A picture containing logo

Description automatically generated

**ANL252**

**Python for Data Analytics:**

**End-Of-Course Assignment**

**JANUARY 2023 Presentation**

|  |  |  |
| --- | --- | --- |
| **Name** | **PI Number** | **Submission Date** |
| Teh Zi Jing | M2211393 | 6 March 2023, 1200Hrs |

**Question 1)**

|  |  |
| --- | --- |
| In [1]: | import pandas as pd  #Defining eca\_file to the .csv file filled with the ECA data  #r is to prevent python from reading \ as an escape character  eca\_file = r"C:\Users\zijin\OneDrive\Desktop\ANL252\ECA.csv"  #Create a list that determine those missing values as blank, Unkn, and ???  missing\_values = ['', 'Unkn', '???']  #As blanks, Unkn and ??? are not considered part of python's NA values, we have to include it in using the na\_values function  df = pd.read\_csv(eca\_file, na\_values=missing\_values)  #Use the isna() function to change all values to true or false, true if value is missing and false if it is not missing  #Apply .any() function on axis=0 to see which column has a missing value (which in this case becomes true)  #Use df.columns to return the variables of columns that contains missing values  #Change the index to just a list of variable names that contains missing values  variables\_missing\_values = df.columns[df.isna().any(axis=0)].tolist()  #Print the variables with missing values  print(variables\_missing\_values) |
| Out [1]: |  |

The variables that contains missing values are: ‘Claim\_ID’, ‘Actual’ and ‘Terms’.

**Question 2)**

(Code is a continuation from question 1)

|  |  |
| --- | --- |
| In [2]: | #for mode function later  import statistics  #Use the isna() function count the number of missing or NA values in each columns  #If there is a missing or NA value, it will show True and will show False if the value is not missing  #The sum() function will treat True as 1 and False as 0 and sum up the total number of missing values in all the columns  print(df.isna().sum()) |
| Out [2]: |  |
| In [3]: | #Use the dropna() function to remove rows with missing values/NA values in "Claim\_ID" column  #Subset="Claim\_ID" because these are the columns we want to remove rows with missing value from  #Use inplace=True to modify the original dataframe "df" instead of creating a new one  df.dropna(subset=['Claim\_ID'], inplace=True)  #Use of .mode() function to find the highest frequency term  #Include [0] because we only want 1 highest frequency term  #Fill missing values in "Terms" column with the mode of the column  #Use inplace=True to modify the original dataframe "df" instead of creating a new one  terms\_mode = df['Terms'].mode()[0]  df['Terms'].fillna(terms\_mode, inplace=True)  #Use the .fillna() function to fill missing values in "Actual" column with corresponding values from the "Planned" column  #Use inplace=True to modify the original dataframe "df" instead of creating a new one  df['Actual'].fillna(df['Planned'], inplace=True)  #Use the isna() function to check if there are any missing values left in the dataframe  #If there is a missing value, it will show True and will show False if the value is not missing  #The sum() function will treat True as 1 and False as 0 and sum up the total number of missing values in all the columns  print(df.isna().sum()) |
| Out [3]: |  |
| In [4]: | #Use df.tail() function to see the last 10 data of df to ensure that the missing data in "Actual" is the same as "Planned"  df.tail(10) |
| Out [4]: |  |

**Explanation:**

Firstly, I check how much missing data is there in the 3 columns “Claim\_ID”, “Actual”, and “Terms”. From output [2], it can be seen that “Claim\_ID”, “Actual”, and “Terms” have 5, 1677, and 7 missing data respectively. The whole data contains around 24000 data, and Claim\_ID contains random unique identifiers. Since the Claim\_ID is randomly created, it is better to remove the 5 missing data as it is only a small portion of the overall data which will not have a major impact on the overall data.

As for the “Terms” column, there are a fixed set of terms, and this allows me to find the mode of the term. I used the mode method to fill up those missing values in the “Terms” column to maintain the same sample size and lessen any bias. I used mode because we are dealing with categorical data sets instead of numerical data.

