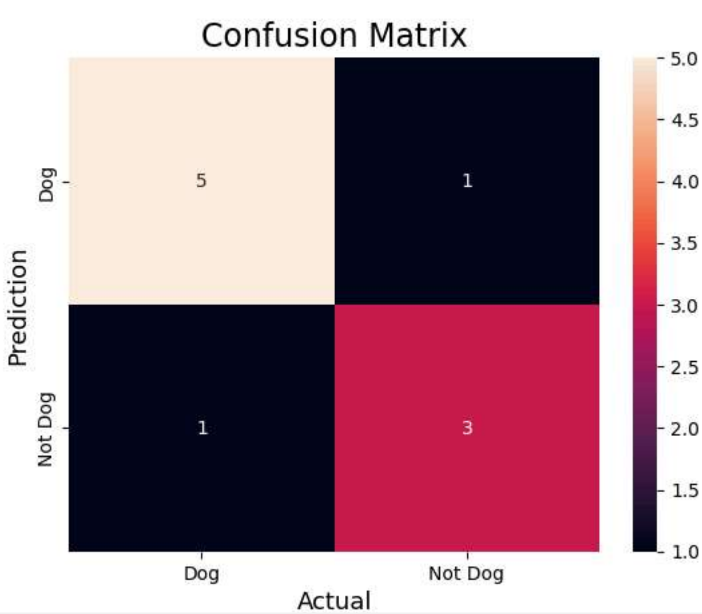
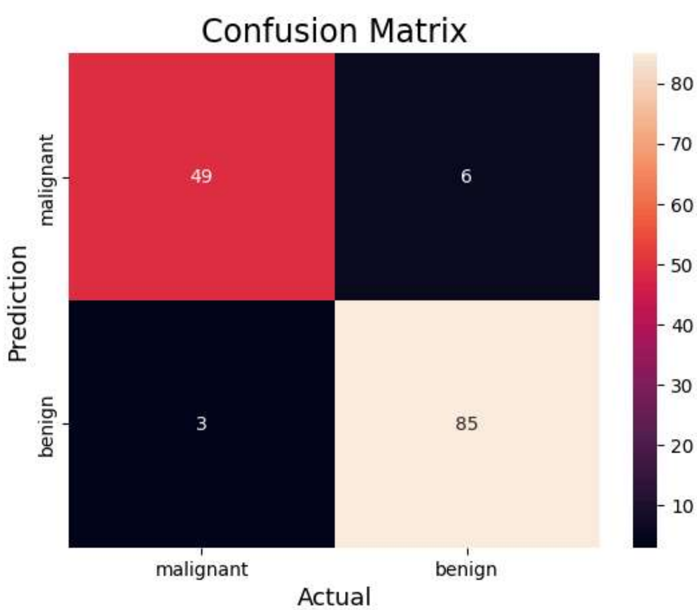
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

def true_fun(X):
    return np.cos(1.5 * np.pi * X)

np.random.seed(0)

n_samples = 30
degrees = [1, 4, 15]

X = np.sort(np.random.rand(n_samples))
y = true_fun(X) + np.random.randn(n_samples) * 0.1

plt.figure(figsize=(14, 5))

for i in range(len(degrees)):
    ax = plt.subplot(1, len(degrees), i + 1)
    plt.setp(ax, xticks=(), yticks=())

    polynomial_features = PolynomialFeatures(degree=degrees[i], include_bias=False)
    linear_regression = LinearRegression()
    pipeline = Pipeline([
        ("polynomial_features", polynomial_features),
        ("linear_regression", linear_regression),
    ])

    pipeline.fit(X[:, np.newaxis], y)

    scores = cross_val_score(
        pipeline, X[:, np.newaxis], y, scoring="neg_mean_squared_error", cv=10
    )

    X_test = np.linspace(0, 1, 100)
    plt.plot(X_test, pipeline.predict(X_test[:, np.newaxis]), label="Model")
    plt.plot(X_test, true_fun(X_test), label="True function")
    plt.scatter(X, y, edgecolor="b", s=20, label="Samples")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.xlim((0, 1))
    plt.ylim((-2, 2))
    plt.legend(loc="best")
    plt.title(
        "Degree {}\nMSE = {:.2e}(+/- {:.2e})".format(
            degrees[i], -scores.mean(), scores.std()
        )
    )

plt.show()
```




```
 #Import the necessary libraries
import numpy as np
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

#Create the NumPy array for actual and predicted labels.
actual    = np.array(
  ['Dog','Dog','Dog','Not Dog','Dog','Not Dog','Dog','Dog','Not Dog','Not Dog'])
predicted = np.array(
  ['Dog','Not Dog','Dog','Not Dog','Dog','Dog','Dog','Dog','Not Dog','Not Dog'])

#compute the confusion matrix.
cm = confusion_matrix(actual,predicted)

