Data Science: Data Cleansing And Visualization In Python For Beginners

I am writing this article to discuss a mini academic project work undertaken in Data Science. The project work was a collaborative effort with other fellow academics trying to learn the ropes of Data Science. The intention of this write up is to share the journey and outcome.

Data science is a fascinating discipline that is both artistic and scientific simultaneously. A project journey in Data Science involves extracting and gathering insightful knowledge from data that can either be structured or unstructured. The entire tour commences with data gathering and ends with exploring the data entirely for deriving business value, during which many procedures are applied systematically. Broadly speaking, the cleansing of the data, selecting the right algorithm to use on the data, and finally devising a machine learning function is the objective in this journey. The machine learning function derived is the outcome of this art that would solve the business problems creatively.

This article focuses exclusively on the Data analysis, cleansing, exploration, and imputation of data. I describe the steps that we undertook in this journey, forming the crux of this article.

Step 1. Import Libraries.

We started by importing the libraries that are needed to preprocess, impute, and render the data. The Python libraries that we used are Numpy, random, re, Matplotlib, Seaborn, and Pandas. Numpy for everything mathematical, random for random numbers, re for regular expression, Pandas for importing and managing the datasets, Matplotlib.pyplot, and Seaborn for drawing figures. These libraries are imported with a shortcut alias as below.

```
import numpy as np
import random
import re
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Step 2. Loading the data set.

We were handed over e-commerce data to explore. The data set was loaded using Pandas. Some necessary information about the data set was obtained using the 'info' method.

```
ecom = pd.read_csv('Ecommerce_Purchases.csv')
ecom.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Address           10000 non-null  object 
 1   Lot               10000 non-null  object 
 2   AM or PM          10000 non-null  object 
 3   Browser Info      10000 non-null  object 
 4   Company           10000 non-null  object 
 5   Credit Card       10000 non-null  int64  
 6   CC Exp Date       10000 non-null  object 
 7   CC Security Code  10000 non-null  int64  
 8   CC Provider       10000 non-null  object 
 9   Email             10000 non-null  object 
 10  Job               10000 non-null  object 
 11  IP Address        10000 non-null  object 
 12  Language          10000 non-null  object 
 13  Purchase Price    10000 non-null  float64
dtypes: float64(1), int64(2), object(11)
memory usage: 1.1+ MB
```

Step 3. Initial exploration of the data set.

This step involved exploring the various facets of the loaded data. This step helps in understanding the data set columns and also the contents.

```python
ecom['Purchase Price'].describe()
```

```
count    10000.000000
mean        50.347302
std         29.015836
min          0.000000
25%         25.150000
50%         50.505000
75%         75.770000
max         99.990000
Name: Purchase Price, dtype: float64
```

```python
# People who have English 'en' as their Language of choice on the website

ecom[ecom['Language']=='en'].count()
```

```
Address             1098
Lot                 1098
AM or PM            1098
Browser Info        1098
Company             1098
Credit Card         1098
CC Exp Date         1098
CC Security Code    1098
CC Provider         1098
Email               1098
Job                 1098
IP Address          1098
Language            1098
Purchase Price      1098
dtype: int64
```

```python
#count number of transacations in AM and PM
ecom['AM or PM'].value_counts()
```

```
PM    5068
AM    4932
Name: AM or PM, dtype: int64
```

```python
#top five jobs
ecom['Job'].value_counts().head(5)
```

```
Interior and spatial designer        31
Lawyer                               30
Social researcher                    28
Designer, jewellery                  27
Research officer, political party    27
Name: Job, dtype: int64
```

```python
#top five email providers
ecom['Email'].apply(lambda x: x.split('@')[1]).value_counts().head(5)
```

```
hotmail.com     1638
yahoo.com       1616
gmail.com       1605
smith.com         42
williams.com      37
Name: Email, dtype: int64
```

Step 3. Interpreting and transforming the data set.

In a real-world scenario, the data information that one starts with could be either raw or unsuitable for Machine Learning purposes. We will need to transform the incoming data suitably.