Lastly, for the “Actual” column, I assume that the dates will go as planned and have replaced those empty values with the planned dates in the “Planned” column. In the end, I used .isna() function to ensure that there is no more missing values and .tail(10) method to check the last 10 data to ensure that those blanks in the “Actual” column have been replaced with values from the corresponding “Planned” column dates.

**Question 3)**

(Code is a continuation from questions 1 & 2)

|  |  |
| --- | --- |
| In [5]: | #Check the number of duplicates  print(df.duplicated(subset=['Claim\_ID']).sum())  #Since Claim\_ID is suppose to be unique, there should not have any duplicates  #Drop any duplicates from Claim\_ID  df.drop\_duplicates(subset=['Claim\_ID'], inplace=True)  #Ensure that there are no more duplicates  print(df.duplicated(subset=['Claim\_ID']).sum()) |
| Out [5]: |  |
| In [6]: | #Ensure all the dates of 'Planned', 'Actual' and 'Created' are in the same format  #Use the pd.to\_datetime() function to ensure that both 'Actual' and 'Planned' are stored as datetime format  #Use .dt.strftime() function to change both columns of 'Actual' and 'Planned' to the specific day month year format  df['Actual'] = pd.to\_datetime(df['Actual']).dt.strftime('%d/%m/%Y')  df['Planned'] = pd.to\_datetime(df['Planned']).dt.strftime('%d/%m/%Y')  #Change 'Created' column to datetime format of year, month and day as shown from excel  df['Created'] = pd.to\_datetime(df['Created'], format='%Y%m%d')  #Change 'Created' column to the specific day month year format  df['Created'] = df['Created'].dt.strftime('%d/%m/%Y')  #Convert everything back to datetime format instead of string format  df['Actual'] = pd.to\_datetime(df['Actual'], format='%d/%m/%Y')  df['Planned'] = pd.to\_datetime(df['Planned'], format='%d/%m/%Y')  df['Created'] = pd.to\_datetime(df['Created'], format='%d/%m/%Y')  #Print df to check if the dates are in correct order  df |
| Out [6]: |  |
| In [7]: | #Check what type is the Column 'Amount'  print(df['Amount'].dtype) |
| Out [7]: |  |
| In [8]: | #Change 'Amount' column to numeric type since it is originally an object  #Use the pd.to\_numeric() function to change 'Amount' column into numbers which can be integer or float  #errors='coerce' turns any values in the 'Amount' column into NA values in the event it cannot be changed to numbers  df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')  #Remove any NA values using dropna() function  df.dropna(subset=['Amount'], inplace=True)  #Round all values in 'Amount' column to 2 decimal points  df['Amount'] = df['Amount'].round(2)  #Check if 'Amount' column has been changed to 2 decimal points  df |
| Out [8]: |  |

**Explanation:**

**First data preparation:** Each ‘Claim\_ID” should have a unique identifier number. There should not be any duplicates. Therefore, I look for duplicated numbers using the duplicated() function and the sum() function to see how many duplicates there are if any. It can be seen from Output [5] that there were 3 duplicates initially, so I used the .drop\_duplicates() function to drop any duplicates in the “Claim\_ID” column and replace the column using the inplace=True function. Lastly, I use the duplicated() function again to ensure no more duplicates exist.

**Second data preparation: Initially, the date format of “Planned”, “Actual”, and “Created” were differently formatted. Since “Planned” and “Actual” are formatted in day, month, and year, I used the pd.to\_datetime() and dt.strftime() functions to place both in the format of just day, month, and year. For the “Created” column, the date format was initially in the year, month, and day format. I had to let python know this through pd.to\_datetime() function before changing it into day, month, and year. I converted everything back to datetime format again since it is in string format now. Finally, I printed the data to ensure that all 3 columns have the same date format.**