#Plot the confusion matrix.
sns.heatmap(cm,
            annot=True,
            fmt='g',
            xticklabels=['Dog','Not Dog'],
            yticklabels=['Dog','Not Dog'])
plt.ylabel('Prediction',fontsize=13)
plt.xlabel('Actual',fontsize=13)
plt.title('Confusion Matrix',fontsize=17)
plt.show()
```



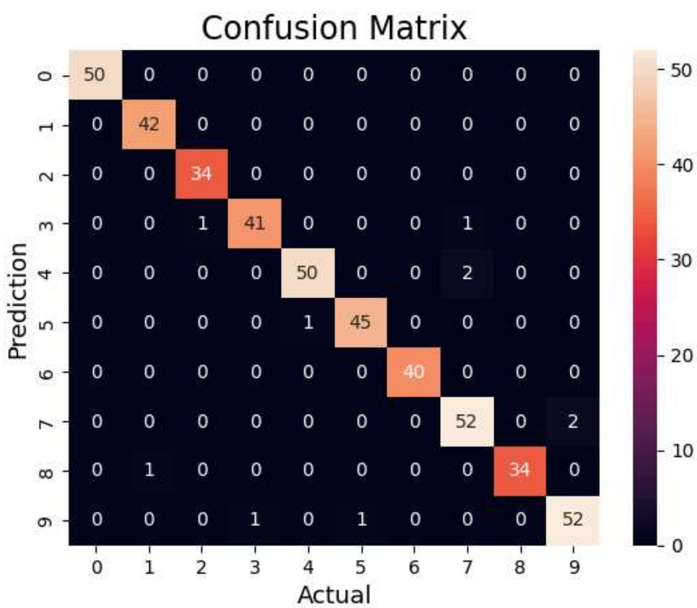
```
#Import the necessary libraries
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Load the breast cancer dataset
X, y= load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.25)

# Train the model
tree = DecisionTreeClassifier(random_state=23)
tree.fit(X_train, y_train)

# preduction
y_pred = tree.predict(X_test)

# compute the confusion matrix
cm = confusion_matrix(y_test,y_pred)
#Plot the confusion matrix.
sns.heatmap(cm,
annot=True,
fmt='g',
xticklabels=['malignant', 'benign'],
yticklabels=['malignant', 'benign'])
plt.ylabel('Prediction',fontsize=13)
plt.xlabel('Actual',fontsize=13)
plt.title('Confusion Matrix',fontsize=17)
plt.show()
# Finding precision and recall
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy   :", accuracy)
precision = precision_score(y_test, y_pred)
print("Precision :", precision)
recall = recall_score(y_test, y_pred)
print("Recall    :", recall)
F1_score = f1_score(y_test, y_pred)
print("F1-score  :", F1_score)
```




```
Accuracy   : 0.9370629370629371
Precision : 0.9340659340659341
Recall    : 0.9659090909090909
```

```
#Import the necessary libraries
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Load the breast cancer dataset
X, y= load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size=0.25)

# Train the model
clf = RandomForestClassifier(random_state=23)
clf.fit(X_train, y_train)

# preduction
y_pred = clf.predict(X_test)

# compute the confusion matrix
cm = confusion_matrix(y_test,y_pred)

#Plot the confusion matrix.
sns.heatmap(cm,
            annot=True,
            fmt='g')
plt.ylabel('Prediction',fontsize=13)
plt.xlabel('Actual',fontsize=13)
plt.title('Confusion Matrix',fontsize=17)
plt.show()


# Finding precision and recall
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy   :", accuracy)
```

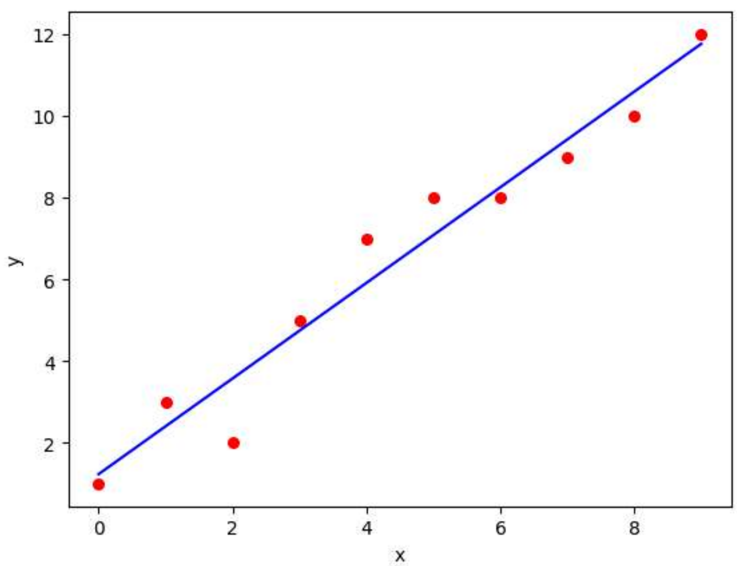



```
Accuracy   : 0.9777777777777777
```

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

def estimate_coef(x, y):
    # number of observations/points
    n = np.size(x)

    # mean of x and y vector
    m_x = np.mean(x)
    m_y = np.mean(y)

    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x

    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
```

