Data cleaning() Documentation

By: Asaf Ben-Menachem

Requirements:

1. Your computer needs to have python v3 installed.
2. Your computer need to have the following libraries installed:
   Pandas, Sklearn, matplotlib.pyplot ,seaborn

How to use:

1. The Script can be run using it from the code itself.
2. The script is located where the CSV file is.
3. All is needed is to write the name of the csv file without the ending of .csv in this varival FileName = 'test'

The flow of the function is using sub functions that have been called from the mean function (DataCleaning()).
Each SunFunction has a different purpose and is built to be as independent as possible from the other SubFunctions.

1. Data exploration is DataExploration()
2. Handling errors is ErrorsSolver()
3. Normalization is RemoveOutlier()
4. Standardization is Standardization () (not in use)
5. Remove columns that are highly correlated (note that different columns have a different type) is RemoveColumnsCorr()
6. Validation is with using in the end of the script DataExploration()

Also I added a way to fill the missing data using MissingData()

In order to debug the code and follow the changes and the mistakes that can happen there are a lot of try: except: sometimes also the user will get a error message about what happen and why.

1. ErrorsSolver- the purpose is to see in general if something in the dataset is unnecessary such as columns that had only Nan Values, Special characters.

1.1: If the dataset is smaller then 2 columns by 2 rows this dataset won't be relevant for ML this why the user will get back a message about this.

```
if 2 > len(df.index) or 2 > len(df.columns):  # limit the DataSet to Min of 2x2
    print("Error | The DataSet is too big or too small")  # print Error message
```

1.2: Renaming all the NA values to similar values.

The problem is that there are few different values that represent NA values. If we can track all of them down and give them the same value it can avoid feature problems.

```
df[column].replace(['nan','none','missing','nfc'], np.nan, inplace = True )
```

1.3: Even that the logic is to remove the special characters and to remove the columns that have a lot of NA in separate parts, because of the efficiency of the code is to combine into one part in the same for loop.

1.3.1

Remove some special characters that can make two similar cells to look unequal although they mean the same.

1.3.2

Remember how much missing values there is in each column and if the prossent of missing values in a column is bigger than 0.3 it will be reomoved from the data frame

```
spec_chars = "[!#%&()*+-./:;<=>?@[\\]^_`{|,},~,–]"  # This is the "bad characters

that I defined as not relevant in the data frame
for column in df.columns:
    try:
        df[column] = df[column].str.replace(spec_chars, "", regex=True)
    except:
        continue # there is no characters to change in this column
    df[column].replace(['nan','None','Missing','nfc'], np.nan, inplace = True )
    df.replace(' ', np.nan, regex=True)
    AvrNan = np.mean(df[column].isnull())
        # We get array of True/False from using isnull, so if we count how many True
        from all we will get the average of nan in all column
    if AvrNan > float(0.3):
        df.drop([column], axis=1)
```

1.4. After we hopefully remove all errors and the messy data we can remove all double columns and empty rows only for

```
    df = df.dropna(how='all')
            # Drop the rows where all elements are missing.
    df = df.dropna(axis='columns', how ='all')
            # Drop the columns where all elements are missing
```

2. MissingData(df)
This function will identify the type of each column, the function will convert the type of the column type to all values according to what type is most common.
For each type the function will fill the missing data with the value that is most likely not do the less hurm the ML model.

2.1: Go through all column and apply the value_count function but because we want The type of the value and we dont need to count how much from each value we can apply the type insdade of value by using apply(type).
And because we want the most frequent the index[0] will give it.

```
def MissingData(df):
    counter = int(0)
    newcolumns = df.columns.tolist() # getting all the Columns name
    for i in newcolumns: # take each column and work on it
        testingCol =df.iloc[:, counter]# Column data
        b = (df[i].apply(type).value_counts()).index[0] #getting the type of this column
        try:
            column1 =testingCol.astype(b, errors='raise') # converting the column to the right
type
            #print("the column type is change to " + str(b))
            column1.replace(['nan','None','Missing','nfc'], np.nan, inplace = True )
                    2.1.1: If its not possible to convert the series the only option is the values are
                                Numeric and this why will be converting to this.
                                Also the user will get a message about this.
        except:
            column1 = pd.to_numeric(testingCol, errors='coerce')
            print('Used to_numric()')
            2.2: getting the type of the column as a string for filtering the right method
                the type of the column that he contains.
                If it's not possible for some reason the user will get a Error message about it
                And the function will convert the column to string as a default.
        try:
            columnType = b.__name__ # getting the Column type as a string
        except:
            columnType = 'str'
            print('Error 2 | Had a problem with converting the column type so it was converted')
```