**Third data preparation: I noticed that the numbers in the “Amount” column are in different decimal points. Therefore, I used the dtype() function to check the data type and from the output [7] we can see that it is an object type. Since it is an amount and should be in numeric values, I used pd.to\_numeric() function to change it into numbers which can be an integer or a float. The errors=’coerce’ help to convert any object in the “Amount” column that cannot be changed into numbers to NA value. I dropped the NA values using dropna() function. Since monetary values are usually in 2 decimal points, I converted all the amounts to 2 decimal points using .round(2) function. Lastly, I printed the data to ensure all amounts are in 2 decimal points.**

**Question 4)**

(Code is a continuation from question 3)

**First data & insight:**

|  |  |
| --- | --- |
| In [9]: | import matplotlib.pyplot as plt  #Find the lower quartile value (q1)  q1 = df["Amount"].quantile(q=0.25)  #Find the upper quartile value (q3)  q3 = df["Amount"].quantile(q=0.75)  #Find the interquartile range (iqr)  iqr = q3-q1  #Find the lower bound value, and since an insurance claim cannot be less than $0, use the if function to restrict the lower bound value to $0  lower\_bound = q1 - 1.5\*iqr  if lower\_bound < 0:  lower\_bound = 0  #Find the upper bound value  upper\_bound = q3 + 1.5\*iqr  #Remove all outlier data and only keep those that are within lower bound of 0 and upper bound  boxplot\_data = df[(df["Amount"] >= lower\_bound) & (df["Amount"] <= upper\_bound)]  #Create a boxplot diagram  boxplot\_data.boxplot(column=["Amount"])  #Add a text showing the lower bound value on the graph and round it off to 2 decimal points  plt.text(1.05, lower\_bound, f"Lower Bound: {lower\_bound:.2f}", fontsize=10)  #Add a text showing the upper bound value on the graph and round it off to 2 decimal points  plt.text(1.05, upper\_bound, f"Upper Bound: {upper\_bound:.2f}", fontsize=10)  #Add a text showing the q1 value on the graph and round it off to 2 decimal points  plt.text(0.75, q1, f"Q1: {q1:.2f}", fontsize=10)  #Add a text showing the q3 value on the graph and round it off to 2 decimal points  plt.text(0.75, q3, f"Q3: {q3:.2f}", fontsize=10)  #Find the median value using .median() function and round it off to 2 decimal points  #Plot the value on the chart  plt.text(1.1, boxplot\_data["Amount"].median(), f"Median: {boxplot\_data['Amount'].median():.2f}", fontsize=10)  #Plot the chart title  plt.title('Box Plot Diagram of Insurance Claim Amount')  #Plot the y axis label  plt.ylabel('Amount paid')  #Display the boxplot  plt.show() |
| Out [9]: |  |
| In [10]: | #Check the number of rows left to see how many data has been removed (outliers)  boxplot\_data |
| Out [10]: |  |

**First analysis of data and insight:**

Initially, the data consisted of 24204 data sets, as shown in the output [8] of question 3, and the claim amount ranges from $55.89 to $150,723.86. To analyse the data, I have created a box plot diagram to show the lower bound, lower quartile (q1), median, upper quartile(q3), and upper bound. A box plot shows the data distribution, with an interquartile range showing where most data are. The whiskers which extend from the box are the lower and upper bounds showing all the data excluding outliers. Those data outside the whiskers are considered outliers.

The median showed an amount of $1679.76 in the diagram, which tells us that 50% of the insurance claims are below $1679.76 while another 50% are above $1679.76. Since the median ($1679.76) is closer to quartile 1 ($525.02), this tells us that there is a larger number of data with lower values than higher values. This suggests that the data is skewed to the right. To calculate the interquartile range (iqr), I took the difference between the upper quartile (q3) and the lower quartile (q1). With the interquartile range, I know that the box range of the box plot ($4803.76 to $525.02) is where 50% of the insurance amounts are in.

Using the formula q1 – 1.5 \* iqr, I have found that the lower bound is a negative number, which is impossible for claims to be negative. Therefore, I have changed it to $0. To find the upper bound, I used q3 + 1.5\*iqr, which is $11221.86. An outlier is data points that are very different from the rest due to more severe claims, which can affect the data analysis. Therefore, anything outside $0 and $11221.86 is considered an outlier and removed. With the outliers removed, we are left with 22962 data.