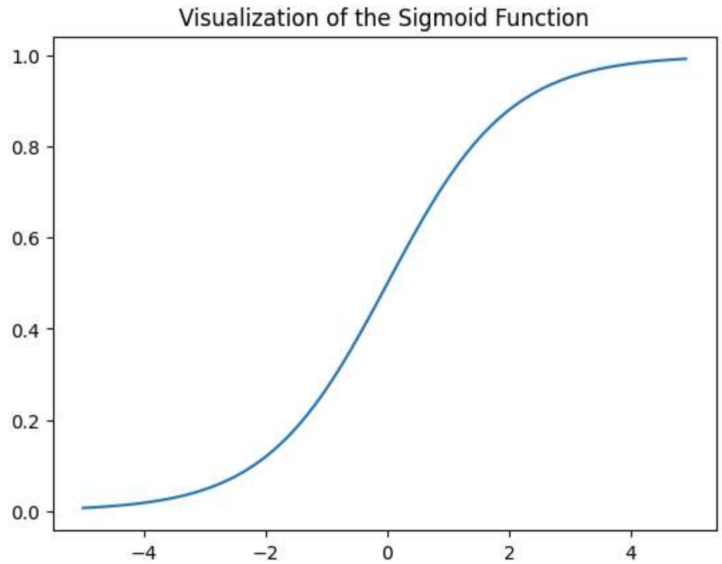
```
    b_0 = m_y - b_1*m_x

    return (b_0, b_1)

def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color = "r",
               marker = "o", s = 30)

    # predicted response vector
    y_pred = b[0] + b[1]*x

    # plotting the regression line
    plt.plot(x, y_pred, color = "b")

    # putting labels
    plt.xlabel('x')
    plt.ylabel('y')

    # function to show plot
    plt.show()

def main():
    # observations / data
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

    # estimating coefficients
    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {}  \  \nb_1 = {}".format(b[0], b[1]))

    # plotting regression line
    plot_regression_line(x, y, b)

if __name__ == "__main__":
  main()
```

```
Estimated coefficients:
b_0 = 1.2363636363636363  \
b_1 = 1.1696969696969697
```



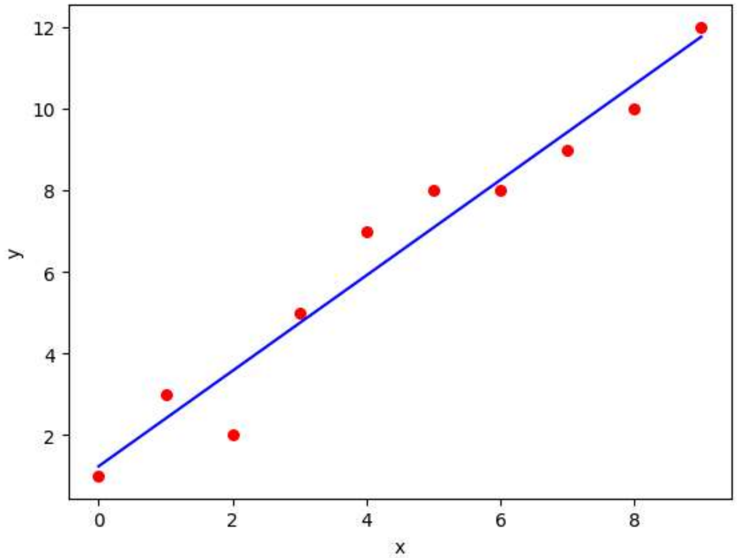
Start coding or generate with AI.

```
import numpy as np
import matplotlib.pyplot as plt
def sigmoid(z):
   return 1 / (1 + np.exp( - z))
plt.plot(np.arange(-5, 5, 0.1), sigmoid(np.arange(-5, 5, 0.1)))
plt.title('Visualization of the Sigmoid Function')
plt.show()
```

