FIT5196-S2-2020 ASSESSMENT 2

Student Name: Ananya Pandey

Student ID: 30757924

Date: 18/10/2020

Environment: Python 3.7.3.final.0 and Anaconda 4.8.3

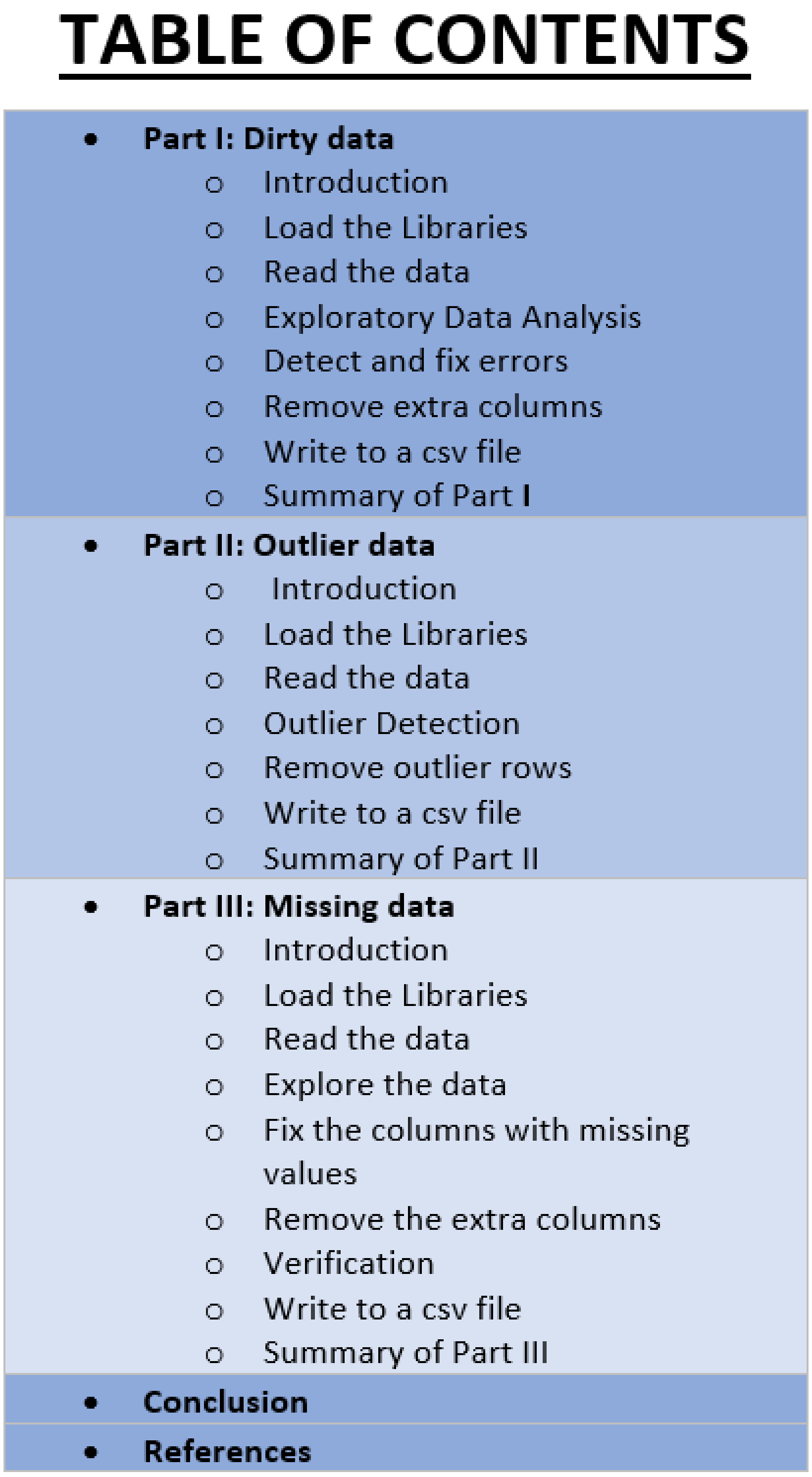
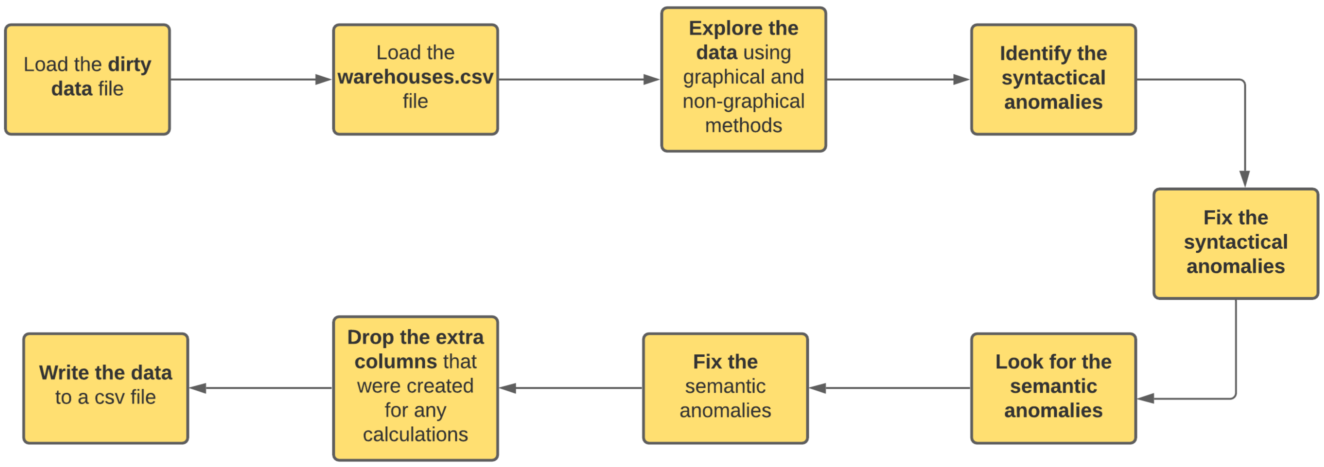


TABLE OF CONTENTS

Part I: DIRTY DATA

Sequence of Activities:


Load the dirty data file
Load the warehouses.csv file
Explore the data using graphical and non-graphical methods
Identify the syntactical anomalies
Fix the syntactical anomalies
Look for the semantic anomalies
Fix the semantic anomalies
Drop the extra columns that were created for any calculations
Write the data to a csv file

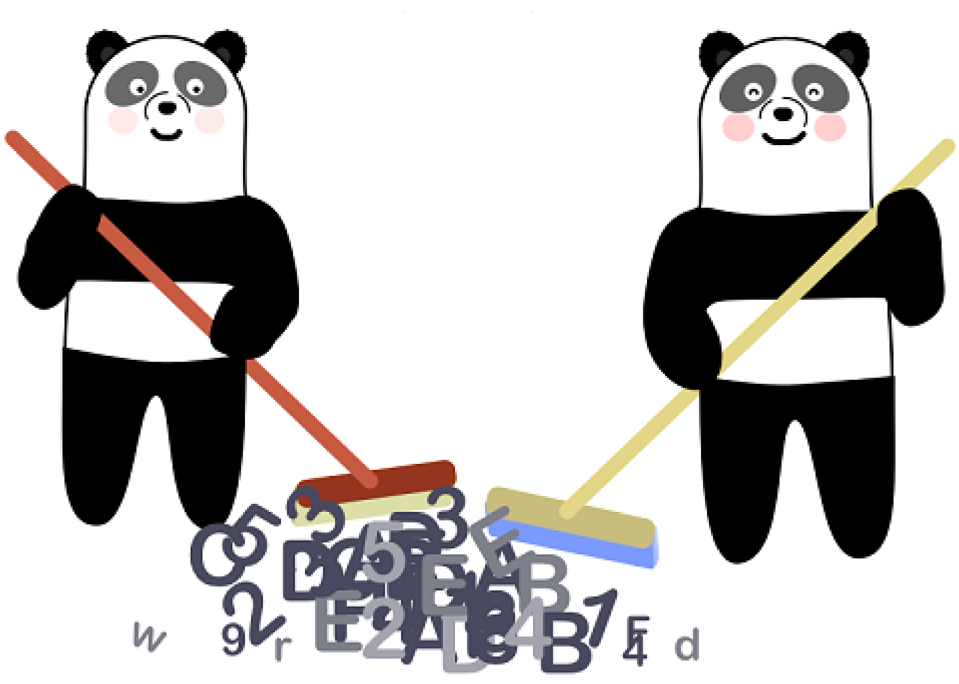
Libraries used:

- pandas 1.0.3 (for data manipulation and analysis, included in Python 3.7.3)
- numpy 1.16.2 (for working with arrays, included in Python 3.7.3)
- itertools (includes a set of functions for working with iterable (sequence-like) data sets, included in Python 3.7.3)
- sklearn (it features various classification, regression and clustering algorithms, included in Python 3.7.3)
- matplotlib 3.0.3 (is a plotting library for the Python programming language and its numerical mathematics extension numpy)
- seaborn 0.9.0 (is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics)
- matplotlib.cbook (a collection of utility functions and classes that can be imported from anywhere within matplotlib)
- re 2.2.1 (for regular expression, included in Anaconda Python 3.7.3)
- nltk 3.5 (the Natural Language Toolkit, is a suite of libraries and programs for symbolic and statistical natural language processing for English written in python)
- functools (it is for higher-order functions that act on or return other functions)

1. Introduction

Data cleaning is the process of detecting and correcting the inaccurate records from a table, or database and refers to identifying incorrect or inaccurate parts of the data and then replacing, modifying, or deleting the dirty data.

Data cleaning is also important because it improves your data quality and in doing so, increases overall productivity. When the data is cleaned, all outdated or incorrect information is gone – leaving us with only correct and good quality information.



