A picture containing plate, drawing, food

Description automatically generated

**ANL252**

**Python For Data Analytics** 

**End-of-Course Assessment**

**Submission Date:** **13 September 2021**

|  |  |
| --- | --- |
| Name | **PI Number** |
| Danial Loke Zhi Weng | Y1981340 |

**Qn1(a)(i)**

|  |  |
| --- | --- |
| **Code:** | import pandas as pd  # Reading the "ship.csv" as a DataFrame  # The missing values in the data are ".", thus need to mention it under na\_values to declare as missing values.  ship = pd.read\_csv("ship.csv", na\_values = ".")  ship  # ship.isnull().sum(). To check if the number of missing values tally with the question. |
| **Output:** |  |

**Qn1(a)(ii)**

|  |  |
| --- | --- |
| **Code:** | # Use .rename to rename the columns.  ship.rename(columns = {"T":"types", "A":"c\_years", "P":"o\_periods", "MS":"s\_months", "Y":"incidents"}, inplace = True)  ship |
| **Output:** |  |

**Qn1(a)(iii)**

|  |  |
| --- | --- |
| **Code:** | # Since variables "types" and "o\_periods" are of interest, we group them using .groupby().  # After grouping, we can calculate the mean of "s\_months" and "incidents" based on the grouping by using .mean().  # We can use round() to round result to integers.  # Store results in "shipgroup"  shipgroup = round(ship.groupby(by = ["types", "o\_periods"])[["s\_months", "incidents"]].mean())  shipgroup |
| **Output:** |  |

**Qn1(a)(iv)**

|  |  |
| --- | --- |
| **Code:** | # To replace missing values, use .fillna().  # .fillna() is based on index, thus "ship" and "shipgroup" need to have the same index in order for .fillna() work.  # Set index for "ship" by using .set\_index so that the index will be the same as the shipgroup DataFrame.  # "drop" parameter should be False because we want to keep the respective columns after setting them as index.  ship.set\_index(["types", "o\_periods"], drop = False, inplace = True)  ship.fillna({"s\_months":shipgroup["s\_months"], "incidents":shipgroup["incidents"]}, inplace = True)  ship |
| **Output:** |  |

**Qn1(a)(v)**

|  |  |
| --- | --- |
| **Code:** | # Creating a pandas DataFrame named "Y" to save target variable "incidents".  Y = pd.DataFrame(ship["incidents"])  Y.reset\_index(drop = True, inplace = True)  Y |
| **Output:** |  |

**Qn1(b)(i)**

|  |  |
| --- | --- |
| **Code:** | # Convert the relevant variables to categorical using .astype().  ship.astype({"types":"category", "c\_years":"category", "o\_periods":"category"}) |
| **Output:** |  |

**Qn1(b)(ii)**

|  |  |
| --- | --- |
| **Code:** | # Use pd.get\_dummies to convert categorical data to dummy variables.  X = pd.get\_dummies(ship[["types", "c\_years", "o\_periods"]], columns = ["types", "c\_years", "o\_periods"])  X |
| **Output:** |  |

**Qn1(b)(iii)**

|  |  |
| --- | --- |
| **Code:** | import numpy as np  # log-transformation on s\_months using np.log.  # Attach to both "X" and "ship" DataFrames.  X["log\_s\_months"] = np.log(ship["s\_months"])  ship["log\_s\_months"] = np.log(ship["s\_months"])  print("This is to check if the transformed variable log\_s\_months is attached to both DataFrames: ")  display(X.head(), ship.head()) |
| **Output:** |  |

**Qn1(c)**

In general, when you split a DataFrame into training and testing datasets, a larger proportion of data will be used for training while a smaller proportion of data will be used for testing. The training dataset is used to train the model whereas the testing dataset is used to access the model's accuracy. This means that the quality and quantity of data used for training is crucial in determining the effectiveness of the model. Hence, assuming that the quality of the data is there, having a larger dataset allows for greater accuracy of a model as there will be more data to train.

In this case, the dataset is already relatively small where there are only 40 observations, thus splitting the DataFrame would not be ideal as it would further reduce the data that can be used for training. Therefore, using the entire dataset for training purpose instead would maximise the accuracy of the model, thus yielding a more statistically meaningful result as compared to splitting the dataset.

**Qn1(d)**

|  |  |
| --- | --- |
| **Code:** | # Saving df 'ship' as a new csv text file using .to\_csv.  ship.to\_csv("ship\_prepared.csv", index = False)  import sqlite3  # Create database by using .connect.  # Create a cursor for the DataBase.  # Create the df by reading the CSV "ship\_prepared".  # Export the df to database as tables by .to\_sql.  conn = sqlite3.connect("ship.db")  cur = conn.cursor()  ship\_prepared = pd.read\_csv("ship\_prepared.csv")  ship\_prepared.to\_sql("ship\_prepared", con = conn, if\_exists = 'replace')  conn.commit()  conn.close() |
| **Output:** | No output. |

**Qn2(a)**

The corresponding scikit-learn module for the Poisson regression would be the "linear\_model" module. This is because Poisson regression is a generalized linear model form of regression analysis that is used for modelling count data, thus being a type of linear model.