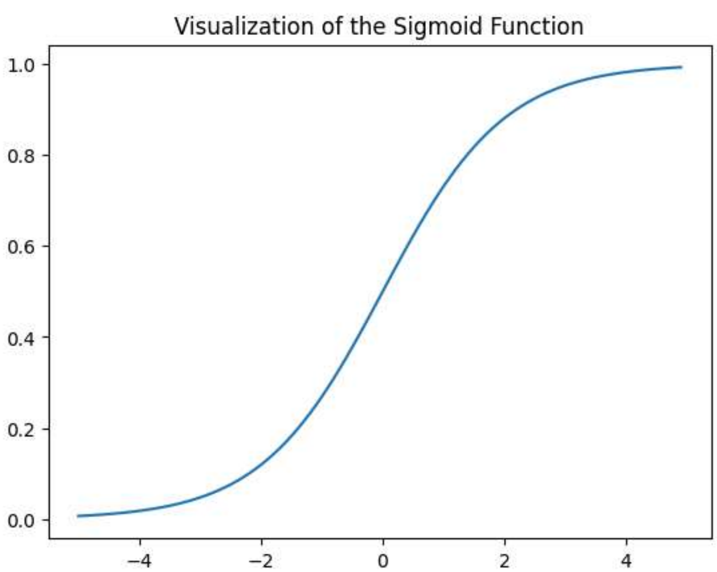



```
import numpy as np
import matplotlib.pyplot as plt

def estimate_coef(x, y):
    n = np.size(x)

    m_x = np.mean(x)
    m_y = np.mean(y)

    SS_xy = np.sum(y*x) - n*m_y*m_x
    SS_xx = np.sum(x*x) - n*m_x*m_x

    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1*m_x

    return (b_0, b_1)

def plot_regression_line(x, y, b):
    plt.scatter(x, y, color = "r",
               marker = "o", s = 30)

    y_pred = b[0] + b[1]*x

    plt.plot(x, y_pred, color = "b")

    plt.xlabel('x')
    plt.ylabel('y')

    plt.show()

def main():
    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

    b = estimate_coef(x, y)
    print("Estimated coefficients:\nb_0 = {}  \ \nb_1 = {}".format(b[0], b[1]))

    plot_regression_line(x, y, b)

if __name__ == "__main__":
  main()
```

```
Estimated coefficients:
b_0 = 1.2363636363636363  \
b_1 = 1.1696969696969697
```




```
import numpy as np
import matplotlib.pyplot as plt
def sigmoid(z):
  return 1 / (1 + np.exp( - z))
plt.plot(np.arange(-5, 5, 0.1), sigmoid(np.arange(-5, 5, 0.1)))
plt.title('Visualization of the Sigmoid Function')
plt.show()
```




```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.datasets import load_iris

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
classifier = KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

print("Confusion Matrix:")
print(cm)
print("\nAccuracy Score:", accuracy)
```

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

Accuracy Score: 1.0
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.datasets import load_iris

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

classifier = GaussianNB()
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

print("Confusion Matrix:")
print(cm)
print("\nAccuracy Score:", accuracy)
```

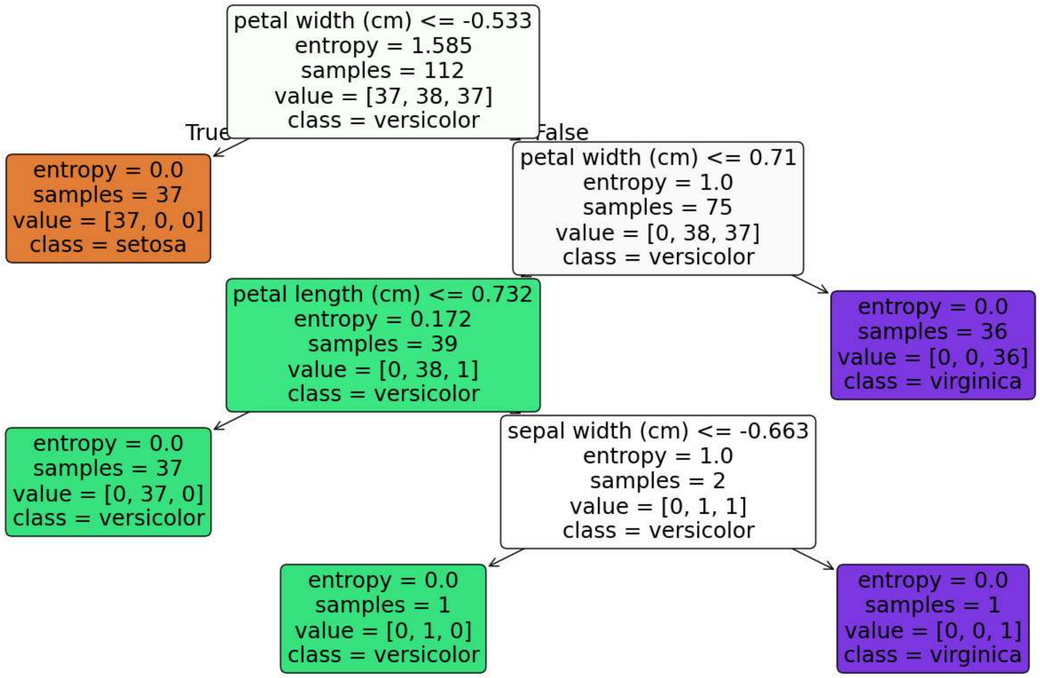
```
Confusion Matrix:
[[13  0  0]
 [ 0 16  0]
 [ 0  0  9]]

Accuracy Score: 1.0
```

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.datasets import load_iris

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

```
classifier = GaussianNB()
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
accuracy = accuracy_score(y_test, y_pred)

print("Confusion Matrix:")
print(cm)
print("\nAccuracy Score:", accuracy)
```

```
Confusion Matrix:
[[13  0  0]
 [ 0 16  0]
 [ 0  0  9]]