Data Anomalies describes the distortion of the data because of any of the problems that might encounter in the life cycle of data that includes its capture, storage, update, transmission, access, archive, restore, deletion and purge.

In this task, the data has to be explored and graphical and/or non-graphical EDA methods are used to understand the data. The data has certain anomalies that have to be identified and fixed.

2. Load the libraries

The required libraries are imported.

In [ ]:

```
import pandas as pd
import numpy as np
from numpy import linalg
import matplotlib.pyplot as plt
import re
from datetime import datetime
import operator
import functools
from itertools import permutations
import sklearn
from sklearn.linear_model import LinearRegression
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

3. Read the data

The first thing is to inpect the `dirty_data.csv` and `warehouses.csv` files and figure out their format. To read the csv file the **read_csv()** function is used.

In [ ]:

```
#load the dirt_data.csv file
dirty_data = pd.read_csv('30757924_dirty_data.csv')
```

In [ ]:

```
#load the warehouses.csv file
warehouses = pd.read_csv('warehouses.csv')
```

In [ ]:

```
#show the warehouses data
warehouses.head()
```

In [ ]:

```
#show top 10 rows of dirt_data
dirty_data.head()
```

4. Exploratory Data Analysis

Exploratory Data Analysis is an approach to analyze data sets to summarize their main characteristics. It can be done using info() and describe() functions or by a visual method as well.

4.1. Non-graphical method

Here, `.shape` tells about the current dimensions of the array. It can be seen that the dirty_data file has 500 data entries i.e. rows and 16 columns. `info()` function gives a very short summary of the entire data. It shows a list of all columns with their data types and the number of non-null values in each of the column.

Numerical Variables:

- order_price,
- delivery_charges,
- customer_lat,
- customer_long,
- coupon_discount,
- order_total,
- distance_to_nearest_warehouse

Categorical Variables:

- order_id
- customer_id
- date
- nearest_warehouse
- shopping_cart
- season
- latest_customer_review

is_expedited_delivery and is_happy_customer are boolean variables.

In [ ]:

```python
#check number of rows and columns of dirty_data
dirty_data.shape
```

In [ ]:

```python
#check the non-null entries and the data types of each column
dirty_data.info()
```

In [ ]:

```python
#check distribution of numerical variables across the data
dirty_data.describe()
```

The `describe()` function shows the distribution of the numerical variables in the data.

The observation are:

- There are total 500 samples so all the orders are unique
- The delivery charges varied from 48.17 to 112.27.
- There is certain inconsistency in the customer latitude and longitude values as it can be seen that customer latitude is maximum at 145 which should not be the case and similarly the minimum customer longitude is -37 which shouldn't be either.
- Coupon discount varies from 0% i.e. no discount to 25% discount.
- Order total varies significantly with some paying as high as $39223.
- Distance to nearest warehouse lies between 0.01 - 2.87.

In [ ]:

```python
#check distribution of categorical variables across the data
dirty_data.describe(include=['O'])
```

`.describe(include=['O'])` shows the distribution for the categorical variables.

The table shows:

- order_id is unique across the dataset
- customer_id's are not unique across the dataset
- neraest_warehouse variable with 6 values seem suspicious
- Season variable with 8 values seems suspicious

In [ ]:

```python
#check for duplicates in the customer_id and date column
dirty_data[dirty_data.duplicated(["customer_id","date"], keep=False)]
```

In [ ]:

```python
#check for duplicates in the order_id column
dirty_data[dirty_data.duplicated(["order_id"], keep=False)]
```

4.2. Graphical method

In this section, the attributes of the data are explored individually using a graphical method in order to understand more about those particular variables in the data.

As it can be seen from the first graph which shows the `delivery_charges` plotted as a histogram, delivery charges vary from around 50 to around 115.

Then, the `distance_to_nearest_warehouse` column is analysed to see the variability in the distances between the customer and the nearest warehouse.

Then, a bar plot is made showing the `nearest_warehouse` column. It can be seen that there are 6 warehouses. The last bar plot shows the `season` column and it can be seen that there are 8 seasons.

In [ ]:

```
#plot delivery_charges as a histogram
plt.hist(dirty_data['delivery_charges'])
plt.title('Histogram for delivery charges')
plt.xlabel("Delivery Charges")
plt.ylabel('Frequency')
#show the plot
plt.show()
```

In [ ]:

```
#plot distance to nearest warehouse as a histogram
plt.hist(dirty_data['distance_to_nearest_warehouse'])
plt.title('Histogram for distance to nearest warehouse')
plt.xlabel("Distance to Nearest Warehouse")
plt.ylabel('Frequency')
#show the plot
plt.show()
```

In [ ]:

```
#plot the nearest_warehouse column as a bar chart
dirty_data['nearest_warehouse'].value_counts().plot(kind='bar')
```

In [ ]:

```
#plot the season column
dirty_data['season'].value_counts().plot(kind='bar')
```

5. Detect and fix errors

Data cleansing is a process of detecting and removing errors and inconsistencies from data in order to improve the quality of data. Thses errors are the data anomalies.

Data Anomalies describes the distortion of the data because of any of the problems that might encounter in the life cycle of data that includes its capture, storage, update, transmission, access, archive, restore, deletion and purge. Many problems can be faced due to data anomalies:

- Missing data
- Inconsistent and faulty data
- Outliers

- Duplicated values

Data Anomalies are of three types:

1. `Syntactical Anomalies` : These are the errors that include format issues and value issues. So, any type of lexical error or domain format errors is a type of syntactical anomaly due to inconsistency and any type of format discrepencies in the data. Any type of irregularities i.e. the non-uniform use of values, units and abbreviations is also a form of syntactical anomaly.
2. `Semantic Anomalies` : These are the errors that includes comprehensiveness issues and non-redundancy problems in the data. Variables can be correlated with each other. One variable might provide information that we can use to validate another variable. These types of anomalies include integrity constraint violations, contradictions, duplicates and invalid tuples.
3. `Coverage Anomalies` : decrease the amount of entities and entity properties from the mini-world that are represented in the data collection. Coverage anomalies are categorised as missing values and missing tuples.

5.1. Identify and rectify the Syntactical Anomalies

`Syntactical Anomalies` : These are the errors that include format issues and value issues. So, any type of lexical error or domain format errors is a type of syntactical anomaly due to inconsistency and any type of format discrepencies in the data. Any type of irregularities i.e. the non-uniform use of values, units and abbreviations is also a form of syntactical anomaly.

5.1.1. Error in `nearest_warehouse` column

As per the business rules, the retail store has three different warehouses in Melbourne. With the help of exploratory data analysis, it is found that there are six warehouses. Typos and inconsistent spelling are the most common errors, particularly whenever the data collection process involves human. While the data is collected, some places have the nearest_warehouse name has inconsistent spelling. It is always a good idea to check the categorical variables to make sure their values are spelled without errors.
The `value_counts()` function and the `unique()` function is used in order to identify the inconsistency in the spellings. The inconsistent spelling error has been fixed by replacing the names of the 'nearest_warehouse' with the correct spelling i.e.

- $'bakers'$ is replaced with $'Bakers'$
- $'nickolson'$ is replaced with $'Nickolson'$
- $'thompson'$ is replaced with $'Thompson'$

After replacing the names of the 'nearest_warehouse' with the correct spelling, the values in the nearest_warehouse column are verified if they have been transformed properly in the data or not and that the nearest_warehouse column has consistent values with three warehouses i.e. Bakers, Nickolson and Thompson.

In [ ]:

```
dirty_data['nearest_warehouse'].value_counts()
```

In [ ]:

```
#check for unique nearest_warehouse names
dirty_data.nearest_warehouse.unique()
```

In [ ]:

```
#replace the incorrect spellings with the correct ones
dirty_data = dirty_data.replace({'nearest_warehouse' : { 'bakers' : 'Bakers', 'nickols
```

In [ ]:

```
#the transformations are verified
dirty_data['nearest_warehouse'].value_counts()
```

5.1.2. Error in `season` column

As it is known, 'season' is the column showing the season in which the order has been placed. With the help of exploratory data analysis, it is found that there are eight warehouses. Typos and inconsistent spelling are the most common errors, particularly whenever the data collection process involves human. While the data is collected, some places have the season name has inconsistent spelling. It is always a good idea to check the categorical variables to make sure their values are spelled without errors.
The `value_counts()` function and the `unique()` function is used in order to identify the inconsistency in the spellings. The inconsistent spelling error has been fixed by replacing the names of the 'season' with the correct spelling i.e.

- $'summer'$ is replaced with $'Summer'$
- $'winter'$ is replaced with $'Winter'$
- $'autumn'$ is replaced with $'Autumn'$
- $'spring'$ is replaced with $'Spring'$

After replacing the names of the 'season' with the correct spelling, the values in the season column are verified if they have been transformed properly in the data or not and it is checked that the column has four values i.e. Summer, Winter, Autumn and Spring.

In [ ]:

```
#check for the values in the season column
dirty_data['season'].value_counts()
```

In [ ]:

```
dirty_data.season.unique()
```

In [ ]:

```
#replace the incorrect spellings with the correct ones
dirty_data = dirty_data.replace({'season' : { 'winter' : 'Winter', 'summer' : 'Summer'
```

In [ ]:

```
#the transformations are verified
dirty_data.season.unique()
```

In [ ]:

```
dirty_data['season'].value_counts()
```

5.1.3 Error in `date` column

As per the business rules, 'date' column tells about the date the order was made and it should be in YYYY-MM-DD format.

On observing the dates in the data column, it is found that in certain rows the date is not in a proper format. On exploring the data in the 'date' column, it is found that certain rows are in DD-MM-YYYY format or in YYYY-DD-MM format.

To find anomalies in the 'date' column, the middle term of the date which supposedly should be the month is extracted and checked if it is between 1 to 12. If it is greater than 12, then for those rows the 'date' has to be fixed by converting it into a proper format of YYYY-MM-DD. Another thing that is checked is that whether the date is in YYYY-MM-DD format or not, if it is not then also the date has to be converted to the proper format of YYYY-MM-DD.

In order to fix the identified anomaly in the date column, first the dates in the 'date' column are converted to a datetime format using the pandas library function `.to_datetime()` if it is not in a proper format and then a `strptime()` function used to parse the date according to the format specified i.e. YYYY-MM-DD because for strptime() the dates need to match, otherwise the `strftime()` function is called in order to format the result in the desired final format.

The 'date' column is then verified after the formatting in order to check if all the dates are in the correct format.

In [ ]:

```
#check for incorrect dates in the date column
months_list=[] #to store the dates with incorrect months
year_list=[] #to store the dates with incorrect formats/year
dates_list = dirty_data['date'].tolist()
for item in range(len(dates_list)):
    if int(dates_list[item][5:7])>12:
        months_list.append(dates_list[item])
    elif str(dates_list[item][0:4])!='2019':
        year_list.append(dates_list[item])
    else:
        continue

print(months_list)
print(year_list)
```

In [ ]:

```python
#fix the problems in the date column and convert them into proper format
for i in range(len(dirty_data['date'])):
    try:
        dirty_data['date'][i]=pd.to_datetime(dirty_data['date'][i])
    except:
        pass

    try:
        dirty_data['date'][i]=datetime.strptime(dirty_data['date'][i], '%Y-%m-%d',dayf
    except:
        dirty_data['date'][i]=pd.to_datetime(dirty_data['date'][i], format='%Y-%d-%m')
```

In [ ]:

```python
#check if the transformations have been applied correctly and the dates are in proper
months_list=[]
year_list=[]
dates_list = dirty_data['date'].tolist()
for item in range(len(dates_list)):
    if int(dates_list[item][5:7])>12:
        months_list.append(dates_list[item])
    elif str(dates_list[item][0:4])!='2019':
        year_list.append(dates_list[item])
    else:
        continue

print(months_list)
print(year_list)
```

5.2. Identify and fix the Semantic Anomalies

`Semantic Anomalies` : These are the errors that includes comprehensiveness issues and non-redundancy problems in the data. Variables can be correlated with each other. One variable might provide information that we can use to validate another variable.
In this task, the following constraints are checked whether they are being followed or not:

- violate the integrity constraints
- contradictions: violation of dependencies between attributes
- duplications: observations representing the same entity.
- invalid observations

5.2.1. Error in `season` names.

In Australia, the seasons are defined by grouping the calendar months in the following way:

- Spring - the three transition months September(09), October(10) and November(11).
- Summer - the three hottest months December(12), January(01) and February(02).
- Autumn - the transition months March(03), April(04) and May(05).
- Winter - the three coldest months June(06), July(07) and August(08).

This shows that the 'season' is correlated with the month in which the order is placed. The month in which the order is placed can be determined from the 'date' on which the order is placed.

First, each row in the data is checked in order to find the errors in the season column. After splitting the data using the split() function, the months are checked according to the seasons and a new list is created for each of the seasons having the wrong entries i.e. if the month is December, January or February and the season is not Summer, then the incorrect values are added in the list for that season in order to understand which incorrect values are present.
It is noticed that in the columns where the season should be Summer, Spring and Winter is present in the data. The columns where the season should be Winter, Autumn and Summer is also present in some of the columns. For Autumn season, the rest of the seasons i.e. Spring, Winter and Summer are present in some of the columns where the season should be Autumn. For some of the rows where the season should be Spring, Autumn season is present instead of it.

The `split()` function applied on the dates in order to get the year, month and date separately for each of the orders i.e. for each row.
If the month is December, January or February, the season is Summer, if the month is March, April or May, the season is Autumn, if the season is June, July or August, the season is Winter and if the month is September, October or November, the season is Spring. The 'season' column is thus corrected according to the months in which the orders are placed.

The 'season' column is then verified after the seasons have been corrected according to the months in which the orders have been placed.

In [ ]:

```
#lists to store the incorrest season in the season column according to the date
summer_list=[]
winter_list=[]
autumn_list=[]
spring_list=[]
for i in range(len(dirty_data)):
    year,month,date=dirty_data['date'][i].split('-')
    if((month == '12' or month == '01' or month == '02') and (dirty_data['season'][i]!
        summer_list.append(dirty_data['season'][i])
    if((month == '03' or month == '04' or month == '05') and (dirty_data['season'][i]!
        autumn_list.append(dirty_data['season'][i])
    if((month == '06' or month == '07' or month == '08') and (dirty_data['season'][i]!
        winter_list.append(dirty_data['season'][i])
    if((month == '09' or month == '10' or month == '11') and (dirty_data['season'][i]!
        spring_list.append(dirty_data['season'][i])

print(summer_list)
print(winter_list)
print(autumn_list)
print(spring_list)
```

In [ ]:

```
#change the season according to the month that can be extracted from the date column
for i in range(len(dirty_data)):

    year,month,date=dirty_data['date'][i].split('-')
    if(month == '12' or month == '01' or month == '02'):
        dirty_data['season'][i]='Summer'
    if(month == '03' or month == '04' or month == '05'):
        dirty_data['season'][i]='Autumn'
    if(month == '06' or month == '07' or month == '08'):
        dirty_data['season'][i]='Winter'
    if(month == '09' or month == '10' or month == '11'):
        dirty_data['season'][i]='Spring'
```

In [ ]:

```
#verify if the transformations have been properly made
summer_list=[]
winter_list=[]
autumn_list=[]
spring_list=[]
for i in range(len(dirty_data)):
    year,month,date=dirty_data['date'][i].split('-')
    if((month == '12' or month == '01' or month == '02') and (dirty_data['season'][i]!
        summer_list.append(dirty_data['season'][i])
    if((month == '03' or month == '04' or month == '05') and (dirty_data['season'][i]!
        autumn_list.append(dirty_data['season'][i])
    if((month == '06' or month == '07' or month == '08') and (dirty_data['season'][i]!
        winter_list.append(dirty_data['season'][i])
    if((month == '09' or month == '10' or month == '11') and (dirty_data['season'][i]!
        spring_list.append(dirty_data['season'][i])

print(summer_list)
print(winter_list)
print(autumn_list)
print(spring_list)
```

5.2.2 Error in `customer_lat` & `customer_long` column values

'customer_lat' is the column showing the customer's latitude. 'customer_long' is the column showing the customer's longitude.

Firstly, as most of the rows have 'customer_lat' around -37 and 'customer_long' around 144, the rows not having such values are checked i.e. the rows having positive latitude is checked. On analysing the data, it is found that in certain rows the values between the customer latitude and customer longitude are swapped to each other i.e. the latitude is the longitude and the longitude is the latitude.

In order to fix this problem, the values of 'customer_lat' and 'customer_long' are swapped to each other for the rows having this issue.

Lastly, the 'customer_lat' and 'customer_long' columns are checked to see and verify that the values have swapped correctly and the values of the customer latitude and customer longitude is correct i.e. latitude is negative and longitude is positive in this case.

In [ ]:

```
dirty_data[dirty_data.customer_lat>0]
```

In [ ]:

```
#swap the values of customer_lat and customer_long for which they are incorrectly plac
for i in range(len(dirty_data)):
    if(dirty_data['customer_lat'][i] > 0):
        tmp = dirty_data['customer_lat'][i]
        dirty_data['customer_lat'][i] = dirty_data['customer_long'][i]
        dirty_data['customer_long'][i] = tmp
```

In [ ]:

```
#check if all the latitudes are positive
dirty_data[dirty_data.customer_lat>0]
```

In [ ]:

```
#check if all the longitudes are negative
dirty_data[dirty_data.customer_long<0]
```

5.2.3. Error in `distance_to_nearest_warehouse` values

In this section, the distance to the nearest warehouse attribute is fixed by correcting the wrong values in this column.
Firstly, we calculate the haversine distance using the haversine function, this function calculates the distance between two points for which the latitude and the longitude is known. The radius of the earth is taken as 6378 km.

New columns,

`dist_from_N` which is the distance from Nickolson warehouse,
`dist_from_T` which is the distance from Thompson warehouse and
`dist_from_B` which is the distance from Bakers warehouse, are created.

These columns are made in order to find the distance between each customer and the warehouse using the latitudes and longitudes of both the customer and the warehouses.

Then the minimum of the three distances is calculated in order to find the distance to the nearest warehouse and its values are stored in a new column named `minimum_dist` .
Lastly, the values in the 'distance_to_the_nearest_warehouse' column are fixed using the calculated values.

The values in the 'distance_to_the_nearest_warehouse' column are then verified in order to check whether the calculated values have been applied properly or not.

In [ ]:

```python
#the haversine distance is calculated to find the distance between two geo-locations
def haversine_distance(lat1, long1, lat2, long2):
    r = 6378 #radius of earth
    phi1 = np.radians(lat1)
    phi2 = np.radians(lat2)
    delta_phi = np.radians(lat2 - lat1)
    delta_lambda = np.radians(long2 - long1)
    a = np.sin(delta_phi / 2)**2 + np.cos(phi1) * np.cos(phi2) *   np.sin(delta_lambda
    res = r * (2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a)))

    return np.round(res, 4)#result is rounded to 4 decimal places
```

In [ ]:

```python
dist_from_N=[]#list to store customer's distance from Nickolson
dist_from_T=[]#list to store customer's distance from Thompson
dist_from_B=[]#list to store customer's distance from Bakers
for i in dirty_data.index:
    dist_from_N.append(haversine_distance(warehouses['lat'][0], warehouses['lon'][0],
    dist_from_T.append(haversine_distance(warehouses['lat'][1], warehouses['lon'][1],
    dist_from_B.append(haversine_distance(warehouses['lat'][2], warehouses['lon'][2],