```
selected_columns = ecom[['AM or PM', 'Company', 'Credit Card',
       'CC Exp Date', 'CC Security Code', 'CC Provider', 'Email', 'Job',
       'IP Address', 'Language', 'Purchase Price']]

cleaned_ecom = selected_columns.copy()
```

```
#clean up address
address = ecom['Address'].str.extract(r",*\W*([A-Z]{2}\W+[0-9\-]*)$", expand = False)
address = address.str.split(pat=r"\W|\-", expand = True)
```

```
#clean browser
browser = ecom['Browser Info'].str.split(pat=r" |\(|\)", expand = True)
browser_and_ver = browser[0].str.split(pat=r"/", expand = True)
```

```
cleaned_ecom['Browser'] = browser_and_ver[0]
cleaned_ecom['Browser Version'] = browser_and_ver[1]
cleaned_ecom['CC Exp Year'] = ecom['CC Exp Date'].str.split(pat=r"/", expand = True)[1].astype(int)
cleaned_ecom['CC Exp Month'] = ecom['CC Exp Date'].str.split(pat=r"/", expand = True)[0].astype(int)
cleaned_ecom['State'] =   address[0]
cleaned_ecom['ZIP Code'] =   address[1]
cleaned_ecom['ZIP Code'] = cleaned_ecom['ZIP Code'].astype(int)
cleaned_ecom['CC Provider'] = cleaned_ecom['CC Provider'].str.split(pat = '\d', expand = True)[0]
cleaned_ecom.nunique() #There are 10000 different credit cards registered and 10000 different IP addresses
#but interestingly not 10000 different email addresses. Hence email address is not used to maintain user account
```

```
AM or PM                2
Company              8653
Credit Card         10000
CC Exp Date           121
CC Security Code     1758
CC Provider             8
Email                9954
Job                   623
IP Address          10000
Language                9
Purchase Price       6349
Browser                 2
Browser Version       181
CC Exp Year            11
CC Exp Month           12
State                  62
ZIP Code             9543
dtype: int64
```

We tried to drop any duplicate rows in the data set using the 'duplicate' method. However, as you would note below, the data set we received did not contain any duplicates.

```
cleaned_ecom = cleaned_ecom.drop_duplicates() #drop duplicates if any
cleaned_ecom
```

	AM or PM	Company	Credit Card	CC Exp Date	CC Security Code	CC Provider	Email	Job	IP Address	Language	Purchase Price	Br
0	PM	Martinez-Herman	6011929061123406	02/20	900	JCB	pdunlap@yahoo.com	Scientist, product/process development	149.146.147.205	el	98.14	
1	PM	Fletcher, Richards and Whitaker	3337758169645356	11/18	561	Mastercard	anthony41@reed.com	Drilling engineer	15.160.41.51	fr	70.73	
2	PM	Simpson, Williams and Pham	675957666125	08/19	699	JCB	amymiller@morales-harrison.com	Customer service manager	132.207.160.22	de	0.95	N
3	PM	Williams, Marshall and Buchanan	6011578504430710	02/24	384	Discover	brent16@olson-robinson.info	Drilling engineer	30.250.74.19	es	78.04	N
4	AM	Brown, Watson and Andrews	6011456623207998	10/25	678	Diners Club / Carte Blanche	christopherwright@gmail.com	Fine artist	24.140.33.94	es	77.82	
...	...	...	...	...	...	...	...	...	...	...	...	
9995	PM	Randall-Sloan	342945015358701	03/22	838	JCB	iscott@wade-garner.com	Printmaker	29.73.197.114	it	82.21	N
9996	AM	Hale, Collins and Wilson	210033169205009	07/25	207	JCB	mary85@hotmail.com	Energy engineer	121.133.168.51	pt	25.63	N
9997	AM	Anderson Ltd	6011539787356311	05/21	1	VISA	tyler16@gmail.com	Veterinary surgeon	156.210.0.254	el	83.98	N
9998	PM	Cook Inc	180003348082930	11/17	987	American Express	elizabethmoore@reid.net	Local government officer	55.78.26.143	es	38.84	N
9999	AM	Greene Inc	4139972901927273	02/19	302	JCB	rachelford@vaughn.com	Embryologist, clinical	176.119.198.199	el	67.59	N

10000 rows × 17 columns