Accuracy Score: 1.0
```

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import confusion_matrix, accuracy_score
import matplotlib.pyplot as plt

iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=8)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

classifier = DecisionTreeClassifier(criterion='entropy', random_state=5)
classifier.fit(X_train, y_train)

plt.figure(figsize=(20, 10))
plot_tree(classifier, filled=True, rounded=True, feature_names=feature_names, class_names=iris.target_names)
plt.show()

y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
print("Accuracy Score:", accuracy_score(y_test, y_pred))
```




```
Confusion Matrix:
[[13  0  0]
 [ 0 11  1]
 [ 0  3 10]]
Accuracy Score: 0.8947368421052632
```

```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, accuracy_score

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=32)


sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

classifier = SVC(kernel='linear', random_state=0)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
print('Accuracy: {:.2f}%'.format(accuracy_score(y_test, y_pred) * 100))
```

```
Confusion Matrix:
[[16  0  0]
 [ 0 11  0]
 [ 0  0 11]]
Accuracy: 100.00%
```

```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

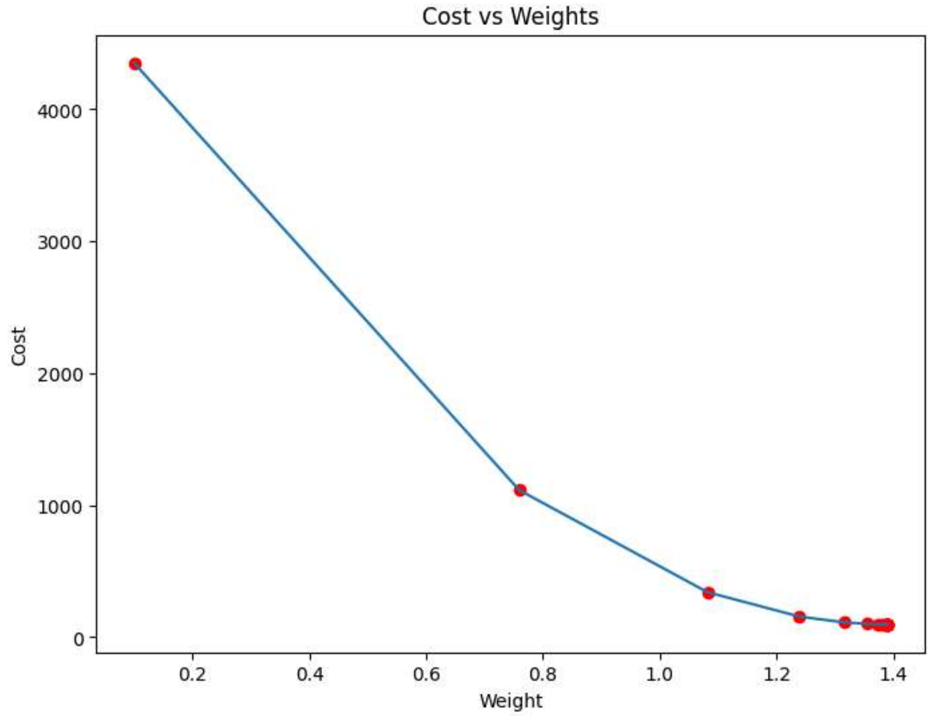
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=39)

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

classifier = RandomForestClassifier(n_estimators=100, random_state=42)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
print('Accuracy:', accuracy_score(y_test, y_pred))
```

```
Confusion Matrix:
[[12  0  0]
 [ 0 13  0]
 [ 0  1 12]]
Accuracy: 0.9736842105263158
```

