

**ANL252 (Online)**

**PYTHON FOR DATA ANALYTICS**

# **END-OF-COURSE ASSESSMENT**

**July 2021 Presentation**

**Submitted by:**

|  |  |
| --- | --- |
| **Name** | **PI No.** |
| **Michelle Tan Ming Hui** | **J2110299** |

**Tutorial Group: ­­­­­­­­­­­­ T09**

**Instructor’s Name: Dr. Munish Kumar**

**Submission Date: 13/09/2021**

**# ANL252\_ECA\_J2110299\_MichelleTanMingHui**

**# Michelle Tan**

|  |
| --- |
| **Question 1(a)(i)** |
| **Codes:**  # importing libraries  import numpy as np  import pandas as pd  import matplotlib.pyplot as plt  # read in the csv file into pandas DataFrame  # import raw .csv to analyse  df\_ship\_raw = pd.read\_csv("ship.csv")  display(df\_ship\_raw) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(i)** |
| **Codes:**  # declaring "." as missing values, "NaN"  df\_ship = pd.read\_csv("ship.csv", na\_values = ".")  display(df\_ship)  # showing the number of missing values in each column  display(df\_ship.isnull().sum())  # showing the total number of missing values  display(df\_ship.isnull().sum().sum())  # printing out the total count of missing values  total\_count = df\_ship.isnull().sum().sum()  print(f"The total count of missing values is {total\_count}.") |
| **Output:** |