```
cleaned_ecom.describe()
```

	Credit Card	CC Security Code	Purchase Price	CC Exp Year	CC Exp Month	ZIP Code
count	1.000000e+04	10000.000000	10000.000000	10000.000000	10000.00000	10000.000000
mean	2.341374e+15	907.217800	50.347302	21.173100	6.42570	49808.190700
std	2.256103e+15	1589.693035	29.015836	2.918114	3.46648	28965.375251
min	6.040186e+10	0.000000	0.000000	16.000000	1.00000	29.000000
25%	3.056322e+13	280.000000	25.150000	19.000000	3.00000	24745.000000
50%	8.699942e+14	548.000000	50.505000	21.000000	6.00000	49695.000000
75%	4.492298e+15	816.000000	75.770000	24.000000	9.00000	75011.250000
max	6.012000e+15	9993.000000	99.990000	26.000000	12.00000	99994.000000

Step 4. Imputing the data:

While looking for invalid values in the data set, it was soon determined that the data set was clean. The question placed on us in the project was to forcibly introduce errors at the rate of 10% overall if the data set supplied was clean. So given this ask, we decided to forcibly introduce errors into the data set. These errors could have been introduced in any column of the data set. Still, we limited introducing errors to just one row for studying imputing, which could have the maximum impact on the data set outcome. Thus, about 10% of the ‘Purchase Price’ data was randomly made as ‘numpy—NaN.’

I wrote a re-usable data frame impute class named ‘DataFrameWithImputor’ that has the following capabilities.

1. Be instantiated with a data frame as a parameter in the constructor.
2. Introduce errors to any numeric column of a data frame at a specified error rate.
3. Introduce errors across the data frame in any cell of the data frame at a specified error rate.
4. Impute error values in a column of the data set.
5. Find empty strings in rows of the data set.
6. Get the ’nan’ count in the data set.
7. Ability to describe the entire data set being imputed, whenever needed.
8. Ability to express any column of the data set being imputed, whenever required.
9. Perform forward fill on the entire data set.
10. Perform backward fill on the entire data set

The code for the impute class is produced below.

```python
class DataFrameWithImputor():
    def __init__(self,df):
        self.df=df

    def get_data_frame(self):
        return self.df


    #Randomly find indexes for x% of the column to populate with NaN values
    def introduce_errors(self, attribute, percent):
        column = self.df[attribute]
        error_data = int(column.size * percent)
        i = [random.choice(range(column.shape[0])) for _ in range(error_data)]
        column[i] = np.NaN
        self.df[attribute] = column
        return len(set(i)) # lengh of error indexes

    #Randomly find indexes for x% of the cells to populate with NaN values
    def introduce_errors_in_dataframe(self, percent):
        rows = len(self.df.index)
        error_data = int(rows * percent)
        columns = len(self.df.columns)
        for i in range(error_data):
            col = i  % columns
            row = i % rows
            self.df.iloc[row,col] = np.NaN
        return self.df.isnull().sum().sum()



    def impute(self,column,value):
        #Impute NaN values in the column with a random value

        null_values = self.df[self.df[column].isnull()].index

        for i in range(len(null_values)):
            self.df[column][null_values]  = value

        col_description = pd.DataFrame(self.df[column].describe())
        col_description.loc['Frequent'] = self.df[column].value_counts().idxmax()
        return col_description
```

```python
    def get_nan_count(self):
        return self.df.isnull().sum()


    def find_empty_string(self):
        return np.where(self.df.applymap(lambda x: x == '')) # return rows with empty string

    def nan_values_in_column(self):
        return np.where(pd.isnull(self.df)) #return indexes for null values in a row

    def describe(self):
        return self.df.describe

    def describe_col(self, col):
        desc = pd.DataFrame(self.df[col].describe())

        desc.loc['Frequent'] = self.df[col].value_counts().idxmax()
        return desc

    def fillforward(self):
        self.df = self.df.fillna(method='ffill',axis = 0)


    def fillbackward(self):
        self.df = self.df.fillna(method='bfill',axis = 0)
```

The un-imputed data set was checked for any Nan or missing strings for one final time before forcibly introducing errors.