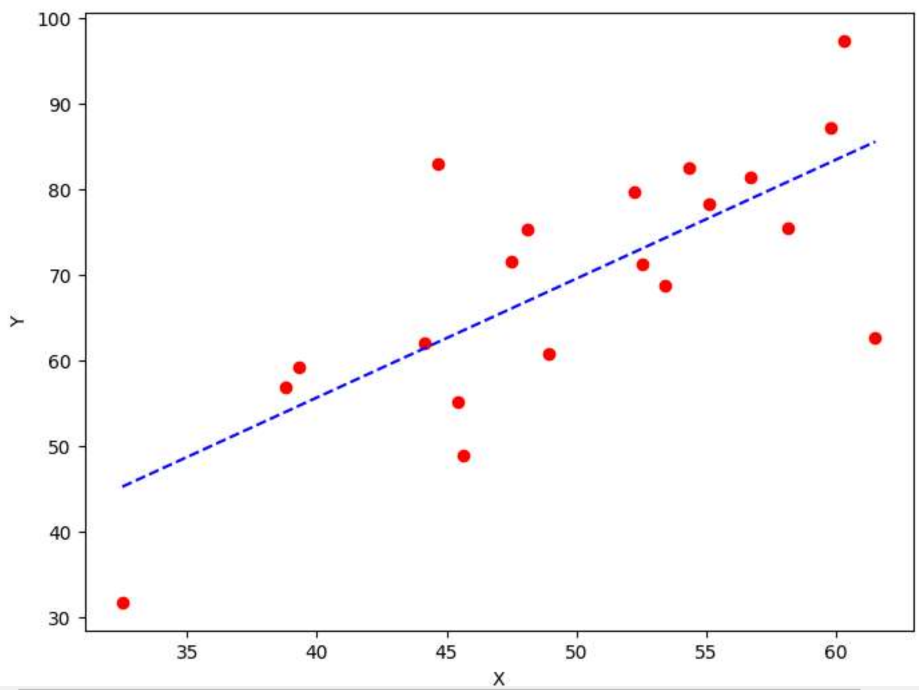
```
 # Importing Libraries
import numpy as np
import matplotlib.pyplot as plt

# Mean Squared Error Function
def mean_squared_error(y_true, y_predicted):
    # Calculating the loss or cost
    cost = np.sum((y_true - y_predicted) ** 2) / len(y_true)
    return cost

# Gradient Descent Function
def gradient_descent(x, y, iterations=1000, learning_rate=0.0001, stopping_threshold=1e-6):
    # Initializing weight, bias, learning rate, and iterations
    current_weight = 0.1
    current_bias = 0.01
    n = float(len(x))

    costs = []
    weights = []
    previous_cost = None

    # Estimation of optimal parameters
    for i in range(iterations):
        # Making predictions
        y_predicted = (current_weight * x) + current_bias

        # Calculating the current cost
        current_cost = mean_squared_error(y, y_predicted)

        # If the change in cost is less than or equal to stopping_threshold, stop the gradient descent
        if previous_cost and abs(previous_cost - current_cost) <= stopping_threshold:
            break

        previous_cost = current_cost

        costs.append(current_cost)
        weights.append(current_weight)

        # Calculating the gradients
        weight_derivative = -(2 / n) * sum(x * (y - y_predicted))
        bias_derivative = -(2 / n) * sum(y - y_predicted)

        # Updating weights and bias
        current_weight = current_weight - (learning_rate * weight_derivative)
        current_bias = current_bias - (learning_rate * bias_derivative)

        # Printing the parameters for every 100th iteration
```

```
        if i % 100 == 0:
            print(f"Iteration {i+1}: Cost {current_cost}, Weight {current_weight}, Bias {current_bias}")

    # Visualizing the weights and cost for all iterations
    plt.figure(figsize=(8, 6))
    plt.plot(weights, costs)
    plt.scatter(weights, costs, marker='o', color='red')
    plt.title("Cost vs Weights")
    plt.ylabel("Cost")
    plt.xlabel("Weight")
    plt.show()

    return current_weight, current_bias

# Main function
def main():
    # Data
    X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963,
                  59.81320787, 55.14218841, 52.21179669, 39.29956669,
                  48.10504169, 52.55001444, 45.41973014, 54.35163488,
                  44.1640495, 58.16847072, 56.72720806, 48.95588857,
                  44.68719623, 60.29732685, 45.61864377, 38.81681754])
    Y = np.array([31.70700585, 68.77759598, 62.5623823, 71.54663223,
                  87.23092513, 78.21151827, 79.64197305, 59.17148932,
                  75.3312423, 71.30087989, 55.16567715, 82.47884676,
                  62.00892325, 75.39287043, 81.43619216, 60.72360244,
                  82.89250373, 97.37989686, 48.84715332, 56.87721319])

    # Estimating weight and bias using gradient descent
    estimated_weight, estimated_bias = gradient_descent(X, Y, iterations=2000)
    print(f"Estimated Weight: {estimated_weight}\nEstimated Bias: {estimated_bias}")

    # Making predictions using estimated parameters
    Y_pred = estimated_weight * X + estimated_bias

    # Plotting the regression line
    plt.figure(figsize=(8, 6))
    plt.scatter(X, Y, marker='o', color='red')
    plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)],
             color='blue', markerfacecolor='red', markersize=10, linestyle='dashed')
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()

if __name__ == "__main__":
    main()
```

```
Iteration 1: Cost 4352.088931274409, Weight 0.7593291142562117, Bias 0.02288558130709
```



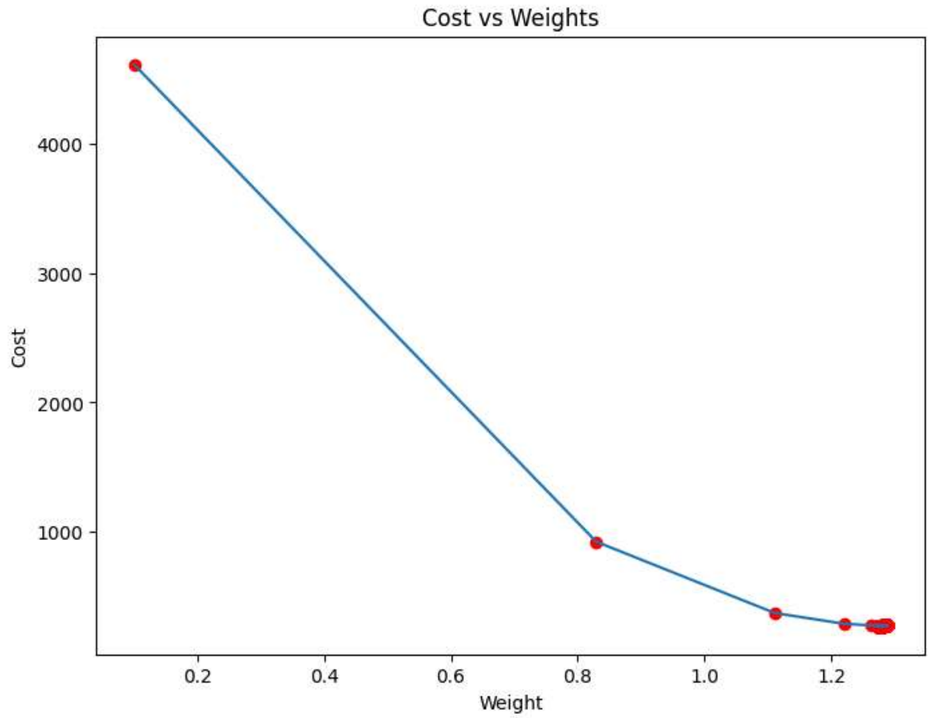

```
Estimated Weight: 1.389738813163012
Estimated Bias: 0.03509461674147458
```



