Question 2

2.1

Calculate the correlation coefficient.

```
import pandas as pd
import statsmodels.api as sm
import statsmodels.formula.api as smf
import numpy as np
from scipy import stats

# Load the dataset
file_path = 'multi linear regression.xlsx'  #Point to the path where the data is located
data = pd.read_excel(file_path) #Read the data.

# Examine the structure of the data by having a glimpse of first 5 rows by using the **head** function
print(data.head())
```

```
   Hours studying  Preparation exams taken  Mark (100)
0               1                        1          76
1               2                        3          78
2               2                        3          85
3               4                        5          88
4               2                        2          72
```

```
# Calculate the correlation coefficient
correlation_matrix = data.corr()
correlation_coefficient = correlation_matrix.loc['Hours studying', 'Preparation exams taken']
print(f"Correlation Coefficient: {correlation_coefficient:.4f}")
```

```
Correlation Coefficient: 0.2550
```

Interpretation The correlation coefficient ranges from -1 to 1. A value closer to 1 implies a strong positive relationship, closer to -1 implies a strong negative relationship, and around 0 implies no linear relationship.

2.2 Estimate the Regression Line Parameters and Intercept

```
# Fit the multiple linear regression model
X = data[['Hours studying', 'Preparation exams taken']]
y = data['Mark (100)']

X = sm.add_constant(X)  # Adds a constant term to the predictor

model = sm.OLS(y, X).fit()
print(model.summary())
```

```python
# Extract parameters
intercept = model.params['const']
coef_hours = model.params['Hours studying']
coef_exams = model.params['Preparation exams taken']
print(f"Intercept: {intercept:.4f}")
print(f"Coefficient for Hours Studying: {coef_hours:.4f}")
print(f"Coefficient for Preparation Exams: {coef_exams:.4f}")
```

```
                            OLS Regression Results                            

======================================================================
========
Dep. Variable:             Mark (100)   R-squared:                       
0.459
Model:                            OLS   Adj. R-squared:                  
0.429
Method:                 Least Squares   F-statistic:                     
15.67
Date:                Thu, 13 Jun 2024   Prob (F-statistic):            
1.18e-05
Time:                        20:34:58   Log-Likelihood:                
-133.34
No. Observations:                  40   AIC:                             
272.7
Df Residuals:                      37   BIC:                             
277.7
Df Model:                           2                                         

Covariance Type:            nonrobust                                         

======================================================================
=====================
                              coef    std err          t      P>|t|      
[0.025      0.975]
----------------------------------------------------------------------
---------------------
const                      68.5551      3.031     22.614      0.000      
62.413      74.697
Hours studying              3.6378      0.727      5.006      0.000       
2.165       5.110
Preparation exams taken     0.7984      0.697      1.146      0.259      
-0.613       2.210
======================================================================
========
Omnibus:                        3.445   Durbin-Watson:                   
1.210
Prob(Omnibus):                  0.179   Jarque-Bera (JB):                
2.585
Skew:                          -0.617   Prob(JB):                        
```

```
0.275
Kurtosis:                       3.167   Cond. No.
13.7
==============================================================================
========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
Intercept: 68.5551
Coefficient for Hours Studying: 3.6378
Coefficient for Preparation Exams: 0.7984
```

2.3 Calculate and Interpret the Coefficient of Determination

```
# Extract R-squared value
r_squared = model.rsquared
print(f"R-squared: {r_squared:.4f}")

R-squared: 0.4586
```

2.4 Test the Model for Significance Using an F Test

```
# Extract F-statistic and its p-value
f_statistic = model.fvalue
f_p_value = model.f_pvalue
print(f"F-statistic: {f_statistic:.4f}")
print(f"p-value of F-statistic: {f_p_value:.6f}")

F-statistic: 15.6697
p-value of F-statistic: 0.000012
```

Q2.5 Test if the Variable "Hours Studied" is Significant (Q.2.5)

```
# Extract p-value for Hours Studying
p_value_hours = model.pvalues['Hours studying']
print(f"p-value for Hours Studying: {p_value_hours:.6f}")

p-value for Hours Studying: 0.000014
```