```python
unimputed = DataFrameWithImputor(cleaned_ecom)
if len(unimputed.find_empty_string()) > 2:
    print('Empty strings in the data frame')

if len(unimputed.nan_values_in_column()) > 2:
    print('NaN in the data frame')
```

A helper function 'do*impute' was defined to introduce errors in a column of the data set and impute the data set column afterwords. This function would take a condition parameter to perform imputation.*

```python
def do_imputation(df,column,error_rate,condition = None  ):

       imp = DataFrameWithImputor(df.copy())

       imp.introduce_errors(column,error_rate)

       if condition != None:
           imp.impute(column,condition)
       else:
           #Impute through backfill and forwardfill
           imp.fillbackward()
           imp.fillforward()

       return imp
```

Various imputation techniques were performed into the data set to simulate real-world scenarios. The error rate at 10% was randomly instituted for all the methods into the clean data set column 'Purchase Price.'

The approach has been that since missing data is the most common in a data set and takes NaN or None.

There are several ways to fill up missing values:

1. We can remove the missing value rows itself from the data set. However, in this case, the error percentage being low at just 10%, so this was not undertaken.
2. Filling the null cell in the data set column with a constant value.
3. Filling the invalid section with mean and median values
4. Filling the null area with a random value
5. Filling null using data frame backfill and forward fill

The above are some of the common strategies applied to impute the data set. However, there are no limits to designing a radically different approach to the data set imputation itself.

```
#####
#  Mean imputation
#####

mean_imputed = do_imputation(cleaned_ecom,'Purchase Price',.1, cleaned_ecom['Purchase Price'].mean())


#####
#  Median imputation
#####
#Impute NaN values in the column with the median


median_imputed = do_imputation(cleaned_ecom,'Purchase Price',.1, cleaned_ecom['Purchase Price'].median())

#####
#  Random imputation
#####

random_imputed = do_imputation(cleaned_ecom,'Purchase Price',.1,random.choice(range(1,99)))

#####
#  Impute with constant
#####

const_imputed = do_imputation(cleaned_ecom,'Purchase Price',.1, 50)

#Forward and backward fill to impute data
#
#
fill_imputed = do_imputation(cleaned_ecom,'Purchase Price',.1)
```

Each of the imputed outcomes was studied separately—the fill (backfill and forward fill) and constant value imputation outcome are shown below.

```
fill_imputed.describe_col('Purchase Price')
```

	Purchase Price
count	10000.000000
mean	50.327816
std	29.071102
min	0.000000
25%	25.060000
50%	50.570000
75%	75.860000
max	99.990000
Frequent	76.530000

```
const_imputed.describe_col('Purchase Price')
```

	Purchase Price
count	10000.000000
mean	50.302625
std	27.581357
min	0.000000
25%	27.750000
50%	50.000000
75%	72.990000
max	99.990000
Frequent	50.000000

The median and random value imputations are shown below.

```
median_imputed.describe_col('Purchase Price')
```

	Purchase Price
count	10000.000000
mean	50.450384
std	27.641255
min	0.000000
25%	27.797500
50%	50.505000
75%	73.205000
max	99.990000
Frequent	50.505000

```
random_imputed.describe_col('Purchase Price')
```

	Purchase Price
count	10000.000000
mean	50.542415
std	27.575375
min	0.000000
25%	27.927500
50%	53.000000
75%	72.862500
max	99.990000
Frequent	53.000000

The mean imputation being compared with the un-imputed data set column below.

```
mean_imputed.describe_col('Purchase Price')
```

	Purchase Price
count	10000.000000
mean	50.445186
std	27.662944
min	0.000000
25%	27.930000
50%	50.347302
75%	73.162500
max	99.990000
Frequent	50.347302

```
unimputed.describe_col('Purchase Price')
```

	Purchase Price
count	10000.000000
mean	50.347302
std	29.015836
min	0.000000
25%	25.150000
50%	50.505000
75%	75.770000
max	99.990000
Frequent	49.730000

From the above techniques, mean imputation was found closer to the un- imputed clean data, thus preferred. Other choices such as fill(forward and backward) also seemed to produce data set column qualitatively very close to clean data from the study above. However, the mean imputation was preferred as it gives a consistent result and a more widespread impute technique.

```
df = mean_imputed.get_data_frame()
```

The data frame adopted for further visualisation was the mean imputed data set.

Step 5. Exploring and Analysing the data: A cleaned up and structured data is suitable for analyzing and finding exemplars using visualization.

Conclusion

<<TBD>>