**EDA**

**(Exploratory Data Analysis)**

|  |  |  |
| --- | --- | --- |
| **STEP-1 : EXAMINE THE DATA** | | |
| **Instructions** | **Example Syntaxes** | **Example Functions** |
| **1. Import the needed libraries**  **-** Pandas, numpy, matplotlib.pyplot, seaborn  - warnings (*to ignore the error warnings*)  - use set\_option to see the all of the columns and rows | **import pandas as pd**  **import numpy as np**  **import warnings**  **pd.set\_option('display.max\_columns', 500)**  **pd.set\_option('display.max\_colwidth', 500)**  **pd.set\_option('display.max\_info\_columns', 500)**  **pd.set\_option('display.max\_info\_rows', 2000)**  **pd.set\_option('display.expand\_frame\_repr', True)**  **pd.set\_option('display.width', 2000)**  **warnings.filterwarnings("ignore")**  **warnings.warn("this will not show")** |  |
| **2.Read the file and assign the dataset into a DataFrame**  - pd.read\_csv()  - pd.read\_excel()  - pd.read\_html()  - pd.read\_jason() | **df = pd.read\_json("file.json", orient = "records", lines = True)**  orient :  - ``'split'`` : dict like  ``{index -> [index], columns -> [columns], data -> [values]}``  - ``'records'`` : list like  ``[{column -> value}, ... , {column -> value}]``  - ``'index'`` : dict like ``{index -> {column -> value}}``  - ``'columns'`` : dict like ``{column -> {index -> value}}``  - ``'values'`` : just the values array  lines : bool, default False  Read the file as a json object per line. |  |
| **df = pd.read\_excel(“file.xlsx” , sheet\_name = “Sheet1”)**  sheet\_name : str, int, list, or None, default 0  Strings are used for sheet names. Integers are used in zero-  indexed sheet positions. Lists of strings/integers are used to  request multiple sheets. Specify None to get all sheets. |
| **3. Examine the data** | **df.info()**  **-**  You can see the all columns, types and row numbers **df.describe()**  -Automatically make calculation with numeric columns  **col\_list = list(df.columns)**   * To use the columns names in a list for further syntaxes |  |
| **4. Identify the amount of the missing values** | **df.shape**   * You can see the counts of row and column   **df.isnull().sum()/df.shape[0]\*100**   * To see the percentage of NaN values for each columns |  |