Q2.6 Estimate the Mark for a Given Student

```
# Estimate the mark for the given student
hours_studied = 3
preparation_exams = 4
estimated_mark = intercept + coef_hours * hours_studied + coef_exams *
preparation_exams
print(f"Estimated Mark: {estimated_mark:.2f}")
```

```
Estimated Mark: 82.66
```

Q 3.1 Standardize the Data and Calculate Covariance

Notes for the code

import numpy as np: Imports the NumPy library and assigns the alias np for easier reference.

from numpy.linalg import eig: Imports the eig function from the linear algebra module of NumPy. This function computes the eigenvalues and eigenvectors of a square matrix.

from sklearn.preprocessing import StandardScaler: Imports the StandardScaler class from the preprocessing module of scikit-learn. This class is used for standardization, which transforms the data such that its mean is 0 and standard deviation is 1.

features = np.array([[2, 3, 6], [5, 2, 4], [8, 8, 16]]): Defines a NumPy array features representing a 3x3 matrix containing sample data with three features.

print("Mean and Standard Deviation Before Normalizing:"): Prints a message indicating that the mean and standard deviation of the data before normalization will be displayed.

print("The Initial Mean of the data is:"): Prints a message indicating that the initial mean of the data will be displayed.

Meanfeatures = np.mean(features.T, axis=1): Calculates the mean of each feature by transposing the features array and then calculating the mean along the columns (axis 1).

print(Meanfeatures): Prints the calculated mean values of the features.

print("The Initial Standard Deviation of the data is:"): Prints a message indicating that the initial standard deviation of the data will be displayed.

Sdfeatures = np.std(features.T, axis=1): Calculates the standard deviation of each feature by transposing the features array and then calculating the standard deviation along the columns (axis 1).

print(Sdfeatures): Prints the calculated standard deviation values of the features.

print("Normalizing the data:"): Prints a message indicating that the data will be normalized.

stdfeatures = StandardScaler().fit_transform(features): Normalizes the data using StandardScaler and stores the normalized data in the variable stdfeatures.

print(stdfeatures): Prints the normalized data.

print("The Mean of the normalized data is:"): Prints a message indicating that the mean of the normalized data will be displayed.

Meanstdfeatures = np.mean(stdfeatures.T, axis=1): Calculates the mean of each feature in the normalized data by transposing the stdfeatures array and then calculating the mean along the columns (axis 1).

print(Meanstdfeatures): Prints the calculated mean values of the normalized features.

print("The Standard Deviation of the normalized data is:"): Prints a message indicating that the standard deviation of the normalized data will be displayed.

Sdstdfeatures = np.std(stdfeatures.T, axis=1): Calculates the standard deviation of each feature in the normalized data by transposing the stdfeatures array and then calculating the standard deviation along the columns (axis 1).

print(Sdstdfeatures): Prints the calculated standard deviation values of the normalized features.

print("The Covariance Matrix of the normalized data is:"): Prints a message indicating that the covariance matrix of the normalized data will be displayed.

cov_matrix = np.cov(stdfeatures.T): Computes the covariance matrix of the normalized data by transposing the stdfeatures array and then calculating the covariance matrix.

print(cov_matrix): Prints the calculated covariance matrix of the normalized data.

eigenvalues, _ = eig(cov_matrix): Calculates the eigenvalues of the covariance matrix using the eig function. The _ is used to discard the second return value, which corresponds to the eigenvectors.

print("The Eigen Values of the normalized data are:"): Prints a message indicating that the eigenvalues of the normalized data will be displayed.

print(eigenvalues): Prints the calculated eigenvalues of the normalized data.

print("The Eigen Vectors of the normalized data are:"): Prints a message indicating that the eigenvectors of the normalized data will be displayed.

eigenvalues_sort = np.sort(eigenvalues)[::-1]: Sorts the eigenvalues in descending order.