|  |
| --- |
| **Question 1(a)(ii)** |
| **Codes:**  # renaming the variables respectively  df\_ship = df\_ship.rename({"T":"types", "A": "c\_years", "P": "o\_periods", "MS": "s\_months", "Y": "incidents"}, axis='columns')    display(df\_ship) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iii)** |
| **Codes:**  # showing the means respectively, with min values and max values  import numpy as np  display(df\_ship.groupby(['types','o\_periods'])[['s\_months', 'incidents']].agg([min, max, np.mean]))  # displaying summary of mean of service months and incidents  newdf\_ship = df\_ship.groupby(['types','o\_periods'])[['s\_months', 'incidents']].mean().reset\_index()  # using np.floor to truncate the 's\_months' & 'incidents' columns  # rounding off to nearest integers  newdf\_ship[['s\_months', 'incidents']] = newdf\_ship[['s\_months', 'incidents']].apply(np.floor)    display(newdf\_ship) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iii)** |
| **Codes:**  # storing it to an object named “shipgroup”  shipgroup = newdf\_ship  display(shipgroup) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting df\_ship out  display(df\_ship) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting the data that have types=1 and o\_periods=1  df\_ship\_11 = df\_ship.loc[(df\_ship['types'] == 1) & (df\_ship['o\_periods'] == 1)]  display(df\_ship\_11) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # finding out the mean of s\_months and incidents from types=1 and o\_periods=1  # displaying the values to the nearest interger  df\_ship\_11[['s\_months','incidents']].mean().round(0).astype(int) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # replacing the na values with the respective mean from types=1 and o\_periods=1, as above  # leveraging the respective mean values as above  values = {"s\_months": 911, "incidents": 3}  df\_ship\_11\_r = df\_ship\_11.fillna(value=values)  display(df\_ship\_11\_r) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting the data that have types=2 and o\_periods=1  df\_ship\_21 = df\_ship.loc[(df\_ship['types'] == 2) & (df\_ship['o\_periods'] == 1)]  display(df\_ship\_21) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # finding out the mean of s\_months and incidents from types=2 and o\_periods=1  # displaying the values to the nearest interger  df\_ship\_21[['s\_months','incidents']].mean().round(0).astype(int) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # replacing the na values with the respective mean from types=2 and o\_periods=1, as above  # leveraging the respective mean values as above  values = {"s\_months": 26852, "incidents": 36}  df\_ship\_21\_r = df\_ship\_21.fillna(value=values)  display(df\_ship\_21\_r) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting the data that have types=3 and o\_periods=1  df\_ship\_31 = df\_ship.loc[(df\_ship['types'] == 3) & (df\_ship['o\_periods'] == 1)]  display(df\_ship\_31) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # finding out the mean of s\_months and incidents from types=3 and o\_periods=1  # displaying the values to the nearest interger  df\_ship\_31[['s\_months','incidents']].mean().round(0).astype(int) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # replacing the na values with the respective mean from types=3 and o\_periods=1, as above  # leveraging the respective mean values as above  values = {"s\_months": 914, "incidents": 2}  df\_ship\_31\_r = df\_ship\_31.fillna(value=values)  display(df\_ship\_31\_r) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting the data that have types=4 and o\_periods=1  df\_ship\_41 = df\_ship.loc[(df\_ship['types'] == 4) & (df\_ship['o\_periods'] == 1)]  display(df\_ship\_41) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # finding out the mean of s\_months and incidents from types=4 and o\_periods=1  # displaying the values to the nearest interger  df\_ship\_41[['s\_months','incidents']].mean().round(0).astype(int) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # replacing the na values with the respective mean from types=4 and o\_periods=1, as above  # leveraging the respective mean values as above  values = {"s\_months": 296, "incidents": 1}  df\_ship\_41\_r = df\_ship\_41.fillna(value=values)  display(df\_ship\_41\_r) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting the data that have types=5 and o\_periods=2  df\_ship\_52 = df\_ship.loc[(df\_ship['types'] == 5) & (df\_ship['o\_periods'] == 2)]  display(df\_ship\_52) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # finding out the mean of s\_months and incidents from types=5 and o\_periods=2  # displaying the values to the nearest interger  df\_ship\_52[['s\_months','incidents']].mean().round(0).astype(int) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # replacing the na values with the respective mean from types=5 and o\_periods=2, as above  # leveraging the respective mean values as above  values = {"s\_months": 1047, "incidents": 7}  df\_ship\_52\_r = df\_ship\_52.fillna(value=values)  display(df\_ship\_52\_r) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting the data that have types=5 and o\_periods=1  df\_ship\_51 = df\_ship.loc[(df\_ship['types'] == 5) & (df\_ship['o\_periods'] == 1)]  display(df\_ship\_51) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # finding out the mean of s\_months and incidents from types=5 and o\_periods=1  # displaying the values to the nearest interger  df\_ship\_51[['s\_months','incidents']].mean().round(0).astype(int) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # replacing the na values with the respective mean from types=5 and o\_periods=1, as above  # leveraging the respective mean values as above  values = {"s\_months": 664, "incidents": 4}  df\_ship\_51\_r = df\_ship\_51.fillna(value=values)  display(df\_ship\_51\_r) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # showing the row of index 6,  # with the na value in s\_months and incidents columns being replaced with the calculated mean values  rows = [6]  columns = ["types", "c\_years", "o\_periods", "s\_months", "incidents"]  df\_ship\_11\_r1 = df\_ship\_11\_r.loc[rows, columns]  display(df\_ship\_11\_r1) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # showing the row of index 14,  # with the na value in s\_months and incidents columns being replaced with the calculated mean values  rows = [14]  columns = ["types", "c\_years", "o\_periods", "s\_months", "incidents"]  df\_ship\_21\_r1 = df\_ship\_21\_r.loc[rows, columns]  display(df\_ship\_21\_r1) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # showing the row of index 22,  # with the na value in s\_months and incidents columns being replaced with the calculated mean values  rows = [22]  columns = ["types", "c\_years", "o\_periods", "s\_months", "incidents"]  df\_ship\_31\_r1 = df\_ship\_31\_r.loc[rows, columns]  display(df\_ship\_31\_r1) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # showing the row of index 30,  # with the na value in s\_months and incidents columns being replaced with the calculated mean values  rows = [30]  columns = ["types", "c\_years", "o\_periods", "s\_months", "incidents"]  df\_ship\_41\_r1 = df\_ship\_41\_r.loc[rows, columns]  display(df\_ship\_41\_r1) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # showing the row of index 33,  # with the na value in s\_months and incidents columns being replaced with the calculated mean values  rows = [33]  columns = ["types", "c\_years", "o\_periods", "s\_months", "incidents"]  df\_ship\_52\_r1 = df\_ship\_52\_r.loc[rows, columns]  display(df\_ship\_52\_r1) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # showing the row of index 38,  # with the na value in s\_months and incidents columns being replaced with the calculated mean values  rows = [38]  columns = ["types", "c\_years", "o\_periods", "s\_months", "incidents"]  df\_ship\_51\_r1 = df\_ship\_51\_r.loc[rows, columns]  display(df\_ship\_51\_r1) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # doing a union for all the rows of index with na values,  # with their na values being replaced with the respective mean values  # putting them in a DataFrame  pd.concat([df\_ship\_11\_r1, df\_ship\_21\_r1, df\_ship\_31\_r1, df\_ship\_41\_r1, df\_ship\_52\_r1, df\_ship\_51\_r1]) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # extracting data set with na values  display(df\_ship) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # dropping the rows of index that have na values  df\_ship\_dropna = df\_ship.drop(df\_ship.index[[6,14,22,30,33,38]])  display(df\_ship\_dropna) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # doing a union for the df\_ship\_dropna and the replaced values DataFrame  df\_unionall = pd.concat([df\_ship\_11\_r1, df\_ship\_21\_r1, df\_ship\_31\_r1, df\_ship\_41\_r1, df\_ship\_52\_r1, df\_ship\_51\_r1, df\_ship\_dropna])  display(df\_unionall) |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # sorting the index numbers to ascending order  # values inside the DataFrame is in ascending order as well  df\_unionall = df\_unionall.sort\_index()  df\_unionall |
| **Output:** |