|  |  |  |
| --- | --- | --- |
| **STEP-2 : FIX THE CORRUPTED DATA FORMATS** | | |
| **Instructions** | **Example Syntaxes** | **Example Functions** |
| 1. You can remove the columns which has high percentage of missing values (NaN).  ( You have to decide what percentage of the missing value is proper for your work.For example: You can drop the columns which have 90 percentage or high of missing values .) | **drop\_list = ["Column1", "Column4", "Column23", ..... ]**  **df.drop(drop\_list1, axis = 1, inplace =True)**  ( You can change the columns you want to drop in the drop\_list. But do not forget the check the values in the columns before dropping. Maybe you can catch a beneficial data) |  |
| 2. At the begining you can focus on the columns that have %35 precentage and higher of missing values.  ( You can change the percentage according to the situation of the data file.) |  | **def show\_nans(df, limit):**  **missing = df.isnull().sum()\*100/df.shape[0]**  **return missing.loc[lambda x : x > limit]**  **show\_nans(df, 35)**  ( To see the only columns that have defined percentage( in this example: > %35) and more of missing values) |
| 3. Investigate and fix all of the selected columns to focus.  a. You can drop the columns are asumed that they don’t have meaningfull effect on the result.  b. If there are similar columns; You can select the one column among the similar ones and drop the others by examining them to choose the more meaningfull one.  (1) You can simplify the values by using str.replace() syntax to get the meaningfull number/data from the values.  (2) You can fill the missing values with proportion of other existing values using bfill method.  (3) You can fill the missing values with the most repeated value (mode) of an meaningfull and have not NaN column.  (4) You can just examine the value\_counts of the similar columns and select one of them according to the situation of values.  (5) Drop the unselected similar columns  c. You can change the values of columns to 1 or 2, as multiplying them 1 after changing them TRUE or FALSE.      d. You can change the list values to string by using “”.join() method or “str[0]” method.  e. You can use regex “str.extract” method to get the only numbers or dates if value is suitable.  f. You can change the time format to datetime by using “to\_datetime”.    g.You can use “loc[condition, “column]” to get the columns values under defined condition.  h. If you have a important but long string values seperated a sign, you can use str.get\_dummies() method to seperated each item in the values. And you can join each seperated items of values to your dataframes as new columns with prefixes defined by you.  ı. You can investigate the columns using sample() method again and again.  i. You can change object type to float by using “to\_numeric()” method. | **drop\_list = ["Column1", "Column4", "Column23", ..... ]**  **df.drop(drop\_list1, axis = 1, inplace =True)**  ( You can change the columns you want to drop in the drop\_list. But do not forget the check the values in the columns before dropping. Maybe you can catch a beneficial data) | **def fill\_most(df, group\_col, col\_name):**  **for group in list(df[group\_col].unique()) :**  **grp\_inx = list(df[df[group\_col] == group][col\_name].index)**  **df[col\_name].iloc[grp\_inx] = df[col\_name].iloc[grp\_inx].fillna(df[df[group\_col] == group][col\_name].mode()[0])**  ( To fill the “col\_name” columns’ NaN values with the most repeated value (mode) of the “group\_col” column.  It’s recommended that “group\_col” column has no NaN values, and has meaningfull values)  **def Nan\_to\_list(data, column\_name):**  **NaN\_rows = data[column\_name].isnull()**  **data.loc[NaN\_rows, column\_name] = data.loc[NaN\_rows, column\_name].apply(lambda x : [])**  ( To cahnge the NaN values as [] )  **def unique\_values(data, column\_name):**  **unique\_vals = set()**  **for rows in data[column\_name]:**  **unique\_vals.update(rows)**  **return list(unique\_vals)**  ( To get the unique values in a list. For this function all NaN values have to be in a list-- []) |
| **df[["Similar\_Column1" , "Similar\_Column2" ]]**  ( To check the more meaningfull one among the similar ones easily) |
| **df[“Column"].value\_counts(dropna = False)**  ( To see the containing counts of unique rows in the columns)  (**dropna :** True =**I**nclude counts of NaN  False = Don’t include counts of NaN) |
| **df["Column"].str.strip("Dirty\_sign")**  ( To get the meaningfull value from the dirty data by Dirty\_sign)  ( If the dirty value in a list [], we will get NaN values for them)  ( To solve this problem we can create a new column that including  only the values in the list  **df["New\_Clmn\_for\_Lists"] = df['Column'].str[0].str[1]**  (And then get the values except ones in lists in a new column)  (İndex number in the str() can be change according to values’ structure)  **df['New\_Column'] = df['Column'].str.strip('Dirty\_Sign')**  (Lastly we will fill the Nan values because of the lists, with the undirty values in the lists)  **df[“New\_Column”].fillna(df[“New\_Clmn\_for\_Lists”], inplace=True)** |
| **df["Column"].fillna(method='bfill', inplace=True)**  (To fill the missing values with proportion of other existing values) |
| **df["column"] = df["column"].str.replace("text", "")**  (To replace space to the “text” in the value for cleaning data. We can use this syntax again and again.) |
| **df["Column"].isnull().sum()/len(df["Column”])**  (To check the column’s NaN value percentage) |
| **df['Column'].str[0].str.contains("Dirty\_sign", na = False)**  (To cover both variables in the lists and out of the lists**,** “Dirty\_sign” has to be both in the list and out of the list. contains change the values to boolean)  **df["Column\_Bool"] = df['Column'].str[0].str.contains("Dirty\_sign", na = False)\*1**  (To change the values of columns to 1 or 2) |
| **nxtins\_list = ["".join(item).strip() if type(item) == list else item for item in df['Column']]**  ( To change the list to string if there are lists in the values)  **df['Column'] = pd.DataFrame(nxtins\_list)**  ( To replace the column with the new one that has no lists)  **df[“Column”].str[0].str.strip(“Dirty\_sign”)**  (To get the value out of list and to clean it) |
| **df["Column"] = df.Column.str.extract("(\d\d/\d\d\d\d)")**  ( To replace the values as MM/YYYY if value is suitable to that.) |
| **df.Column = pd.to\_datetime(df.Column, format = "%m/%Y")**  ( To change the MM/YYYY format to datetime, m has to be lowercase, Y has to be the uppercase) |
| **df.loc[df[“Columns”].isnull(), “Columns”]**  (condition : if the value is NaN ( True), then bring the columns’ related rows) |
| **df["Columns"].str.get\_dummies(sep = ",")**  (to seperated each item in the values)  **df = df.join(df["Columns"].str.get\_dummies(sep = "Seperator").add\_prefix("pp\_"))**  (to add each seperated items of values to your dataframes as new columns with prefixes defined by you) |
| **df['Columns'].sample(55)**  ( To investigate the sample of values (you can define the number))  **df['Columns'].str[0].value\_counts (dropna =False).sample(55)**  (for sampling the value\_counts of each first item of the values) |
| **variable = pd.to\_numeric(variable)**  (To change the object to float after clean the string value with str.extract()) |