**Second data & insight:**

(Code is a continuation of question 4’s first data & insight)

|  |  |
| --- | --- |
| In [11]: | #Both libraries has been imported previously  #Create a pivot table with name claims\_paid using the pd.pivot\_table() function  #Choose index as 'Paid' column because I want to know if the claims has been paid or not  #Use 'Claim\_ID' as the values because I want to count the number of claims  #Lastly use aggfunc to count the values  claims\_paid = pd.pivot\_table(df, index='Paid', values='Claim\_ID', aggfunc='count')  #Create a variable total\_data as the total number of claims both paid and unpaid  total\_data = claims\_paid.loc['Yes', 'Claim\_ID'] + claims\_paid.loc['No', 'Claim\_ID']  #Add a new column for the percentage of "Paid" claims to show the percentage paid  claims\_paid['% Paid'] = claims\_paid['Claim\_ID'] / total\_data \* 100  #Create a pie chart from the pivot table 'claims\_paid' using plt.pie function  #use labels as claims\_paid.index to show the 'Paid' column showing Yes or No labels to each slice of the pie chart  #Use autopct ='%1.1f%%' to show the percentage of the chart with only 1 decimal point  #Use startangle=90 to let the chart start at 12 o clock position  #Slightly separate the chart with the explode function for users to see the chart clearer  plt.pie(claims\_paid['Claim\_ID'], labels=claims\_paid.index, autopct='%1.1f%%', startangle=90, explode=[0.1, 0.1], shadow=True)  #Add chart title  plt.title('Total Claims Paid and Total Claims Unpaid')  #Add chart legend showing whether claim has been paid or not  plt.legend(title="Claims Paid", loc="center left", bbox\_to\_anchor=(1, 0, 0.5, 1))  #Show the pivot table and the chart  print(claims\_paid)  plt.show() |
| Out [11]: |  |

**Second analysis of data and insight:**

From the chart above in Output [11], we can see that 93.1% of the insurance claims have been paid, and only 6.9% are unpaid. This shows that the insurance company processes the claims efficiently and has a relatively high approval rate. Even though most claims (93.1%) have been approved and paid out, a significant number of unpaid claims (6.9%) still require the insurance company’s attention. This 6.9% of unpaid claims total up to 1677 insurance claims waiting for approval or payment. This data also gives the insurance company an estimate of the amount to be paid if they use this number multiplied by the average amount excluding the outlier values. This allows the company to set aside enough cash to ensure that it can process the claims upon approval. However, the estimate may be inaccurate due to the possibility of many outliers in the 6.9% of unpaid claims.

**Third data & insight:**

(Code is a continuation of question 4’s second data & insight)

|  |  |
| --- | --- |
| In [12]: | #Create a pivot table with Type as the index number of Claim\_ID as the values and set aggfunc to count  claim\_type = df.pivot\_table(values='Claim\_ID', index='Type', aggfunc='count')  #Create a bar chart of the pivot table with no spacing and zorder 2 to prevent the gridlines from blocking the chart  ax = claim\_type.plot(kind='bar', width=1, zorder=2)  #Set the range from 0 to 25000 with 5000 intervals  plt.yticks(range(0, 25001, 5000))  #Set the Y axis number limit to 25000  plt.ylim(top=25000)  #Add gridlines behind the histogram on the y axis  plt.grid(axis='y', zorder=1)  #Set x and y axis labels and chart title  ax.set\_xlabel('Type of insurance')  ax.set\_ylabel('Counts of insurance')  ax.set\_title('Types of Insurance Claims')  #Add a blue legend showing 'Total' on the right side of the chart and in the middle  plt.legend(['Total'], loc='center right', bbox\_to\_anchor=(1.2, 0.5))  #Add data labels on top of the bars with +200 so the numbers are not touching the bars  for i, v in enumerate(claim\_type.values):  ax.text(i, v+200, str(v.item()), ha='center')  #Show the bar chart  plt.show() |
| Out [12]: |  |

**Third analysis of data and insight:**

From the bar chart above in Output [12], we can see that the insurance company has six different types of insurance claims, L001, L002, L003, L004, L005, and O001. The distribution across the different types of claims is significantly imbalanced, and L001 is the most common type of claim with a claim number of 21973. The rest of the claims have a much lower claim number than L001, with L002, L003, L004, L005, and O001 having an amount of 69, 7, 1, 247, and 1907 respectively. O001 has a much higher claim amount than the other four claim types but is still significantly lower than L001.