eigenvectors = np.array(...).T: Transposes the reordered eigenvectors and converts them into a NumPy array.

print(eigenvectors): Prints the calculated eigenvectors of the normalized data.

print("The principal components of the normalized data are:"): Prints a message indicating that the principal components of the normalized data will be displayed.

pca_output = eigenvectors.T.dot(stdfeatures.T): Calculates the principal components using the dot product of the transposed eigenvectors and the transposed normalized data.

print(pca_output): Prints the calculated principal components of the normalized data.

```
import numpy as np
import pandas as pd

# Original data
data = np.array([
    [2, 3, 6],
    [5, 2, 4],
    [8, 8, 16]
])
```

```
# Standardize the data
mean = np.mean(data, axis=0)
std_dev = np.std(data, axis=0)
standardized_data = (data - mean) / std_dev

# Calculate the covariance matrix
cov_matrix = np.cov(standardized_data.T)



# Print results
print("Mean:\n", mean)
print("Standard Deviation:\n", std_dev)
print("Standardized Data:\n", standardized_data)
print("Covariance Matrix:\n", cov_matrix)
```

```
Mean:
 [5.         4.33333333 8.66666667]
Standard Deviation:
 [2.44948974 2.62466929 5.24933858]
Standardized Data:
 [[-1.22474487 -0.50800051 -0.50800051]
 [ 0.         -0.88900089 -0.88900089]
 [ 1.22474487  1.3970014   1.3970014 ]]
Covariance Matrix:
 [[1.5        1.16657066 1.16657066]
 [1.16657066 1.5        1.5       ]
 [1.16657066 1.5        1.5       ]]
```

3.2 Calculate the eigenvalues that result from the covariance matrix calculated in Q.3.1.

```
# Calculate the eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
print("Eigenvalues:\n", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

```
Eigenvalues:
 [ 4.06225666e+00  4.37743342e-01 -1.43600073e-32]
Eigenvectors:
 [[ 5.41364634e-01  8.40787924e-01  1.88233218e-19]
 [ 5.94526843e-01 -3.82802604e-01 -7.07106781e-01]
 [ 5.94526843e-01 -3.82802604e-01  7.07106781e-01]]
```

Q3.4 Calculate and Interpret Principal Components We will project the standardized data onto the eigenvector matrix to get the principal components.

```
# Calculate the principal components
principal_components = np.dot(standardized_data, eigenvectors)
print("\nPrincipal Components:\n", principal_components)


Principal Components:
 [[-1.26707344e+00 -6.40822863e-01 -5.07802097e-17]
 [-1.05706978e+00  6.80623710e-01 -9.79810508e-17]
 [ 2.32414322e+00 -3.98008471e-02  1.39237053e-16]]
```

Q4.1 Create a Decision Tree

Loading the Data

```
import pandas as pd

df = pd.read_excel('decision trees.xlsx')
df

   Loves ice cream Loves chocolate  Age Loves the movie "Ice age"
Unnamed: 4
0               yes             yes    7                        no
NaN
1                no             yes    9                       yes
8.0
2               yes              no   12                        no
10.5
3               yes              no   14                        no
13.0
4                no             yes   18                       yes
16.0
5               yes             yes   18                       yes
18.0
6                no             yes   34                       yes
26.0
7                no             yes   35                       yes
34.5
8                no             yes   37                       yes
36.0
9               yes             yes   38                       yes
37.5
10              yes              no   48                        no
43.0
11              yes              no   50                        no
49.0
12              yes              no   77                        no
63.5
13               no              no   83                        no
80.0
```

Preprocess the data: Convert categorical variables into numerical ones. For example, if the preference for ice cream and chocolate is given as 'yes' or 'no', you could convert these to 1 and 0 respectively.