|  |
| --- |
| **Question 1(a)(iv)** |
| **Codes:**  # saving the result into a panda DataFrame, named "ship"  # link to Q1(b)(iii)  ship = df\_unionall  ship |
| **Output:** |

|  |
| --- |
| **Question 1(a)(v)** |
| **Codes:**  # displaying the column of incidents  df\_unionall\_incidents = df\_unionall[["incidents"]]  df\_unionall\_incidents |
| **Output:** |

|  |
| --- |
| **Question 1(a)(v)** |
| **Codes:**  # saving the result into a panda DataFrame, named "Y"  Y = df\_unionall\_incidents  Y |
| **Output:** |

|  |
| --- |
| **Question 1(b)(i)** |
| **Codes:**  # passing a list of columns  # converting column type to categorical  cols = ['types', 'c\_years', 'o\_periods']  df\_ship[cols] = df\_ship[cols].astype('category')  # printing out the output that shows categorical  print(df\_ship.dtypes) |
| **Output:** |

|  |
| --- |
| **Question 1(b)(ii)** |
| **Codes:**  # using dummy coding to create dummy variables from categorical variables (types, c\_years, o\_periods)  df\_unionall\_dummies = pd.get\_dummies(df\_unionall, prefix='', prefix\_sep='',  columns=['types', 'c\_years', 'o\_periods'])  df\_unionall\_dummies |
| **Output:** |

|  |
| --- |
| **Question 1(b)(ii)** |
| **Codes:**  # saving the result into a panda DataFrame, named "X"  X = df\_unionall\_dummies  display(X) |
| **Output:** |

|  |
| --- |
| **Question 1(b)(iii)** |
| **Codes:**  # transform s\_months by the natural logarithm  # attaching to DataFrame "X"  X["log\_s\_months"] = np.log(X["s\_months"])  X |
| **Output:** |

|  |
| --- |
| **Question 1(b)(iii)** |
| **Codes:**  # transform s\_months by the natural logarithm  # attaching to DataFrame "ship"  ship["log\_s\_months"] = np.log(ship["s\_months"])  ship |
| **Output:** |

|  |
| --- |
| **Question 1(c)** |
| **Answer:**  After evaluating the current dataset, it is not logical to use the Train-Test split here. One of the reasons is that the dataset is very small here, with 39 rows of index only. If the dataset is segregated into train and test sets; the training dataset will not contain enough data for the model to develop an efficient input-output mapping. There will also be insufficient data in the test applied to analyse the model's performance successfully. The predicted performance may be highly optimistic (excellent) or pessimistic (poor).  Instead, we shall use the entire dataset for training purpose. The more data our deployed model has seen, the greater it should generalize, in theory. As a result, if we trained the model on the whole set of data we have, it should generalize better than those using the Train-Test split, a model that have only a certain percentage for their training set. Nevertheless, because we do not have a test dataset, we are unable to make statistical or performance assumptions based on it.  Furthermore, we may compute real-time performance metrics if we release a model to production using the full training dataset and discover the true values of the target variable of the new incoming data (i.e. the data the production model is making predictions on). The hyperparameters of the model might be updated as a result of this procedure to improve performance. Thus, if a dataset contains enough data and variety, it is advisable to use the Train-Test split to 80:20. It would be satisfactory to train a robust model with no generalisation issues. |