Without additional information on what the claim types stand for, I assume that the insurance company’s product mix relies heavily on one type of claim, which could show that the insurance company specializes in certain insurance coverage. Another assumption is that the less common claim types (L002, L003, L004, L005, and O001) are rare and more severe illnesses, making the claim amount so low.

**Question 5)**

(Run code by itself, not a continuation from the code above)

|  |  |
| --- | --- |
| In [1]: | import pandas as pd  #Defining eca\_file to the .csv file filled with the ECA data  #r is to prevent python from reading \ as an escape character  eca\_file = r"C:\Users\zijin\OneDrive\Desktop\ANL252\ECA.csv"  #Create a list that determine those missing values as blank, Unkn, and ???  missing\_values = ['', 'Unkn', '???']  #As blanks, Unkn and ??? are not considered part of python's NA values, we have to include it in using the na\_values function  df = pd.read\_csv(eca\_file, na\_values=missing\_values)  #Use the isna() function to change all values to true or false, true if value is missing and false if it is not missing  #Apply .any() function on axis=0 to see which column has a missing value (which in this case becomes true)  #Use df.columns to return the variables of columns that contains missing values  #Change the index to just a list of variable names that contains missing values  variables\_missing\_values = df.columns[df.isna().any(axis=0)].tolist()  #Print the variables with missing values  print(variables\_missing\_values)  #for mode function later  import statistics  #Use the isna() function count the number of missing or NA values in each columns  #If there is a missing or NA value, it will show True and will show False if the value is not missing  #The sum() function will treat True as 1 and False as 0 and sum up the total number of missing values in all the columns  print(df.isna().sum())  #Use the dropna() function to remove rows with missing values/NA values in both "Claim\_ID" and "Actual" columns  #Subset='Claim\_ID' and 'Actual' because these are the columns we want to remove rows with missing value from  #Use inplace=True to modify the original dataframe "df" instead of creating a new one  df.dropna(subset=['Claim\_ID', 'Actual'], inplace=True)  #Use of .mode() function to find the highest frequency term  #Include [0] because we only want 1 highest frequency term  #Fill missing values in "Terms" column with the mode of the column  #Use inplace=True to modify the original dataframe "df" instead of creating a new one  terms\_mode = df['Terms'].mode()[0]  df['Terms'].fillna(terms\_mode, inplace=True)  #Use the isna() function count the number of missing or NA values in each columns  #If there is a missing or NA value, it will show True and will show False if the value is not missing  #The sum() function will treat True as 1 and False as 0 and sum up the total number of missing values in all the columns  print(df.isna().sum())  #Check the number of duplicates  print(df.duplicated(subset=['Claim\_ID']).sum())  #Since Claim\_ID is suppose to be unique, there should not have any duplicates  #Drop any duplicates from Claim\_ID  df.drop\_duplicates(subset=['Claim\_ID'], inplace=True)  #Ensure that there are no more duplicates  print(df.duplicated(subset=['Claim\_ID']).sum())  #Use the pd.to\_datetime() function to ensure that both 'Actual' and 'Planned' are stored as datetime format  #Use .dt.strftime() function to change both columns of 'Actual' and 'Planned' to the specific day month year format  df['Actual'] = pd.to\_datetime(df['Actual']).dt.strftime('%d/%m/%Y')  df['Planned'] = pd.to\_datetime(df['Planned']).dt.strftime('%d/%m/%Y')  #Change 'Amount' column to numeric type since it is originally an object  #Use the pd.to\_numeric() function to change 'Amount' column into numbers which can be integer or float  #errors='coerce' turns any values in the 'Amount' column into NA values in the event it cannot be changed to numbers  df['Amount'] = pd.to\_numeric(df['Amount'], errors='coerce')  #Remove any NA values using dropna() function  df.dropna(subset=['Amount'], inplace=True)  #Round all values in 'Amount' column to 2 decimal points  df['Amount'] = df['Amount'].round(2) |