```
df['ice_cream'] = df['Loves ice cream'].map({'yes': 1, 'no': 0})
df['chocolate'] = df['Loves chocolate'].map({'yes': 1, 'no': 0})
```

Split the data: Separate the features (age, ice cream preference, chocolate preference) from the target variable (whether they will love "Ice Age" or not).

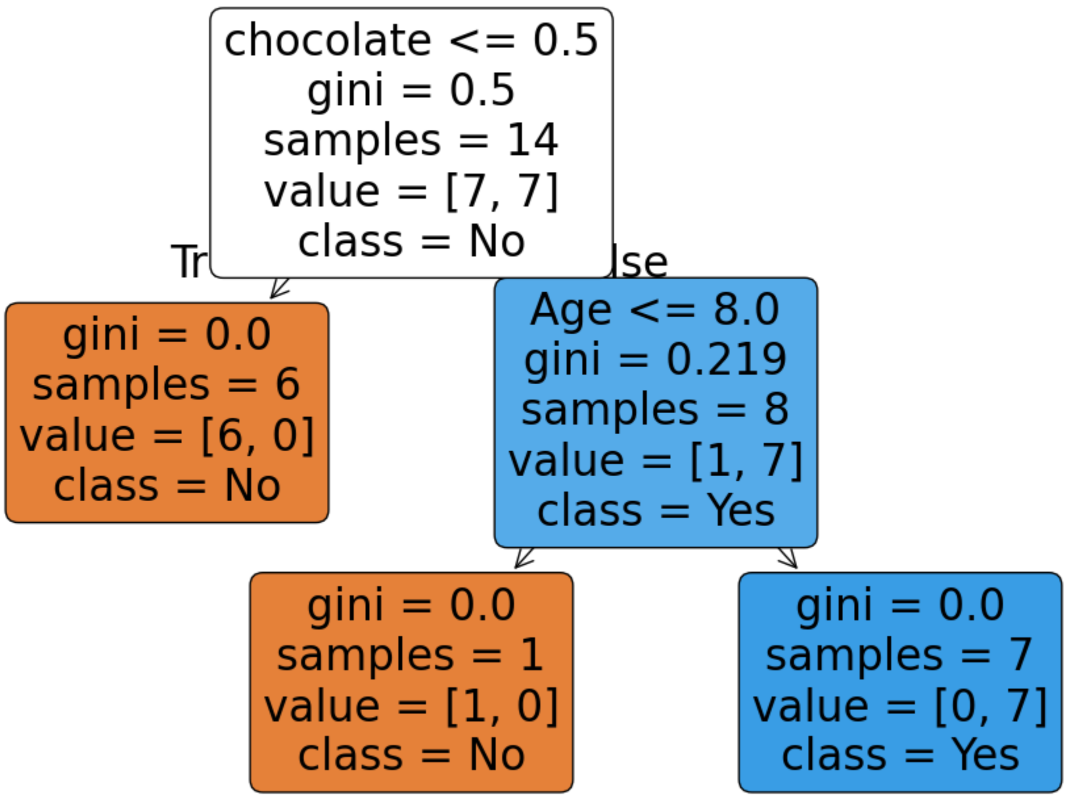
```
X = df[['Age', 'ice_cream', 'chocolate']]
y = df['Loves the movie "Ice age"']
```

Create the decision tree: Use scikit-learn to create and train the decision tree.

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

# Create the decision tree
clf = DecisionTreeClassifier()
clf.fit(X, y)

# Plot the decision tree
plt.figure(figsize=(12, 8))
plot_tree(clf, filled=True, feature_names=X.columns,
class_names=['No', 'Yes'], rounded=True)
plt.show()
```