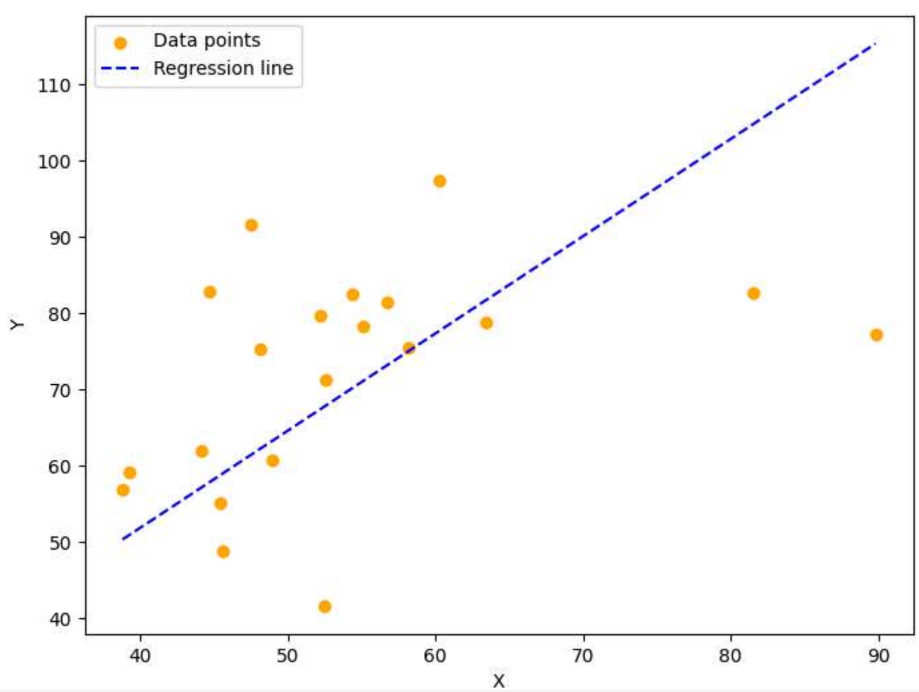

```
# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt

# Function to calculate mean squared error
def mean_squared_error(y_true, y_predicted):
    # Calculating the loss or cost
    cost = np.sum((y_true - y_predicted) ** 2) / len(y_true)
    return cost

# Gradient Descent Function
def gradient_descent(x, y, iterations=1000, learning_rate=0.0001, stopping_threshold=1e-6):
    # Initializing weight, bias, and other parameters
    current_weight = 0.1
    current_bias = 0.01
    n = float(len(x))
```

```
    costs = []
    weights = []
    previous_cost = None

    # Estimation of optimal parameters
    for i in range(iterations):
        # Making predictions
        y_predicted = (current_weight * x) + current_bias

        # Calculating the current cost
        current_cost = mean_squared_error(y, y_predicted)

        # If the change in cost is less than or equal to stopping_threshold, stop gradient descent
        if previous_cost and abs(previous_cost - current_cost) <= stopping_threshold:
            break

        previous_cost = current_cost

        costs.append(current_cost)
        weights.append(current_weight)

        # Calculating the gradients
        weight_derivative = -(2 / n) * np.sum(x * (y - y_predicted))
        bias_derivative = -(2 / n) * np.sum(y - y_predicted)

        # Updating weights and bias
        current_weight -= learning_rate * weight_derivative
        current_bias -= learning_rate * bias_derivative

        # Printing the parameters for each 100th iteration
        if (i + 1) % 100 == 0:
            print(f"Iteration {i + 1}: Cost {current_cost}, Weight {current_weight}, Bias {current_bias}")

    # Visualizing the weights and cost for all iterations
    plt.figure(figsize=(8, 6))
    plt.plot(weights, costs)
    plt.scatter(weights, costs, marker='o', color='red')
    plt.title("Cost vs Weights")
    plt.ylabel("Cost")
    plt.xlabel("Weight")
    plt.show()

    return current_weight, current_bias

def main():
    # Data
    X = np.array([52.50234527, 63.42680403, 81.53035803, 47.47563963,
                  89.81320787, 55.14218841, 52.21179669, 39.29956669,
                  48.10504169, 52.55001444, 45.41973014, 54.35163488,
                  44.1640495, 58.16847072, 56.72720806, 48.95588857,
                  44.68719623, 60.29732685, 45.61864377, 38.81681754])
    Y = np.array([41.70700585, 78.77759598, 82.5623823, 91.54663223,
                  77.23092513, 78.21151827, 79.64197305, 59.17148932,
                  75.3312423, 71.30087989, 55.16567715, 82.47884676,
                  62.00892325, 75.39287043, 81.43619216, 60.72360244,
                  82.89250373, 97.37989686, 48.84715332, 56.87721319])

    # Estimating weight and bias using gradient descent
    estimated_weight, estimated_bias = gradient_descent(X, Y, iterations=2000)
    print(f"Estimated Weight: {estimated_weight}\nEstimated Bias: {estimated_bias}")

    # Making predictions using estimated parameters
    Y_pred = estimated_weight * X + estimated_bias

    # Plotting the regression line
    plt.figure(figsize=(8, 6))
    plt.scatter(X, Y, color='orange', label='Data points')
    plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='blue', linestyle='dashed', label='Regression line')