The estimator for the poisson regression can be found under "Generalized Linear Regression" from the "linear\_model" module. The estimator is called "PoissonRegressor". The parameters and the following default values of this estimator are sklearn.linear\_model.PoissonRegressor(\*, alpha = 1.0, fit\_intercept = True, max\_iter = 100, tol = 0.0001, warm\_start = False, verbose = 0).

"alpha" is a constant that multiplies the penalty term, thus influencing the regularization strength. "alpha" must be a float and an "alpha" of 0 refers to unpenalized GLMs. "fit\_intercept" is a boolean value (True/False) that determines if a constant should be added to the linear predictor. "max\_iter" requires an integer value that refers to the maximum number of iterations for the solver. "tol" is a float value that refers to the stopping criterion of the solver. "warm\_start" is a boolean value (True/False). If it is set as True, the solution of the previous call to fit will be reused as initialization to solve for the coefficient (coef\_) and intercept (intercept\_). "verbose" is a positive integer value that controls the level of verbosity.

The information of the model are known as attributes. The attributes of the "PoissonRegressor" are "coef\_", "intercept\_" and "n\_iter\_". "coef\_" is used to check the estimated coefficients for the linear predictor in the GLM. "intercept\_" is used to check the value of the intercept. "n\_iter\_" is used to check the actual number of iterations used in the solver.

The fit and predict are 2 of the functions of the "PoissonRegressor" estimator. "fit" is used to fit a Generalized Linear Model through the chosen X and Y variable values. The parameters and default values of "fit" are fit(X, y, sample\_weight = None). "X" refers to the training data. The object should be array-like or a sparse matrix that is 2 dimensional in shape (rows, columns). "y" refers to the target values. The object should be array-like that 1 dimensional (rows,). The object for "sample\_weight" should be array-like and 1 dimensional in shape. The default value as mentioned is "None".

The predict function returns the predicted values of Y with the X variables. The parameters of "predict" is predict(X). The object of "X" should be array-like or a sparse matrix that is 2 dimensional in shape (rows, columns). Predict will return the predicted values of Y in a 1 dimensional shaped array.

**Qn2(b)**

|  |  |
| --- | --- |
| **Code:** | # import the "linear\_model" module from sklearn.  from sklearn import linear\_model  clf = linear\_model.PoissonRegressor()  # For .fit() to work, the y parameter needs to be in a 1 dimensional array.  # Check the shape of the df X and Y.  print(f"The shape of DataFrames X and Y are {X.shape} and {Y.shape} respectively")  # Convert df Y to a 1 dimensional numpy array by using .to\_numpy().flatten().  Y\_array = Y.to\_numpy().flatten()  print(f"The shape of the Y array is {Y\_array.shape}")  # Use df X for the X parameter and Y\_array for the y parameter.  clf.fit(X, Y\_array)  # After fitting, use .coef\_ to find the coefficients.  coef = clf.coef\_  coef\_df = pd.DataFrame(coef).rename(columns = {0:"Coef"})  coef\_df |
| **Output:** |  |

**Qn2(c)**

|  |  |
| --- | --- |
| **Code:** | # To find the Esitmated Y values (EY), use .predict() to predict Y using GLM feature.  # .predict() will return the expected value EY.  E\_Y = clf.predict(X)  E\_Y\_df = pd.DataFrame(E\_Y).rename(columns = {0:"EY"})  E\_Y\_df |
| **Output:** |  |

|  |  |
| --- | --- |
| **Code:** | # Defining the function for the Deviance.  # 'for' loop will be used for the summation of the equation  # The list will be the index/row of the observations.  # The equation within the curly braces is D3 and will be split into 2 parts D1 and D2.  # if and else statements will be used for D1 because if Y = 0, D1 will be 0 as well.  # D2 will be the second part of the equation.  # We use 'total = total + D3' to sum the different observations.  # D\_value will x2 the summation (total) and the user-defined function will return D\_value.  def D\_function(Y, EY, index):  total = 0  for i in index:  if Y.iloc[i] == 0:  D1 = 0  else:  D1 = Y.iloc[i]\*np.log(Y.iloc[i]/EY.iloc[i])  D2 = Y.iloc[i] - EY.iloc[i]  D3 = D1 - D2  total = total + D3    D\_value = round(total\*2, 4)  return D\_value  # Create an a list that matches the number of rows/index of df Y and EY. In this case, ranging from 0-39.  index\_list = list(range(0,40))  # Calculating the Deviance. Round to 4dp.  D = D\_function(Y["incidents"], E\_Y\_df["EY"], index\_list)  print(f"The value for D is:\n{D}") |
| **Output:** |  |