| Out [1]: |  |
| In [2]: | #import all the libraries that i might use later  import numpy as np  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  %matplotlib inline  #Convert both 'Actual' and 'Planned' columns to datetime format  df['Actual'] = pd.to\_datetime(df['Actual'])  df['Planned'] = pd.to\_datetime(df['Planned'])  #Calculate the delay in days between Actual and Planned in days  #Make all the number of days an absolute value  df['Delay'] = abs((df['Actual'] - df['Planned']).dt.days)  #Print the 'Delay' column  print(df['Delay']) |
| Out [2]: |  |
| In [3]: | from sklearn.preprocessing import LabelEncoder  #Set LabelEncoder() to variable name le  le = LabelEncoder()  #Use fit\_transform to change categorical data of 'Type' into numerical data because only numerical data is accepted as input  df['Type'] = le.fit\_transform(df['Type'])  #Split the data into 80% training data and 20% testing data  #Set 'Type' as X variable and 'Delay' as y variable  #Set random\_state to 0 because I want the same split every time I run the code  X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['Type'], df['Delay'], test\_size=0.2, random\_state=0)  #Define the linear regression model as variable lr  lr = LinearRegression()  #Fit the model to the training data  #Use values.reshape(-1, 1) to reshape the training dataset into 2D array  #Use .fit() function to train the model with training data  lr.fit(X\_train.values.reshape(-1, 1), y\_train)  #Use .predict function to make predictions on the test data  y\_pred = lr.predict(X\_test.values.reshape(-1, 1))  #Find the mean squared error (MSE) and R-squared values  mse = mean\_squared\_error(y\_test, y\_pred)  r2 = r2\_score(y\_test, y\_pred)  print("Mean Squared Error:", mse)  print("R-squared:", r2)  #Create a scatter plot of 'Delay' values on y axis and 'Type' values on x axis and the predicted values of 'Delay' is the red line shown in the chart  plt.scatter(X\_test, y\_test)  plt.plot(X\_test, y\_pred, color='red')  #Label x axis  plt.xlabel("Type of claim")  #Label y axis  plt.ylabel("Delay in Days")  #Label Title  plt.title("Linear Regression Model")  #Show the diagram  plt.show()  #Find the coefficient value of the linear regression  coefficient = lr.coef\_[0]  #Find the intercept value of the linear regression  intercept = lr.intercept\_  #Print coefficient and intercept values in 2dp  print(f"Coefficient Value/m Value : {coefficient:.2f}")  print(f"Intercept Value/c Value: {intercept:.2f}")  print(f"Linear Equation: y ={coefficient:.2f}x + {intercept:.2f}") |
| Out [3]: |  |
| In [4]: | from sklearn.preprocessing import LabelEncoder  #Set LabelEncoder() to variable name le  le = LabelEncoder()  #Use fit\_transform to change categorical data of 'Terms' into numerical data because only numerical data is accepted as input  df['Terms'] = le.fit\_transform(df['Terms'])  #Split the data into 80% training data and 20% testing data  #Set 'Terms' as X variable and 'Delay' as y variable  #Set random\_state to 0 because I want the same split every time I run the code  X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['Terms'], df['Delay'], test\_size=0.2, random\_state=0)  #Define the linear regression model as variable lr  lr = LinearRegression()  #Fit the model to the training data  #Use values.reshape(-1, 1) to reshape the training dataset into 2D array  #Use .fit() function to train the model with training data  lr.fit(X\_train.values.reshape(-1, 1), y\_train)  #Use .predict function to make predictions on the test data  y\_pred = lr.predict(X\_test.values.reshape(-1, 1))  #Find the mean squared error (MSE) and R-squared values  mse = mean\_squared\_error(y\_test, y\_pred)  r2 = r2\_score(y\_test, y\_pred)  print("Mean Squared Error:", mse)  print("R-squared:", r2)  #Create a scatter plot of 'Delay' values on y axis and 'Type' values on x axis and the predicted values of 'Delay' is the red line shown in the chart  plt.scatter(X\_test, y\_test)  plt.plot(X\_test, y\_pred, color='red')  #Label x axis  plt.xlabel("Terms of claim")  #Label y axis  plt.ylabel("Delay in Days")  #Label Title  plt.title("Linear Regression Model")  #Show the diagram  plt.show()  #Find the coefficient value of the linear regression  coefficient = lr.coef\_[0]  #Find the intercept value of the linear regression  intercept = lr.intercept\_  #Print coefficient and intercept values in 2dp  print(f"Coefficient Value/m Value : {coefficient:.2f}")  print(f"Intercept Value/c Value: {intercept:.2f}")  print(f"Linear Equation: y ={coefficient:.2f}x + {intercept:.2f}") |