2.3: This part will fill the missing data in the columns.
According to the type of each column that we got in the last part of this Function the function can filter the column and implement the right methodologi for each type,
For Int the missing value will be replaced with median sinse it will be a full Number. It can be used only with numeric data.
For String the missing value will be replaced with the most frequent value Of each column.
For float the missing value will be replaced with the mean value Of each column.

```
        df[i] = column1
        CulomnsAsRow =df.iloc[:, counter:counter+1].values
        try:
            if columnType == 'str': # for each column type the Null will be field with diffrent
                imr = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
                imr = imr.fit(CulomnsAsRow)
                df[i] = imr.transform( CulomnsAsRow)

            elif columnType == 'int':
                imr = SimpleImputer(missing_values=np.nan, strategy='median')
                imr = imr.fit(CulomnsAsRow)
                df[i] = imr.transform( CulomnsAsRow)

            elif columnType == 'float':
                df[i] = df[i].fillna(df[i].mean())
```

2.3.1: The function was not able to fill the missing data, the user will get a message about it that the function did not recognize the type of this column

```
            else:
                print('Error3 | there is a new type that the script dont know how to handle ')
```

2.1.2: if for any reason the part of the function didnt work for this column the user Will get a message about it.

```
        except:
            print('Erroe | could not add values to the missing data at culomn: ' + str( i))
        counter = counter +1
    return (df)
```

3.DataExploration(df)
this function gives a visual view at the dataframe.
The concept is to try to make a plot (if possible).
This is why there are many try & Except.

Also the function give information about how many NaN there are in each column and from what type()

```
def DataExploration(df):
    print(df.info())
    print(df.head(n=3))
    for column in df.columns:
        print(column)
        try:
            df[column].value_counts().plot(kind='bar', figsize=(50,10))
            plt.show()

        except:
            continue
        try:
            df[column].plot.kde()
        except:
            continue

        try:
            df[column].plot.hist()
        except:
            continue

        try:
            print(df[column].apply(type).value_counts())
            df[column].apply(type).value_counts()
        except:
            continue
```

4. RemoveColumnsCorr()

This function removes high collinear columns in the dataframe with a correlation value bigger than the threshold.
Deleting collinear columns will help the model to generalize and improve the ML model.
The function gets a dataframe and a threshold and if the correlations are bigger than this value the column is removed.
The function returns a dataframe that contains only the highly collinear columns.
The threshold is 0.8

```
def RemoveColumnsCorr(df):
    mat = df.corr().abs()
    highCorr=np.where(mat>0.8)
    highCorr=[(mat.columns[x],mat.columns[y]) for x,y in zip(*highCorr) if x!=y and x<y]
    return df
```

5.RemoveOutlier()

The outliers indicate a mistake in data collection. It can influence a data set, so it's important to keep them out of the dataframe.
I will do this according to a popular method for cleaning exceptions

```
def RemoveOutlier(df):
    df1 = df.copy()
    df = df._get_numeric_data()

    low = df.quantile(0.25)
    high = df.quantile(0.75)
    dif = high - low
    lower_bound = low -(1.5 * dif)
    upper_bound = high +(1.5 * dif)


    for col in df.columns:
        for i in range(0,len(df[col])):
            if df[col][i] < lower_bound[col]:
                df[col][i] = lower_bound[col]

            if df[col][i] > upper_bound[col]:
                df[col][i] = upper_bound[col]

    for col in df.columns:
        df1[col] = df[col]

    return(df1)
```

6. Standardization
According to the flow we need to Standardization the data

```
def Standardization(df):
    for i in df: # take each column and work on it
        try:
            sc_x = StandardScaler()
            X_train = sc_x.fit_transform(X_train)
            X_test = sc_x.transform(X_test)
        except:
            print('Error | Function Normalization() | Column: ' + str(testingCol ))
    return df
```

7.DataCleaner()
This is the main function in the Task. According to the flow of DataCleaning, sometimes I will send a few times the dataset to a function because there is maybe a new action that this function can do that in the last time it couldn't do because of the change that the last function did.
Also using sub function we are able to do the validation step, using the DataExploration() we will see the chage that the function did to the dataframe.

```
def DataCleaner(FileName):

    locasion = str(FileName+ '.csv')
    newfileLoc = str(FileName+'_cleaned.csv')
    df = pd.read_csv(locasion)
    DataExploration(df)
    df = ErrorsSolver(df) #
    df =MissingData(df)
    #df = Standardization(df)
    df = RemoveColumnsCorr(df)
    df = ErrorsSolver(df)
    df = RemoveOutlier(df)
    df.to_csv(newfileLoc)
    DataExploration(df)
    return df
```

Main

Using these lines the script will run :)
Only what the user needs to do is change the file name and follow the “How to use” in the beginning of this documentation.

```
if __name__ == '__main__':
    FileName = 'tes'
    df = DataCleaner(FileName)
```

The End