Extra Code

```
import pandas as pd

# Load the data
file_path = 'decision trees.xlsx'
df = pd.read_excel(file_path)

print("Data:\n", df)

Data:
    Loves ice cream Loves chocolate  Age Loves the movie "Ice age"
Unnamed: 4
0             yes             yes    7                          no
NaN
1              no             yes    9                         yes
8.0
2             yes              no   12                          no
10.5
3             yes              no   14                          no
13.0
4              no             yes   18                         yes
16.0
5             yes             yes   18                         yes
18.0
```

```
6                  no              yes   34                          yes
26.0
7                  no              yes   35                          yes
34.5
8                  no              yes   37                          yes
36.0
9                 yes              yes   38                          yes
37.5
10                yes               no   48                           no
43.0
11                yes               no   50                           no
49.0
12                yes               no   77                           no
63.5
13                 no               no   83                           no
80.0
```

```python
import pandas as pd
import numpy as np

# Assume the data is already loaded in df



# Rename the columns
df.rename(columns={
    'Age': 'Age',
    'Loves ice cream': 'Likes Ice Cream',
    'Loves chocolate': 'Likes Chocolate',
    'Loves the movie "Ice age"': 'Loves Ice Age'
}, inplace=True)

print("Renamed DataFrame:\n", df)

# Helper function to calculate Gini impurity
def gini_impurity(y):
    unique_classes, counts = np.unique(y, return_counts=True)
    probabilities = counts / counts.sum()
    return 1 - np.sum(probabilities**2)

# Helper function to calculate weighted Gini impurity of a split
def weighted_gini_impurity(left_y, right_y):
    left_impurity = gini_impurity(left_y)
    right_impurity = gini_impurity(right_y)
    total_count = len(left_y) + len(right_y)
    weighted_impurity = (len(left_y) / total_count) * left_impurity +
(len(right_y) / total_count) * right_impurity
    return weighted_impurity

# Find the best split for a feature
```

```
def find_best_split(df, feature, target):
    best_gini = float('inf')
    best_split_value = None
    unique_values = df[feature].unique()

    for value in unique_values:
        left_split = df[df[feature] <= value]
        right_split = df[df[feature] > value]
        gini = weighted_gini_impurity(left_split[target],
right_split[target])

        if gini < best_gini:
            best_gini = gini
            best_split_value = value

    return best_split_value, best_gini

# Function to build the decision tree
def build_tree(df, target, features, depth=0, max_depth=3):
    if len(np.unique(df[target])) == 1 or depth == max_depth:
        leaf_value = df[target].mode()[0]
        return leaf_value

    best_feature = None
    best_split_value = None
    best_gini = float('inf')

    for feature in features:
        split_value, gini = find_best_split(df, feature, target)
        if gini < best_gini:
            best_gini = gini
            best_feature = feature
            best_split_value = split_value

    if best_feature is None:
        return df[target].mode()[0]

    left_split = df[df[best_feature] <= best_split_value]
    right_split = df[df[best_feature] > best_split_value]

    tree = {'feature': best_feature, 'split_value': best_split_value,
'gini': best_gini}
    tree['left'] = build_tree(left_split, target, features, depth+1,
max_depth)
    tree['right'] = build_tree(right_split, target, features, depth+1,
max_depth)

    return tree
```

```
# Define features and target
features = ['Age', 'Likes Ice Cream', 'Likes Chocolate']
target = 'Loves Ice Age'

# Build the decision tree
decision_tree = build_tree(df, target, features, max_depth=3)
print("Decision Tree:\n", decision_tree)
```

```
Renamed DataFrame:
    Likes Ice Cream Likes Chocolate  Age Loves Ice Age  Unnamed: 4
0              yes             yes    7            no         NaN
1               no             yes    9           yes         8.0
2              yes              no   12            no        10.5
3              yes              no   14            no        13.0
4               no             yes   18           yes        16.0
5              yes             yes   18           yes        18.0
6               no             yes   34           yes        26.0
7               no             yes   35           yes        34.5
8               no             yes   37           yes        36.0
9              yes             yes   38           yes        37.5
10             yes              no   48            no        43.0
11             yes              no   50            no        49.0
12             yes              no   77            no        63.5
13              no              no   83            no        80.0
Decision Tree:
 {'feature': 'Likes Chocolate', 'split_value': 'no', 'gini': 0.125,
'left': 'no', 'right': {'feature': 'Age', 'split_value': 7, 'gini':
0.0, 'left': 'no', 'right': 'yes'}}
```