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.legend()
    plt.show()

if __name__ == "__main__":
    main()
```

```
Iteration 100: Cost 274.7135068175031, Weight 1.2883463473455743, Bias 0.07762035027873679
Iteration 200: Cost 274.49587741697513, Weight 1.2875253901861716, Bias 0.1242635392098511
Iteration 300: Cost 274.27868462255617, Weight 1.2867052569385722, Bias 0.17085991708211085
Iteration 400: Cost 274.0619275583309, Weight 1.2858859467758992, Bias 0.2174095308750488
Iteration 500: Cost 273.84560535014145, Weight 1.285067458872105, Bias 0.2639124275210493
Iteration 600: Cost 273.6297171255836, Weight 1.2842497924019718, Bias 0.3103686539053953
Iteration 700: Cost 273.4142620140034, Weight 1.2834329465411094, Bias 0.35677825686631576
Iteration 800: Cost 273.19923914649354, Weight 1.282616920465955, Bias 0.40314128319503306
Iteration 900: Cost 272.98464765588983, Weight 1.2818017133537727, Bias 0.4494577796358098
Iteration 1000: Cost 272.77048667676803, Weight 1.2809873243826522, Bias 0.49572779288599633
Iteration 1100: Cost 272.5567553454398, Weight 1.2801737527315074, Bias 0.5419513695960768
Iteration 1200: Cost 272.34345279994983, Weight 1.2793609975800773, Bias 0.5881285563697172
Iteration 1300: Cost 272.1305781800719, Weight 1.2785490581089232, Bias 0.6342593997638125
Iteration 1400: Cost 271.91813062730546, Weight 1.2777379334994292, Bias 0.6803439462885345
Iteration 1500: Cost 271.7061092848723, Weight 1.2769276229338014, Bias 0.7263822424073747
Iteration 1600: Cost 271.49451329771335, Weight 1.2761181255950655, Bias 0.772374334537196
Iteration 1700: Cost 271.28334181248476, Weight 1.2753094406670682, Bias 0.8183202690482769
Iteration 1800: Cost 271.07259397755456, Weight 1.274501567334475, Bias 0.864220092264357
Iteration 1900: Cost 270.8622689429993, Weight 1.2736945047827692, Bias 0.9100738504626883
Iteration 2000: Cost 270.65236586060104, Weight 1.272888252198252, Bias 0.9558815898740776
```




```
Estimated Weight: 1.272888252198252
Estimated Bias: 0.9558815898740776
```



```
import numpy as np
import cv2
from matplotlib import pyplot as plt
img = cv2.imread(r"/download (2).png")
b,g,r = cv2.split(img)
rgb_img = cv2.merge([r,g,b])
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
# noise removal
kernel = np.ones((2,2),np.uint8)
#opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 2)
closing = cv2.morphologyEx(thresh,cv2.MORPH_CLOSE,kernel, iterations = 2)
# sure background area
sure_bg = cv2.dilate(closing,kernel,iterations=3)
# Finding sure foreground area
dist_transform = cv2.distanceTransform(sure_bg,cv2.DIST_L2,3)
# Threshold
ret, sure_fg = cv2.threshold(dist_transform,0.1*dist_transform.max(),255,0)
# Finding unknown region
sure_fg = np.uint8(sure_fg)
unknown = cv2.subtract(sure_bg,sure_fg)
# Marker labelling
ret, markers = cv2.connectedComponents(sure_fg)
# Add one to all labels so that sure background is not 0, but 1
markers = markers+1
# Now, mark the region of unknown with zero
markers[unknown==255] = 0
markers = cv2.watershed(img,markers)
img[markers == -1] = [255,0,0]
plt.subplot(211),plt.imshow(rgb_img)
plt.title('Input Image'), plt.xticks([]), plt.yticks([])
plt.subplot(212),plt.imshow(thresh, 'gray')
plt.imsave(r'thresh.png',thresh)
plt.title("Otsu's binary threshold"), plt.xticks([]), plt.yticks([])
plt.tight_layout()
plt.show()
```