| Out [4]: |  |
| In [5]: | from sklearn.preprocessing import LabelEncoder  #Split the data into 80% training data and 20% testing data  #Set 'Amount' as X variable and 'Delay' as y variable  #Set random\_state to 0 because I want the same split every time I run the code  X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['Amount'], df['Delay'], test\_size=0.2, random\_state=0)  #Define the linear regression model as variable lr  lr = LinearRegression()  #Fit the model to the training data  #Use values.reshape(-1, 1) to reshape the training dataset into 2D array  #Use .fit() function to train the model with training data  lr.fit(X\_train.values.reshape(-1, 1), y\_train)  #Use .predict function to make predictions on the test data  y\_pred = lr.predict(X\_test.values.reshape(-1, 1))  #Find the mean squared error (MSE) and R-squared values  mse = mean\_squared\_error(y\_test, y\_pred)  r2 = r2\_score(y\_test, y\_pred)  print("Mean Squared Error:", mse)  print("R-squared:", r2)  #Create a scatter plot of 'Delay' values on y axis and 'Type' values on x axis and the predicted values of 'Delay' is the red line shown in the chart  plt.scatter(X\_test, y\_test)  plt.plot(X\_test, y\_pred, color='red')  #Label x axis  plt.xlabel("Claim Amount")  #Label y axis  plt.ylabel("Delay in Days")  #Label Title  plt.title("Linear Regression Model")  #Show the diagram  plt.show()  #Find the coefficient value of the linear regression  coefficient = lr.coef\_[0]  #Find the intercept value of the linear regression  intercept = lr.intercept\_  #Print coefficient and intercept values in 2dp  print(f"Coefficient Value/m Value : {coefficient:.2f}")  print(f"Intercept Value/c Value: {intercept:.2f}")  print(f"Linear Equation: y ={coefficient:.2f}x + {intercept:.2f}") |
| Out [5]: |  |
| In [6]: | from sklearn.preprocessing import LabelEncoder  #Set LabelEncoder() to variable name le  le = LabelEncoder()  #Use fit\_transform to change categorical data of 'Terms' into numerical data because only numerical data is accepted as input  df['Region'] = le.fit\_transform(df['Region'])  #Split the data into 80% training data and 20% testing data  #Set 'Region' as X variable and 'Delay' as y variable  #Set random\_state to 0 because I want the same split every time I run the code  X\_train, X\_test, y\_train, y\_test = train\_test\_split(df['Region'], df['Delay'], test\_size=0.2, random\_state=0)  #Define the linear regression model as variable lr  lr = LinearRegression()  #Fit the model to the training data  #Use values.reshape(-1, 1) to reshape the training dataset into 2D array  #Use .fit() function to train the model with training data  lr.fit(X\_train.values.reshape(-1, 1), y\_train)  #Use .predict function to make predictions on the test data  y\_pred = lr.predict(X\_test.values.reshape(-1, 1))  #Find the mean squared error (MSE) and R-squared values  mse = mean\_squared\_error(y\_test, y\_pred)  r2 = r2\_score(y\_test, y\_pred)  print("Mean Squared Error:", mse)  print("R-squared:", r2)  #Create a scatter plot of 'Delay' values on y axis and 'Type' values on x axis and the predicted values of 'Delay' is the red line shown in the chart  plt.scatter(X\_test, y\_test)  plt.plot(X\_test, y\_pred, color='red')  #Label x axis  plt.xlabel("Region")  #Label y axis  plt.ylabel("Delay in Days")  #Label Title  plt.title("Linear Regression Model")  #Show the diagram  plt.show()  #Find the coefficient value of the linear regression  coefficient = lr.coef\_[0]  #Find the intercept value of the linear regression  intercept = lr.intercept\_  #Print coefficient and intercept values in 2dp  print(f"Coefficient Value/m Value : {coefficient:.2f}")  print(f"Intercept Value/c Value: {intercept:.2f}")  print(f"Linear Equation: y ={coefficient:.2f}x + {intercept:.2f}") |
| Out [6]: |  |