Q.4.2 Predict whether a 45-year-old individual who likes chocolate and does not like ice cream will like "Ice Age" or not. Make predictions: Use the trained model to predict whether a 45-year-old individual who likes chocolate and does not like ice cream will like "Ice Age" or not.

```
prediction = clf.predict([[45, 0, 1]])
print("The result of the prediction is the 45 year likes Ice Age or
Not is ", prediction)
```

```
The result of the prediction is the 45 year likes Ice Age or Not is
['yes']

C:\ProgramData\anaconda3\Lib\site-packages\sklearn\base.py:493:
UserWarning: X does not have valid feature names, but
DecisionTreeClassifier was fitted with feature names
  warnings.warn(
```

Q5.1 Using K-fold Cross-Validation to Compare Logistic Regression and KNN

```
import pandas as pd
```

```python
# Load the dataset
file_path = 'cross validation.xlsx'
data = pd.read_excel(file_path)

print("Data:\n", data.head())
```

```
Data:
    Machines working  Age( month)  Average number of shift/ week
0                 1           70                              5
1                 0           59                              4
2                 1           68                              4
3                 0           50                              6
4                 0           40                              7
```

```python
# Extract features and target
X = data[['Age( month)', 'Average number of shift/ week']]
y = data['Machines working']
```

Performing K-fold Cross-Validation We will use 5-fold cross-validation to evaluate the performance of logistic regression and KNN models.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score

# Define the models
logreg = LogisticRegression()
knn = KNeighborsClassifier()

# Perform K-fold cross-validation
logreg_scores = cross_val_score(logreg, X, y, cv=5)
knn_scores = cross_val_score(knn, X, y, cv=5)

# Print the average accuracy of each model
print("Logistic Regression Accuracy: ", logreg_scores.mean())
print("KNN Accuracy: ", knn_scores.mean())
```

```
Logistic Regression Accuracy:  0.43333333333333333
KNN Accuracy:  0.43333333333333333
```

```python
if logreg_scores.mean() > knn_scores.mean():
    better_model = 'Logistic Regression'
else:
    better_model = 'KNN'

print(f"The better model based on cross-validation is:
{better_model}")
```

```
The better model based on cross-validation is: KNN
```

```
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Define the models or instantiate the model functions
logistic_model = LogisticRegression()
knn_model = KNeighborsClassifier()

# Create the K-fold cross-validator
kf = KFold(n_splits=5, shuffle=True, random_state=1)

# Perform cross-validation for Logistic Regression
logistic_scores = cross_val_score(logistic_model, X_scaled, y, cv=kf,
scoring='accuracy')
print("Logistic Regression Cross-Validation Accuracy Scores:",
logistic_scores)
print("Logistic Regression Mean Accuracy:", logistic_scores.mean())

# Perform cross-validation for KNN
knn_scores = cross_val_score(knn_model, X_scaled, y, cv=kf,
scoring='accuracy')
print("KNN Cross-Validation Accuracy Scores:", knn_scores)
print("KNN Mean Accuracy:", knn_scores.mean())

Logistic Regression Cross-Validation Accuracy Scores: [0.33333333
0.16666667 0.33333333 0.66666667 0.33333333]
Logistic Regression Mean Accuracy: 0.36666666666666664
KNN Cross-Validation Accuracy Scores: [0.33333333 0.33333333 0.5
0.5        0.16666667]
KNN Mean Accuracy: 0.36666666666666664
```

Selecting the Better Model Compare the mean accuracy of both models to select the better one.

Q.5.2 Predicting the Working Status

```
# Test Data
new_sample = pd.DataFrame({'Age( month)': [80], 'Average number of
shift/ week': [7]})
new_sample_scaled = scaler.transform(new_sample)

if better_model == 'Logistic Regression':
    logistic_model.fit(X_scaled, y)
    prediction = logistic_model.predict(new_sample_scaled)
else:
    knn_model.fit(X_scaled, y)
```

```
    prediction = knn_model.predict(new_sample_scaled)

print(f"Prediction for the new sample: {'Working' if prediction[0] ==
1 else 'Not Working'}")

Prediction for the new sample: Not Working
```