|  |  |  |
| --- | --- | --- |
| 4. Start to analyze the variables which has below previously determined percentage of missing values.  a. You can fill the missing values with “bfill” or “ffill” methods.  b. You can drop the columns are asumed that they don’t have meaningfull effect on the result.  c. You can simplify the values by using str.replace() syntax to get the meaningfull number/data from the values.  d. You can fill the missing values with median.  e. You can replace the signs as multiply.  f. You can simplfy the value as combine the similar item into one name.  g. You can change some variables which do not make sense after investigating the the related domain. | **df.Column.fillna(method = "bfill", inplace = True)**  (to fill the missing values with bfill method) |  |
| **df.drop("Column", axis = 1, inplace = True)**  (To drop the the columns) |
| **df["Column"].str[0].str.strip("\n").str.replace("sign", "new\_sign")**  (To replace to a sign with another or space) |
| **df.Column.fillna(df.Column.median(), inplace = True)**  (To fill the missing values with median) |
| **df.replace({"Column" : {"Sign1" : "New\_Sign1", "Sign2 " : " New\_Sign2", "Sign3" : " New\_Sign3"}}, regex = True, inplace = True)**  (To replace the signs as multiply) |
| **var1 = df["Column"].str.contains("Name1", case = False, regex = True)**  **var2 = df["Column"].str.contains("Name2|Name3|Name4", case = False, regex = True)**  **df.loc[var1, "Column"] = "New\_Name1"**  **df.loc[var2, "Column"] = "New\_Name2"**  **var3 = list(df.Column.loc[lambda x : x != "New\_Name1"][lambda x : x != "New\_Name2"].index)**  **df.loc[var3, "Column"] = "New\_Name3"**  (To combine the similar names into one name) |
| **df.Column.iloc[index number] = "Real value"**  (To change the nonsense value with the correct one) |
| 5. After analyiznr and fix the all columns you can drop repeated columns you have. | **drop\_columns= ["Column1", "Column4", "Column23", ..... ]**  **df1 = df.drop(drop\_columns, axis = 1)**  ( To drop the columns you want to delete) |  |
| 6. You can write your file in a new file, so you can begin to next step.  Next step will be :  - Handling with Missing Values and Outliers,  - Continue to clean DataSet.  - Make the DataSet ready to go into ML Models. | **df1.to\_csv("file\_name.csv", index = False)**  (To save the new df) |  |

|  |  |  |
| --- | --- | --- |
| **STEP-3 : HANDLING WITH MISSING VALUES AND OUTLIERS** | | |
| **Instructions** | **Example Syntaxes** | **Example Functions** |
| 1. Read the previous cleared file to work on it. | **df1 = pd.read\_csv("file\_name.csv")**  (To read the file) |  |
| 2. Look at the columns which have any missing values. | **miss\_val = []**  **[miss\_val.append(i) for i in df1.columns if any(df1[i].isnull())]**  **miss\_val**  (To check the remaining missing values) |  |
| 3. Handle with Missing Values    a. In some cases, you will use again fill\_most function that we defined previusly. You can change the some values of columns ton np.nan after that fill them with the most repeated values in acoordance with the pivot column.  b. We can analyze the values by using Data Visiulation methods.  c. After you fill the some columns, you can anyway decide to drop them.  d. You can fill some NaN with using “transform” method. | **df1.Column[df1.Column == "Unusefull Value"] = np.nan**  **fill\_most("pivot Column", "Column")**  (To fill the unusefull value with the usefull one) | **def fill\_most(group\_col, col\_name):**  """  This function fills the missing values with the most frequent values groupby column 'group\_col'.  For example : based on 'make\_model' column, the function fill using .mode(), the most frequent value of variables.  """  **for group in list(oto1[group\_col].unique()):**  **grp\_inx=list(oto1[oto1[group\_col]==group][col\_name].index)**  **oto1[col\_name].iloc[grp\_inx]=oto1[col\_name].iloc[grp\_inx].fillna(oto1[oto1[group\_col]==group][col\_name].mode()[0])** |
| **df1.Column.value\_counts(dropna = False).plot(kind = "bar")**  (To plot a bar chart) |
| **df1['Column'] = df1['Column'].fillna(df1.groupby('Pivot Column')['Column'].transform('median'))**  (To fill the NaN with median) |
| 4. Handle with Outliers  a. Although some values are somehow categorical variables, we should clear it in terms of outliers.  b. Some columns contains numerical values, you can either consider them as categorical or continuous. When you implement get\_dummies(), you'll see the result.  c. The domain knowledge is important.  d. If there are meaningless values you can convert them into NaN s and then you can fill them. |  |  |

|  |  |  |
| --- | --- | --- |
| **STEP-4 : FINALIZE THE WORK** | | |
| **Instructions** | **Example Syntaxes** | **Example Functions** |
| 1. Re-inspect all features & Re-apply EDA | **df1.corr()**  (To check the correlation)  **var = df1[Numeric Columns]**  **plt.figure()**  **sns.heatmap(var.corr(), annot = True, cmap="RdYlGn", linewidths=0.2, annot\_kws = {"size": 8})**  ( To see the correlation heatmap of the numeric columns) |  |
| 2. Focus on the Target Column ( What is the aim of the EDA |  |  |
| 3Use pd.get\_dummies() method to get all values as numeric.  Next Step will be the running th Regression ML Model | **df2 = pd.get\_dummies(df1)** |  |