**Approach Explanation:**

I have changed my codes for the handling of the NA values such as blanks, ‘Unkn’, and ‘???’. Since now the ‘Actual’ column is needed to find the delays in days, I have removed the 1677 NA values from the ‘Actual’ columns as compared to previously when I replaced it with the ‘Planned’ dates. This is to avoid biased data showing that there are no delays between planned and actual dates. The rest of the data preparation and treatments of data are the same as questions 2 and 3. To find the delay in days, I used the abs() function to get the absolute value of the Actual date minus the Planned date as shown in input [2]. From there, I used the fit\_transform() function to change the independent variables: ‘Type’, ‘Terms’, and ‘Region’ from categorical data to numerical data in input [3], [4], and [6] respectively. I used the train\_test\_split() function for all the models with ‘Delay’ as the dependent variable in the y-axis. With a test size of 0.2, it splits the data into 80% training and 20% testing. The fit() and predict() functions are used to fit the linear regression model to training data and predict the testing data respectively. R2 and mean squared error functions are both used to calculate the R2 and mean squared error. A scatter diagram is created with the independent variable on the x-axis while the dependent variable is on the y-axis. The red line in the diagram represents the predicted values of days delayed. The coefficient and intercept values are also created for me to form the linear regression equation.

**Question 6)**

**Results obtained (Type and Delay):**

Chart

Description automatically generated

The linear regression equation is y = mx + c, so in this case it is y = 0.62x + 3.57.

With the R2 value of only 0.0084, it suggests that there is a very weak correlation between Claim Type and Delay in days. This means that only a small amount of delay can be explained by the claim type and claim type should not be a predictor of delay values. The equation y = 0.62x + 3.57 tells us that for every 1 unit increase in claim type, delay value is expected to increase by 0.62 units and if the claim type is zero, the predicted delay value is 3.57.

**Results obtained (Terms and Delay):**

Chart, scatter chart

Description automatically generated

The linear regression equation is y = mx + c, so in this case it is y = 0.11x + 2.20.

With the R2 value of only 0.051, it suggests that there is a very weak correlation between Claim Terms and Delay in days. This means that only a small amount of delay can be explained by the claim terms and claim terms should not be a predictor of delay values. The equation y = 0.11x + 2.20 tells us that for every 1 unit increase in claim type, delay value is expected to increase by 0.11 units and if the claim type is zero, the predicted delay value is 2.20.

**Results obtained (Amount and Delay):**

Chart, scatter chart

Description automatically generated

The Linear Regression Equation of y = 0x + 4.35, shows that the amount of claim has no effect on the delay value. This means that there is no relationship between the amount and delay and the predicted delay value will always be 4.35.

**Results obtained (Region and Delay):**

Graphical user interface

Description automatically generated

The Linear regression equation is y = -2.89x + 6.51 which suggests that Region and Delay are negatively correlated. For every one-unit increase in the region, the delay value will decrease by 2.89.