|  |
| --- |
| **Question 1(d)** |
| **Codes:**  # saving the prepared DataFrame "ship" as a new csv file called "ship\_prepared.csv"  # export DataFrame to csv  ship.to\_csv('ship\_prepared.csv', index = False)  # importing the libraries  import pandas as pd  import sqlite3  # connect database and create cursor  conn = sqlite3.connect("ship.db")  cur = conn.cursor()  # reading in the csv file  ship\_prepared\_raw = pd.read\_csv("ship\_prepared.csv")  ship\_prepared\_raw  ship\_prepared\_raw.to\_sql("ship\_prepared\_raw", conn, if\_exists = "replace", index = False)  # executing the query  cur.execute("SELECT \* FROM ship\_prepared\_raw;")  print(cur.fetchone())  print(cur.fetchall())  # creating a dataframe object  ship\_prepared = pd.DataFrame(cur.fetchall())  # find database attributes from cursor.description to use as column names  cur.description  # creating dataframe columns  cols = [column[0] for column in cur.description]  cols |
| **Output:** |

|  |
| --- |
| **Question 2(a)** |
| **Answer:**  Scikit-learn is a Python module for machine learning based on SciPy that is released under the 3-Clause BSD license. Scikit-learn offers a standard Python interface for a variety of supervised and unsupervised learning algorithms. Classification, regression, clustering, and dimensionality reduction are just a few of the powerful methods in the sklearn library for machine learning and statistical modelling.  Next, an estimator is one of the leading object in API (application programming interface). It provides a uniform interface for a broad range of machine learning applications, which is why the Estimator API is used to construct all machine learning algorithms in Scikit-Learn. An estimator is an object that derives from data (fits the data). It may be used with any algorithm that extracts valuable features from original data, such as classification, regression, clustering, or even a transformer. Parameters are determined from the data after it has been fitted with an estimator. The Estimator object is used for model estimating and decoding.  The training phase of the modeling process is simply fitting your model to the training data (i.e. using the.fit() function on it). It uses the algorithm to get the coefficients for the equation provided (e.g. Linear Regression equation). It generates and stores the parameters or weights based on the training data (e.g. parameters provided by coef() in the case of Linear Regression). After we have trained our model, we apply the weights obtained above to the test data to construct the predictions. The method is called predict(), it make predictions based on unknown test data. The predict() function makes a prediction for each test case, and it generally only takes one argument (X). The predicted value for classifiers and regressors would be in the same domain as the one observed in the training set. The projected value in clustering estimators would be an integer. The projected values of the supplied test instances will be delivered as an array or sparse matrix output. The fit() function have to be carried out before you attempt to run predict().  Lastly, we shall apply algorithm tuning in the process of applied machine learning before displaying the results. It's also known as hyperparameter optimization, because the algorithm parameters are named hyperparameters, while the coefficients discovered by the machine learning algorithm are named parameters. The problem's search-nature is suggested by optimization. Some examples like kernel, C and gamma for Support Vector Classifier are passed as arguments to the operaters of the estimator classes. Examining the hyper-parameter space for the finest cross validation score is encouraged and achievable. In scikit-learn, two main methods to parameter search are offered. They are namely grid search and random search. Grid search is a technique of parameter tuning that develops and evaluates a model for each combination of algorithm parameters provided on a grid in a systematic manner. Random search is a technique of parameter tuning in which a set number of repetitions is used to sample algorithm parameters from a probability sampling. For each parameter combination specified, a classifier is developed and analysed. |

|  |
| --- |
| **Question 2(b)** |
| **Answer:**  After analysing the data, the Poisson log-linear model, in essence, is a model for n answers Y1,..., Yn with integer count values. Each Yi is represented as a separate Poisson(i) random variable, with logE(Y) being a linear combination of the variables relating to the k th observation. We regard the variables as fixed constants, same as we do in linear and logistic regression, and the model parameters to be presumed are the Poisson regression coefficients β = (𝛽0, 𝛽1,. . ,𝛽k). |

|  |
| --- |
| **Question 2(b)** |
| **Codes:**  # importing libraries  import statsmodels.api as sm  from statsmodels.formula.api import glm  # fit Poisson regression  Y = glm('incidents ~ incidents', data = Y, family = sm.families.Poisson()).fit()  # display model results  print(Y.summary()) |
| **Output:** |

|  |
| --- |
| **Question 2(b)** |
| **Codes:**  # poissson distribution  # trying to plot a graph  fig, axes = plt.subplots(1, 3, figsize=(14, 3), sharey=True)  xx = np.arange(20)  lambdas = [1, 5, 10]  y = s\_months  for i, lam in enumerate(lambdas):  yy = y.poisson.pmf(xx, lam)  axes[i].bar(xx, yy)  axes[i].set\_title(r'$\Y={}$'.format(lam)) |