#column to store customer's distance from Nickolson
dirty_data['dist_from_N'] = dist_from_N
#column to store customer's distance from Thompson
dirty_data['dist_from_T'] = dist_from_T
#column to store customer's distance from Bakers
dirty_data['dist_from_B'] = dist_from_B
```

In [ ]:

```python
#find the minimum of the three distances
new = dirty_data[['dist_from_N', 'dist_from_T', 'dist_from_B']].min(axis=1)
#store the minimum distance in a new column
dirty_data['minimum_dist'] = new
```

In [ ]:

```python
dirty_data['distance_to_nearest_warehouse']=dirty_data['minimum_dist']
```

In [ ]:

```python
dirty_data.loc[~(dirty_data['distance_to_nearest_warehouse'] == dirty_data['minimum_di
```

5.2.4. Error in `nearest_warehouse` names

In this section, the nearest warehouse attribute is fixed by correcting the values in this column.
Using the minimum distance that was calculated in the previous section, the nearest warehouse can be found by checking for the minimum distance.

If the minimum distance was 'dist_from_N', then the nearest warehouse is Nickolson.
If the minimum distance was 'dist_from_T', then the nearest warehouse is Thompson.
If the minimum distance was 'dist_from_B', then the nearest warehouse is Bakers.

A new column 'nearest_retail' is made in order to store the name of the nearest warehouse which can be found from the minimum distance column. Thses values are then stored in the 'nearest_warehouse' column.

The values in the 'nearest_warehouse' column are then verified in order to check whether the calculated values have been applied properly or not.

In [ ]:

```python
#list to store the nearest warehouse names
nearest_retail=[]
for i in dirty_data.index:
    if(dirty_data['minimum_dist'][i]==dirty_data['dist_from_N'][i]):
        nearest_retail.append('Nickolson')
    if(dirty_data['minimum_dist'][i]==dirty_data['dist_from_T'][i]):
        nearest_retail.append('Thompson')
    if(dirty_data['minimum_dist'][i]==dirty_data['dist_from_B'][i]):
        nearest_retail.append('Bakers')

#column to store nearest warehouse names
dirty_data['nearest_retail'] = nearest_retail
```

In [ ]:

```python
dirty_data['nearest_warehouse']=dirty_data['nearest_retail']
```

In [ ]:

```python
dirty_data.loc[~(dirty_data['nearest_warehouse'] == dirty_data['nearest_retail'])]
```

5.2.5. Error in `order_price` values

In order to fix the order price column, the `outlier_data.csv` file has been used. As there are no anomalies in the outlier data file and it is also generated on the same dataset, it is better to use that file to find the correct price of each of the 10 branded items that are sold at competitive prices. In this section, the order price of each of the orders is calculated by:

$$OrderPrice = \sum quantity * price$$

Firstly, the outlier data file is read using the pandas library .read_csv() function. Then, the products for each of the rows' shopping cart is determined by applying a regular expression on the shopping cart values. The regular expression catches everything lies between the `(` and the `'` . Then, another regular expression is applied on the shopping cart in order to get the quantitites of each of the product for that order. The regular expression catches everything lies between the `,` and the `)` . Then, a list of all the 10 branded items that the retail store sells them at competitive prices is made. Now, the quantity of each of the product for each order is determined according to the products in the unique 10 products list i.e.

products = ['Alcon 10', 'Candle Inferno', 'Lucent 330S', 'Olivia x460', 'Thunder line', 'Toshika 750', 'Universe Note', 'iAssist Line', 'iStream', 'pearTV']

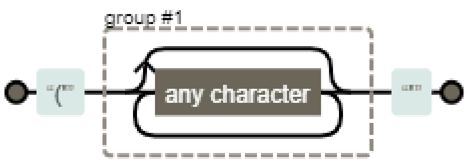
product_quantity = [0, 1, 0, 0, 1, 2, 0, 0, 1, 0] So, this product_quantity tells, that there is 1 Candle Inferno, 1 Thunderline, 2 Toshika 750, 1 iSteeam.

For finding the price of each item, numpy modules' `numpy.linalg()` function is used in order to implement linear algebra algorithms. In order to calculate the prices of 10 items, ten rows have been sampled from the orders for which the nearest warehouse is Bakers. `linalg.solve()` is used to solve 10 linear equations in order to get the price of the ten individual products.

In [ ]:

```
#load the outlier data file using read_csv()
outlier_data = pd.read_csv('30757924_outlier_data.csv')
```

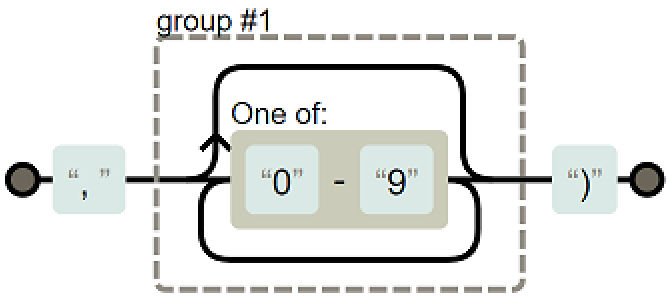
REGULAR EXPRESSION FOR GETTING EACH PRODUCT IN THE CART:



In [ ]:

```
#get the item names from the shopping cart
regex_items = r'(?:\(\')(.*?)\''
dirty_data['products'] = dirty_data['shopping_cart'].apply(lambda x: re.findall(regex_
outlier_data['products'] = outlier_data['shopping_cart'].apply(lambda x: re.findall(re
```

REGULAR EXPRESSION FOR GETTING THE QUANTITY FOR EACH PRODUCT IN THE CART:



In [ ]:

```
#get item quantity from the shopping cart
regex_quantity = r', ([0-9]*?)\)'
dirty_data['quantity'] = dirty_data['shopping_cart'].apply(lambda x: re.findall(regex_
outlier_data['quantity'] = outlier_data['shopping_cart'].apply(lambda x: re.findall(re
```

In [ ]:

```
#get the list of products
products = []
product_list = outlier_data['products'].to_list()
for i in range(len(product_list)):
    for j in range(len(product_list[i])):
        products.append(product_list[i][j])
```

In [ ]:

```
#get list of unique 10 products
list_of_products = sorted(list(set(products)))
```

In [ ]:

```python
#function to find the quantity of each product
def quantity_of_each_product(value):

    quants = value['quantity'] #get the quantities
    prods = value['products'] #get the products
    final_products_list = []

    for prod in list_of_products:
        if prod in prods:
            i = int(quants[prods.index(prod)])
            final_products_list.append(i)
        else:
            final_products_list.append(0)

    return final_products_list
```

In [ ]:

```python
#a new column to store the quantities of the products
outlier_data['product_quantity']=outlier_data.apply(quantity_of_each_product,axis=1)
```

In [ ]:

```python
#in order to get the prices of each of the 10 items, sample of 10 rows which have Bake
A = outlier_data[outlier_data["nearest_warehouse"] == "Bakers"].sample(n=10)
#the product_quantity for theses rows is taken
B = np.array(A["product_quantity"].to_list())
```

In the following cell, as a sample of 10 rows which have Bakers as the nearest_warehouse is taken and the product_quantity of these rows is considered. Determinant is to be checked an if it is zero then only the process to find the price of each item can be continued. This is done in order to get the 10 samples in which each item is present once and that each item has a coefficient.

In [ ]:

```python
while np.linalg.det(B) == 0:
    A = outlier_data[outlier_data["nearest_warehouse"] == "Bakers"].sample(n=10)
    B = np.array(A["product_quantity"].to_list())
```

In [ ]:

```python
#find the price of the order for those rows that have been sampled
price_of_order = np.array(A["order_price"].apply(lambda x: float(x)).to_list())
```

In [ ]:

```python
#use linalg.solve() to find the individual prices
X = np.linalg.solve(B, price_of_order)
```

In [ ]:

```python
X #prices of each of the items
```

In [ ]:

```
final_price=list(X) #list to store the prices of the items
```

In a similar way, the product quantity is also calculated for the dirty data and this column tells about the quantity of the productsaccording the products index in the products list.

In [ ]:

```
#list storing the quantity of the product
prod_quant = []
for i in range(len(dirty_data['order_id'])):
    prod_quant.append(quantity_of_each_product(dirty_data.iloc[i]))
dirty_data['product_quantity'] = prod_quant
```

In [ ]:

```
dirty_data['product_quantity']
```

Now, the order price is calculated for the entire order by multiplying the quantity of the product to the price of the product and adding all the values for each order. So, a `new_order_price` column is created in order to store the correct values of the order price. In order to calculate the new_order_price, the `reduce()` function has been used to apply a particular function passed in its argument, in this case operator.mul to multiply the quantity and the price to all of the list of elements.

In [ ]:

```
#function to calculate the order price
def sum_of_products(value):

    prod_quants = value['product_quantity']
    final_price_list = final_price
    s = 0
    for data in zip(prod_quants, final_price_list):
        s+=functools.reduce(operator.mul, data)


    return s
```

In [ ]:

```
new_price_of_order = [] #initialise an empty list to store the order prices
for i in range(len(dirty_data['order_id'])):
    new_price_of_order.append(sum_of_products(dirty_data.iloc[i]))
dirty_data['new_order_price'] = new_price_of_order
```

In [ ]:

```
new_order_price_list = dirty_data['new_order_price'].to_list()
```

Here, the `order_total` is calculated from the given order total in the dirty_data dataset.

$$OrderTotal = OrderPrice - (Discount * OrderPrice)/100 + DeliveryCharges$$

A new column `total_from_old_price` is created which stores the order total for the all the orders using the given order prices.

But as the the given order prices can be incorrect, in order to check for the order total, again a new column `total_from_new_price` is created which stores the order total for all the orders using the new_order_price that has been calculated by using the correct price of each of the item that is calculated using the outlier_data which has no data anomalies.

In [ ]:

```
#function to calculate order total from the given order price
def order_total_old_price(value):

    order_price = value['order_price']
    delivery_charge = value['delivery_charges']
    discount = value['coupon_discount']
    result = (order_price - ((discount * order_price)/100)) + delivery_charge

    return round(result,2) #return the result rounded off to 2 decimal places
```

In [ ]:

```
total = [] #initialize an empty list
for i in range(len(dirty_data['order_id'])):
    total.append(order_total_old_price(dirty_data.iloc[i]))
dirty_data['total_from_old_price'] = total
```

In [ ]:

```
order_total_list_old = dirty_data['total_from_old_price'].to_list()
```

In [ ]:

```
#function to calculate order total from the new calculated order price
def order_total_new_price(value):

    order_price=value['new_order_price']
    delivery_charge=value['delivery_charges']
    discount=value['coupon_discount']
    result=(order_price - ((discount * order_price)/100)) + delivery_charge

    return round(result,2)
```

In [ ]:

```
total = [] #initialize an empty list
for i in range(len(dirty_data['order_id'])):
    total.append(order_total_new_price(dirty_data.iloc[i]))
dirty_data['total_from_new_price'] = total
```

In [ ]:

```
order_total_list_new = dirty_data['total_from_new_price'].to_list() #list to store ord
order_price_list = dirty_data['order_price'].to_list()#list to store the order prices
order_total_list = dirty_data['order_total'].to_list()#list to store the order total o
```

In the following cell, it is checked that if the order_price for the orders is same as the new_order_price calculated or is it different. If it is different, it means that the order_price that is given is wrong and the new_order_price needs to be replaced with the older one for that row. The indices of such rows are stored in check1.

In [ ]:

```
check1 = [] #initialize an empty list
for val in range(len(order_price_list)):
    if order_price_list[val] != new_order_price_list[val] and order_total_list[val] ==
        check1.append(val)#append the index of the item to the list if the condition i
    else:
        continue
```

In [ ]:

```
#replace the incorrect values with the correct ones in the data
for values in check1:
    dirty_data['order_price'][values] = dirty_data['new_order_price'][values]
```

5.2.6. Incorrect `order _total` values

Here, using the above calculations, the `order_total` column is fixed.

In the following cell, it is checked that if the order_price for the orders is same as the new_order_price calculated and whether the order total that is calculated is the same as the order total calculated from the given order price. If the order total is different, it means that the order_price is correct but the order_total is wrong. To fix this the indices with the wrong order total are fixed using the 'total_from_old_price' column for that row. The indices of such rows are stored in the check2 list.

In [ ]:

```
check2 = [] #initialize an empty list
for val in range(len(order_price_list)):
    if order_price_list[val] == new_order_price_list[val] and order_total_list[val] !=
        check2.append(val) #append the index of the item to the list if the condition 
    else:
        continue
```

In [ ]:

```
#replace the incorrect values with the correct ones in the data
for values in check2:
    dirty_data['order_total'][values] = dirty_data['total_from_old_price'][values]
```

5.2.7. Incorrect `shopping_cart` items

In the following cell, it is checked for each order whether the order_price is the same as the calculated new_order_price and that the order_total is the same as the order total calculated from the given order price. This means that there are some glitches in the shopping cart.

A list having the wrong product quantities is made to store the `product_quantity` for the orders for which the cart is incorrect.

Similarly, a list having the wrong order price is created to store the incorrect order prices of the orders for which the shopping cart is incorrect.

In [ ]:

```
check3 = []  #initialize an empty list
for val in range(len(order_price_list)):
    if order_price_list[val] != new_order_price_list[val] and order_total_list[val] ==
        check3.append(val)#append the index of the item to the list if the condition i
    else:
        continue
```

In [ ]:

```
product_quantity_list=dirty_data['product_quantity'].tolist()
```

In [ ]:

```
wrong_product_quantity = [] #the list of product_quantity for which the cart is incorre
for i in check3:
    #append the item to the list for which the cart is incorrect
    wrong_product_quantity.append(product_quantity_list[i])
```

In [ ]:

```
len(wrong_product_quantity)
```

In [ ]:

```
wrong_order_price = [] #the list of order_price for which the cart is incorrect
for i in check3:
    #append the order_price to the list for which the cart is incorrect
    wrong_order_price.append(order_price_list[i])
```

In the following cell, for the `wrong_product_quantity` values, its permutations are taken in order to get all the possible ways in which the cart can be designed using the product quantities that are given.

A combined list `list_of_perms` having the possible permutations of the wrong_product_quantity is created.

In [ ]:

```
perm = [] #list of permutations
for d in wrong_product_quantity:
    perm.append(permutations(d))
len(perm)
```

In [ ]:

```
list_of_perms = []#initialize an empty list
for i in perm:
    list_of_perms.append(i)
```

Now, for the items in the wrong_product_quantity it is checked by multiplying the quantity and the price to all of the list of elements in order to get the order_price according to that permutation. If the order_price calculated for that particular permutation equals the order price for the order, it means that this permutation is correct.

In [ ]:

```python
#function to find the sum of the multiplied values i.e. sum of products
def sum_product(a,b):

    s = 0
    for data in zip(a, b):
        s += functools.reduce(operator.mul,data)

    return s
```

In [ ]:

```python
right_list=[] #initialize an empty list
for item in range(len(wrong_product_quantity)):

    perms_list = list(list_of_perms[item])
    for val in perms_list:

        check = sum_product(val, final_price)
        #check if the order_price calculated for that particular permutation equals the
        if check == wrong_order_price[item]:
            right_list.append(val)
            break;

len(right_list)
```

Finally, using the `right_list` created in the previous step, the list of products and the right_list containing the quantities are zipped together. If the quantity is not zero, then the product and their quantities are combined in a tuple to get the final shopping_cart for that order.

Now this right_list is then replaced at the indices at which the 'shopping_cart' is wrong in order to get the correct `shopping_cart` .

In [ ]:

```python
final_replace_list = []
for y in right_list:

    list_of_vals = list(y)
    #zip to make a tuple of the products and their quantities
    correct_list = list(zip(list_of_products, list_of_vals))
    #keep only the products which have a quantity and the ones with 0 are not appended
    final_correct_list = [(x,y) for x,y in correct_list if y != 0]
    final_replace_list.append(final_correct_list)
```

In [ ]:

```python
#replace the caert items for the indices on which the cart is incorrect
dirty_data.loc[check3, "shopping_cart"] = final_replace_list
```

5.2.8. Incorrect `is_happy_customer` values

In this section, the is_happy_customer attribute is fixed by checking whether a customer is happy with their last order, the customer's latest review is classified using a sentiment analysis classifier. To check whether a customer is happy with their last order, the customer's latest review is classified using a sentiment analysis classifier SentimentIntensityAnalyzer from nltk.sentiment.vader which is used to obtain the polarity score. A sentiment is considered positive if it has a 'compound' polarity score of 0.05 or higher and is considered negative otherwise.

A new column 'sents' is created in order to store the sentiments of the customer based on whether the customer is happy with his previous order or not using SentimentIntensityAnalyzer from the nltk.sentiment.vader module. Then the values of this column are used to fix the anomalies in the 'is_happy_customer' attribute.

In [ ]:

```python
#define the object
sentiment_analyser = SentimentIntensityAnalyzer()
sents=[] #initialize an empty list
for i in range(len(dirty_data['latest_customer_review'])):
    if (sentiment_analyser.polarity_scores(dirty_data['latest_customer_review'][i])['c
        sents.append(True) #if the condition is true i.e. the review is positive so th
    else:
        sents.append(False)
#make a new column and put the values of the sents list in the column
dirty_data['sents']=sents
```

In [ ]:

```python
dirty_data['is_happy_customer'] = dirty_data['sents']
```

5.2.9. Incorrect `is_expedited_delivery`

For fixing the data issues in the `is_expedited_delivery` column, `missing_data` file has been used as it just has missing data and no other data anomalies are present in this data.

As per the specifications provided, the store has different business rules depending on the season to match the different demands of each season. The delivery charge is calculated using a linear model which differs depending on the season. The model depends linearly (but in different ways for each season) on:

1. Distance between customer and nearest warehouse
2. Whether the customer wants an expedited delivery
3. Whether the customer was happy with his/her last purchase (if no previous purchase, it is assumed that the customer is happy)

In order to apply linear regression, all the rows from 'is_expedited_delivery', 'distance_to_nearest_warehouse', 'is_happy_customer' and 'delivery_charges' having non-null values are considered.

So, the model for delivery_charges depends on distance_to_nearest_warehouse, is_expedited_delivery and is_happy_customer attributes. This is the reason why these columns have been filtered for the prediction of delivery_charges for each of the season i.e. summer, winter, autumn and spring.

In the following section, the original outlier dataframe has been filtered according to the seasons i.e. 'season'== Summer/Winter/Autumn/Spring and four new dataframes for each of the seasons have been created i.e. summer_df, winter_df, autumn_df, spring_df.

Now, the predictor variables i.e. 'is_expedited_delivery', 'distance_to_nearest_warehouse' and 'is_happy_customer attributes are taken from each of the season's dataframes i.e. summer_cols, winter_cols, autumn_cols and spring_cols.

The predicted variable i.e. the delivery charges attribute is extracted from each of the data and then 4 dataframes i.e. summer_delivery, winter_delivery, autumn_delivery and spring_delivery are created.

In [ ]:

```
#read the missing_data file using pandas library read_csv() function
missing_data = pd.read_csv('30757924_missing_data.csv')
```

In [ ]:

```
#rows having null in 'distance_to_nearest_warehouse' are not not considered
data_linreg = missing_data[pd.notna(missing_data['distance_to_nearest_warehouse'])]
```

In [ ]:

```
data_linreg.info()
```

In [ ]:

```
#rows having null in 'is_happy_customer' are not not considered
data_linreg = data_linreg[pd.notna(data_linreg['is_happy_customer'])]
```

In [ ]:

```
data_linreg.info()
```

In [ ]:

```
#rows having null in 'delivery_charges' are not not considered
data_linreg = data_linreg[pd.notna(data_linreg['delivery_charges'])]
```

In [ ]:

```
data_linreg.info()
```

In [ ]:

```
#rows having null in 'is_expedited_delivery' are not not considered
data_linreg = data_linreg[pd.notna(data_linreg['is_expedited_delivery'])]
data_linreg.info()
```

In [ ]:

```
summer_df = data_linreg[data_linreg.season=='Summer']
summer_cols = summer_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
summer_delivery = summer_df[['delivery_charges']]
```

In [ ]:

```
winter_df = data_linreg[data_linreg.season=='Winter']
winter_cols = winter_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
winter_delivery = winter_df[['delivery_charges']]
```

In [ ]:

```
autumn_df = data_linreg[data_linreg.season=='Autumn']
autumn_cols = autumn_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
autumn_delivery = autumn_df[['delivery_charges']]
```

In [ ]:

```
spring_df = data_linreg[data_linreg.season=='Spring']
spring_cols = spring_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
spring_delivery = spring_df[['delivery_charges']]
```

Using the `sklearn.linear_models` package, a LinearRegression model is made so that the data can be fitted on it for the calculation of the the coefficients and the intercepts of the model for further usage.

The linear model is then fitted on the training data and the target values. Training data i.e. the predictor variables which are summer_cols, winter_cols, autumn_cols and spring_cols that contain the attributes 'is_expedited_delivery', 'distance_to_nearest_warehouse' and 'is_happy_customer'.

Now, the coefficients of the models are calculated for each of the model using `.coef_` and the intercepts of each of the model is calculated using `.intercept_` .

In [ ]:

```
model = LinearRegression()
summer_model = model.fit(summer_cols, summer_delivery) #fit the model
summer_coef = model.coef_ #coefficients of the model
summer_intercept = model.intercept_ #intercept of the model
print(summer_coef)
print(summer_intercept)
```

In [ ]:

```
model = LinearRegression()
winter_model = model.fit(winter_cols, winter_delivery) #fit the model
winter_coef = model.coef_ #coefficients of the model
winter_intercept = model.intercept_ #intercept of the model
print(winter_coef)
print(winter_intercept)
```

In [ ]:

```
model = LinearRegression()
autumn_model = model.fit(autumn_cols, autumn_delivery) #fit the model
autumn_coef = model.coef_ #coefficients of the model
autumn_intercept = model.intercept_#intercept of the model
print(autumn_coef)
print(autumn_intercept)
```

In [ ]:

```
model = LinearRegression()
spring_model = model.fit(spring_cols, spring_delivery) #fit the model
spring_coef = model.coef_ #coefficients of the spring_model
spring_intercept = model.intercept_ #intercept of the model
print(spring_coef)
print(spring_intercept)
```

The `is_happy_customer` , `is_expedited_delivery` and `distance_to nearest_warehouse` are copied to new columns with numeric data type for calculations, so that the original columns remain undisturbed. The new columns that are created are `is_happy_customer_num` , `is_expedited_delivery_num` and `distance_to nearest_warehouse_num` .

Now, the delivery charges are calculated using the general linear regression equationin the calculate_delivery function:

$y = \beta_0 * x_0 + \beta_1 * x_1 + \beta_2 * x_2 + c$
where,

- β_0, β_1 and β_2 are the coefficients
- c is the intercept

Thus for every model the delivery charges are calculated in this way:

$delivery$ = β_0∗(is_expedited_delivery_num) + β_1∗(distance_to_nearest_warehouse_num) + β_2∗(is_happy_customer_num) + $intercept$

A new column `new_calculated_delivery` is created by applying the calculate_delivery function on each row of the data. Then, the difference between the calculated delivery charges and the given delivery charges is found and a new column `delivery_difference` is made to store these values.

In [ ]:

```
#numerical columns of the following columns used for easier calculations
dirty_data['is_happy_customer_num'] = dirty_data['is_happy_customer'].astype(int)
dirty_data['is_expedited_delivery_num'] = dirty_data['is_expedited_delivery'].astype(i
dirty_data['distance_to_nearest_warehouse_num'] = dirty_data['distance_to_nearest_warel
```

In [ ]:

```
#calculate delivery charges using linear regression model
def calculate_delivery(value):
    if value['season'] == 'Summer':
        delivery = summer_coef[0][0] * value['is_expedited_delivery_num'] + summer_coe
    elif value['season'] == 'Winter':
        delivery = winter_coef[0][0] * value['is_expedited_delivery_num'] + winter_coe
    elif value['season'] == 'Autumn':
        delivery = autumn_coef[0][0] * value['is_expedited_delivery_num'] + autumn_coe
    else:
        delivery = spring_coef[0][0] * value['is_expedited_delivery_num'] + spring_coe
    return np.round(delivery, decimals=2)
```

In [ ]:

```
#a new column to store the new calculated delivery charges
dirty_data['new_calculated_delivery'] = dirty_data.apply(calculate_delivery, axis = 1)
```

In [ ]:

```
#difference between the calculated delivery charges and the given delivery charges is 
dirty_data['delivery_difference'] = dirty_data['new_calculated_delivery'] - dirty_data
```

In order to check if the order had an expedited delivery or not, a threshold value of 10 is taken. If the delivery difference is too high, it means it is an expedited_delivery else it is not. If the delivery difference is greater than the threshold value of 10, it is an expedited_delivery else it is not.

A new column `is_expedited_delivery_num` is created in which these 1's and 0's are stored which represent whether it is an expedited delivery or not.

Another column `new_expedited` is made which stores these numeric values as a bollean value of True or False.

Finally, the `is_expedited_delivery` column is fixed using this new_expedited column.

In [ ]:

```
#function to check whether the delivery is expedited or not
def check_if_is_expedited(item):

    if abs(item['delivery_difference']) > 10:#condition for the threshold value
        return 1

    else:
        return item['is_expedited_delivery_num']
```

In [ ]:

```
dirty_data['new_expedited'] = dirty_data.apply(check_if_is_expedited, axis = 1).astype
```

In [ ]:

```
#check for the counts of unique values of this column
dirty_data['new_expedited'].value_counts()
```

In [ ]:

```
#check for the counts of unique values of this column
dirty_data['is_expedited_delivery'].value_counts()
```

In [ ]:

```
dirty_data[dirty_data['is_expedited_delivery'] != dirty_data['new_expedited']]
```

In [ ]:

```
dirty_data['is_expedited_delivery'] = dirty_data['new_expedited']
```

6. Remove the extra columns

Some extra columns have been made in this task of cleaning this dirty data file. These columns are to be removed in order to make the data consistent and in the same format as the original dirty_data file.

In [ ]:

```
cols = ['dist_from_N', 'dist_from_B', 'dist_from_T', 'minimum_dist', 'nearest_retail',
dirty_data = dirty_data.drop(cols, 1)
```

In [ ]:

```
dirty_data.info()
```

7. Write to a csv file

The dataframe from which the outlier rows have been removed is then written to a csv file named "30757924_dirty_data_solution.csv" using the **to_csv()** function of the pandas library.

In [ ]:

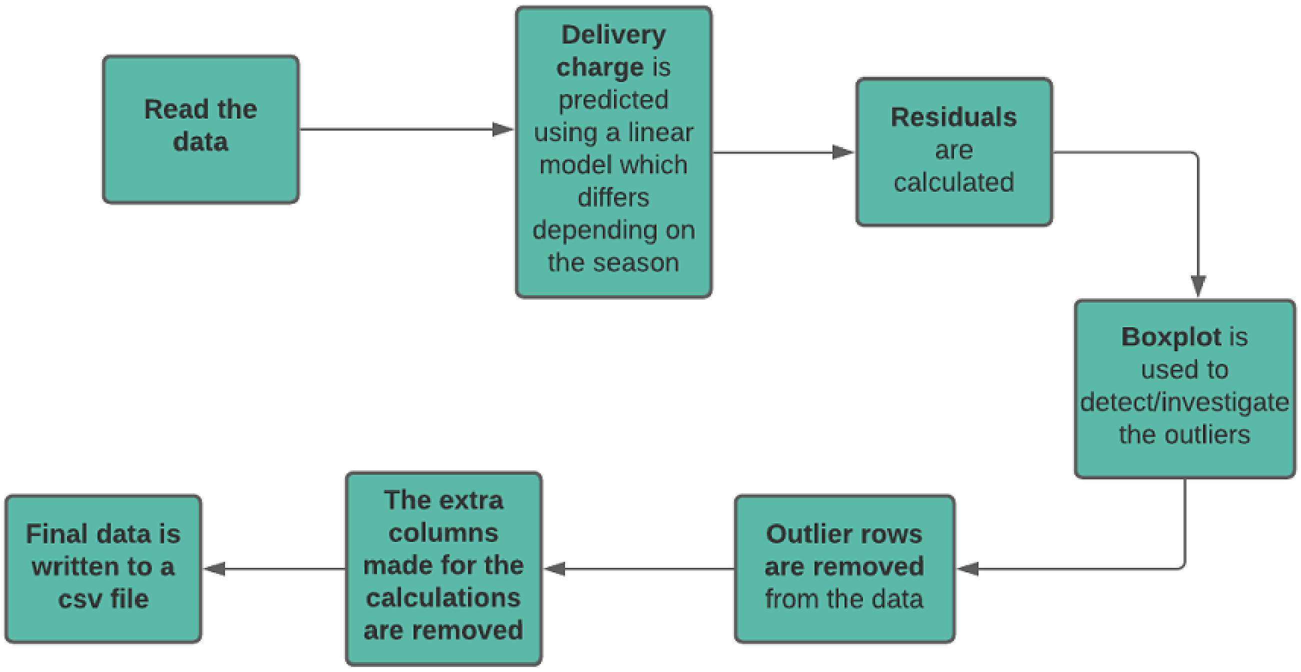
```
dirty_data.to_csv('30757924_dirty_data_solution.csv', index=False)
```

8. Summary of Part 1

- In this task, the data has been explored and graphical and non-graphical EDA methods have been used to understand the data.
- The syntactical anomalies i.e. the the anomalies related to the format and values are identified and then fixed accordingly.
- The semantic anomalies i.e. the anomalies related to the comprehensiveness and non-redundancy of the data are identified and then fixed accordinly.

Part II: OUTLIER DATA

Sequence of activities:


Read the data
Delivery charge is predicted using a linear model which differs depending on the season
Residuals are calculated
Boxplot is used to detect/investigate the outliers
Outlier rows are removed from the data
The extra columns made for the calculations are removed
Final data is written to a csv file

Libraries used:

- pandas 1.0.3 (for data manipulation and analysis, included in Python 3.7.3)
- numpy 1.16.2 (for working with arrays, included in Python 3.7.3)
- itertools (includes a set of functions for working with iterable (sequence-like) data sets, included in Python 3.7.3)
- sklearn (it features various classification, regression and clustering algorithms, included in Python 3.7.3)
- matplotlib 3.0.3 (is a plotting library for the Python programming language and its numerical mathematics extension numpy)
- seaborn 0.9.0 (is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics)
- matplotlib.cbook (a collection of utility functions and classes that can be imported from anywhere within matplotlib)

1. Introduction

“An outlier is an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism” (Hawkins, D. 1980. Identification of Outliers. Chapman and Hall.)

Outliers are data points that differ significantly from the other data points and observations. It lies at an abnormal distance from other values in the data. Thus it can cause serious problems in statistical analyses. This is the reason for the data set to not have rows having outliers and should be removed.

In this task, the outlier rows have to be detected and then removed. The outliers are to be found with respect to the delivery_charges column only.

2. Load the libraries

The required libraries are imported.

In [ ]:

```
import pandas as pd
import numpy as np
from numpy import linalg
import sklearn
from sklearn.linear_model import LinearRegression
from matplotlib.cbook import boxplot_stats
import seaborn as sns
from matplotlib import pyplot
import matplotlib.pyplot as plt
from itertools import chain
%matplotlib inline
```

3. Read the data

The first thing is to inpect the file and figure out the file format. To read the csv file the **read_csv()** function is used.

In [ ]:

```
#load the outlier_data to a dataframe
outlier_data = pd.read_csv('30757924_outlier_data.csv')
```

In [ ]:

```
outlier_data.head()
```

4. Outlier Detection

4.1. Filter the data according to the seasons

As stated in the business rules, the retail stores' delivery charges depend linearly i.e. it is calculated using a linear model which depends on the season to match the different demands of each season. The model depends linearly (but in different ways for each season) on:

1. Distance between customer and nearest warehouse
2. Whether the customer wants an expedited delivery
3. Whether the customer was happy with his/her last purchase (if no previous purchase, it is assumed that the customer is happy).

So, the model for delivery_charges depends on distance_to_nearest_warehouse, is_expedited_delivery and is_happy_customer attributes. This is the reason why these columns have been filtered for the prediction of delivery_charges for each of the season i.e. summer, winter, autumn and spring.

In the following section, the original outlier dataframe has been filtered according to the seasons i.e. 'season'== Summer/Winter/Autumn/Spring and four new dataframes for each of the seasons have been created i.e. summer_df, winter_df, autumn_df, spring_df.

Now, the predictor variables i.e. 'is_expedited_delivery', 'distance_to_nearest_warehouse' and 'is_happy_customer attributes are taken from each of the season's dataframes i.e. summer_cols, winter_cols, autumn_cols and spring_cols.

The predicted variable i.e. the delivery charges attribute is extracted from each of the data and then 4 dataframes i.e. summer_delivery, winter_delivery, autumn_delivery and spring_delivery are created.

The summer_items, winter_items, autumn_items and spring_items are the lists having the indices or the positions of the rows with respect to each of the weather.

In [ ]:

```
#get the rows for which the season is Summer
summer_df = outlier_data[outlier_data.season=='Summer']
summer_cols = summer_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
summer_delivery = summer_df[['delivery_charges']]
summer_items = summer_df.index.tolist() #list to store indices of thr rows in which th
```

In [ ]:

```
#get the rows for which the season is Winter
winter_df = outlier_data[outlier_data.season=='Winter']
winter_cols = winter_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
winter_delivery = winter_df[['delivery_charges']]
winter_items = winter_df.index.tolist() #list to store indices of thr rows in which th
```

In [ ]:

```
#get the rows for which the season is Autumn
autumn_df = outlier_data[outlier_data.season=='Autumn']
autumn_cols = autumn_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
autumn_delivery = autumn_df[['delivery_charges']]
autumn_items = autumn_df.index.tolist() #list to store indices of thr rows in which th
```

In [ ]:

```
#get the rows for which the season is Spring
spring_df = outlier_data[outlier_data.season=='Spring']
spring_cols = spring_df[['is_expedited_delivery', 'distance_to_nearest_warehouse','is_
spring_delivery = spring_df[['delivery_charges']]
spring_items = spring_df.index.tolist() #list to store indices of thr rows in which th
```

4.2. Fit the Linear Regression model for delivery_charges

Using the `sklearn.linear_models` package, a LinearRegression model is made so that the data can be fitted on it for the prediction of delivery_charges and for further usage.

The linear model is then fitted on the training data and the target values. Training data i.e. the predictor variables which are summer_cols, winter_cols, autumn_cols and spring_cols that contain the attributes 'is_expedited_delivery', 'distance_to_nearest_warehouse' and 'is_happy_customer'.

Target values, which here is the value that has to be predicted, in this case, the delivery_charges i.e. summer_delivery, winter_delivery, autumn_delivery and spring_delivery.

In [ ]:

```
model = LinearRegression()
```

In [ ]:

```
#the models are fitted for all the four seasons
summer_model = model.fit(summer_cols, summer_delivery)
winter_model = model.fit(winter_cols, winter_delivery)
autumn_model = model.fit(autumn_cols, autumn_delivery)
spring_model = model.fit(spring_cols, spring_delivery)
```

4.3. Predict the delivery_charges values

The models i.e. the summer_model, winter_model, autumn_model and spring_model are then used to predict the delivery_charges for each of them. The predicted values are then appended in a new column predicted_delivery_charges according to the indices of the values for that particular row.

In [ ]:

```
#a new column to store the predicted values
outlier_data['predicted_delivery_charges'] = 0
```

In [ ]:

```
#delivery charges are predicted for each of the fitted model
outlier_data.loc[summer_items, 'predicted_delivery_charges'] = summer_model.predict(su
outlier_data.loc[winter_items, 'predicted_delivery_charges'] = winter_model.predict(wi
outlier_data.loc[autumn_items, 'predicted_delivery_charges'] = autumn_model.predict(au
outlier_data.loc[spring_items, 'predicted_delivery_charges'] = spring_model.predict(sp
```

4.2. Calculate the residuals

`Residuals` , the distances of the data points from this hyperplane, are used to quantify the outlier scores i.e. it is the difference between the observed y-value and the predicted y-value (from regression equation line). It is the vertical distance from the actual plotted point to the point if plotted on the regression line.

Here, summer_predicted, winter_predicted, autumn_predicted and spring_predicted are the lists for each of the seasons having the delivery charges that have been predicted using the linear regression model.

Now, summer_observed, winter_observed, autumn_observed and spring_observed are the lists for each of the seasons having the actual delivery charges which can be extracted from the seemer, winter, autumn and spring season dataframes respectively.

$Residual = predicted - actual$

Then the residuals for each of the seasons is calculated by finding the difference between the predicted value and the actual/observed value for each of the four seasons and thus these residuals are then stored in residual_summer, residual_winter, residual_autumn and residual_spring.

In [ ]:

```
summer_predicted = outlier_data.loc[summer_items, 'predicted_delivery_charges'].to_lis
summer_observed = summer_df['delivery_charges'].to_list()
#residuals for the orders placed in the summer season
residual_summer = np.subtract(summer_predicted, summer_observed)
```

In [ ]:

```
winter_predicted = outlier_data.loc[winter_items, 'predicted_delivery_charges'].to_lis
winter_observed = winter_df['delivery_charges'].to_list()
#residuals for the orders placed in the winter season
residual_winter =np.subtract(winter_predicted, winter_observed)
```

In [ ]:

```
autumn_predicted = outlier_data.loc[autumn_items, 'predicted_delivery_charges'].to_lis
autumn_observed = autumn_df['delivery_charges'].to_list()
#residuals for the orders placed in the autumn season
residual_autumn =np.subtract(autumn_predicted, autumn_observed)
```

In [ ]:

```
spring_predicted = outlier_data.loc[spring_items, 'predicted_delivery_charges'].to_lis
spring_observed = spring_df['delivery_charges'].to_list()
#residuals for the orders placed in the spring season
residual_spring =np.subtract(spring_predicted, spring_observed)
```

In [ ]:

```
#a new column to store the calculated residuals
outlier_data['residuals'] = 0

outlier_data.loc[summer_items, 'residuals'] = list(residual_summer)
outlier_data.loc[winter_items, 'residuals'] = list(residual_winter)
outlier_data.loc[autumn_items, 'residuals'] = list(residual_autumn)
outlier_data.loc[spring_items, 'residuals'] = list(residual_spring)
```

4.3. Use boxplot to detect outliers

In this task, boxplot (boxes and whiskers plot) from matplotlib is used to investigate possible data outliers. It is used to show the shape of the distribution of the data and its values, its central value, and its variability with respect to the 'season' column.

In [ ]:

```
#show a boxplot for all the season
outlier_data.boxplot(column = 'residuals', by = 'season')
```

4.4. Investigate outliers for "season"

In this section, the outliers for each of the four seasons i.e. summer, winter, autumn and spring are investigated individually in order to find the outliers for each of the season.

4.4.1. Outliers for `summer` season

Here, the residual for the summer season is plotted using a boxplot in order to look for the outliers for the summer season.

The outliers are the values that will be more than the upper limit or lesser than the lower limit i.e. the points that lie outside the whiskers of the boxplot. As it can be seen from the boxplot, outliers are the points that are greater than 40 and the ones that are less than -30.
In order to check whether the outliers that have been found from the boxplot are correct or not, a `boxplot_stats()` function is used which return the 'fliers' i.e. the outliers for that data.
The indices for the rows having outliers are then stored in a list (summer_drop).

In [ ]:

```
#show boxplot for summer season
sns.boxplot(y = residual_summer)
summer = outlier_data[outlier_data.season=='Summer']
#get the outliers for summer season
boxplot_stats(summer.residuals).pop(0)['fliers']
```

In [ ]:

```
#stores the indices with outliers
summer_drop = summer.loc[(summer.residuals > 40)].index
#stores the indices with outliers
summer_drop = summer_drop.append(summer.loc[(summer.residuals < -30)].index)
summer_drop = summer_drop.tolist()
```

4.4.2. Outliers for `winter` season

Here, the residual for the winter season is plotted using a boxplot in order to look for the outliers for the winter season.

The outliers are the values that will be more than the upper limit or lesser than the lower limit i.e. the points that lie outside the whiskers of the boxplot. As it can be seen from the boxplot, outliers are the points that are greater than 40 and the ones that are less than -10.
In order to check whether the outliers that have been found from the boxplot are correct or not, a `boxplot_stats()` function is used which return the 'fliers' i.e. the outliers for that data.
The indices for the rows having outliers are then stored in a list (winter_drop).

In [ ]:

```
#show boxplot for winter season
sns.boxplot(y = residual_winter)
winter = outlier_data[outlier_data.season=='Winter']
#get the outliers for winter season
boxplot_stats(winter.residuals).pop(0)['fliers']
```

In [ ]:

```
#stores the indices with outliers
winter_drop = winter.loc[(winter.residuals > 40)].index
#stores the indices with outliers
winter_drop = winter_drop.append(winter.loc[(winter.residuals < -10)].index)
winter_drop = winter_drop.tolist()
```

4.4.3. Outliers for `autumn` season

Here, the residual for the autumn season is plotted using a boxplot in order to look for the outliers for the autumn season.

The outliers are the values that will be more than the upper limit or lesser than the lower limit i.e. the points that lie outside the whiskers of the boxplot. As it can be seen from the boxplot, outliers are the points that are greater than 42 and the ones that are less than -5.
In order to check whether the outliers that have been found from the boxplot are correct or not, a `boxplot_stats()` function is used which return the 'fliers' i.e. the outliers for that data.
The indices for the rows having outliers are then stored in a list (autumn_drop).

In [ ]:

```
#show boxplot for autumn season
sns.boxplot(y = residual_autumn)
autumn = outlier_data[outlier_data.season=='Autumn']
#get outliers for autumn season
boxplot_stats(autumn.residuals).pop(0)['fliers']
```

In [ ]:

```
#stores the indices with outliers
autumn_drop = autumn.loc[(autumn.residuals > 42)].index
#stores the indices with outliers
autumn_drop = autumn_drop.append(autumn.loc[(autumn.residuals < -5)].index)
autumn_drop = autumn_drop.tolist()
```

4.4.4. Outliers for `spring` season

Here, the residual for the spring season is plotted using a boxplot in order to look for the outliers for the spring season.

The outliers are the values that will be more than the upper limit or lesser than the lower limit i.e. the points that lie outside the whiskers of the boxplot. As it can be seen from the boxplot, outliers are the points that are greater than 2.8 and the ones that are less than -4.
In order to check whether the outliers that have been found from the boxplot are correct or not, a `boxplot_stats()` function is used which return the 'fliers' i.e. the outliers for that data.
The indices for the rows having outliers are then stored in a list (spring_drop).

In [ ]:

```
#show the boxplot for spring season
sns.boxplot(y = residual_spring)
spring = outlier_data[outlier_data.season=='Spring']
#get the outliers for spring season
boxplot_stats(spring.residuals).pop(0)['fliers']
```

In [ ]:

```
#stores the indices with outliers
spring_drop = spring.loc[(spring.residuals > 2.8)].index
#stores the indices with outliers
spring_drop = spring_drop.append(spring.loc[(spring.residuals < -4)].index)
spring_drop = spring_drop.tolist()
```

5. Remove the outlier rows

In this task, the outlier rows will be removed from the original data.

Now, summer_drop, winter_drop, autumn_drop and spring_drop, the lists having the indices for the outlier rows for each of the season are combined together and a new list (drop_list) is made which has the indices of all the outlier rows that have to be dropped. Finally, the outlier rows are removed from the original dataframe.

The column that were made for certain calculations are also removed in order to have the exact same columns as the input file.

In [ ]:

```
drop_list = [] #initialize an empty list
drop_list = list(chain(summer_drop, winter_drop, autumn_drop, spring_drop))
len(drop_list)
```

In [ ]:

```
#drop the outlier rows
outlier_data = outlier_data.drop(drop_list)
```

In [ ]:

```
#remove the extra columns
cols = ['predicted_delivery_charges', 'residuals']
outlier_data = outlier_data.drop(cols, 1)
```

In [ ]:

```
outlier_data.info()
```

6. Write to a csv file

The dataframe from which the outlier rows have been removed is then written to a csv file named "30757924_outlier_data_solution.csv" using the **to_csv()** function of the pandas library.

In [ ]:

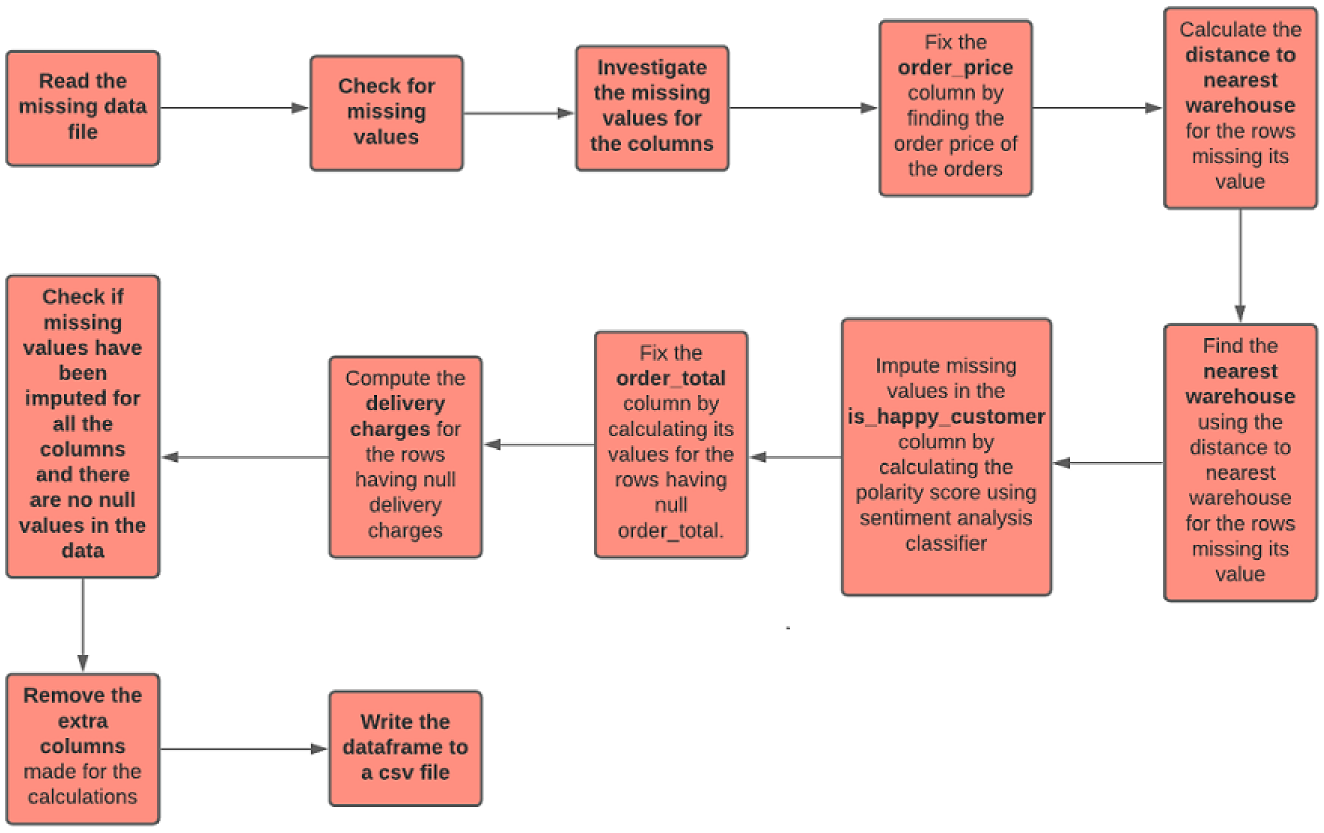
```
outlier_data.to_csv('30757924_outlier_data_solution.csv', index=False)
```

7. Summary of Part II

- In this task, the outlier rows have to be detected and removed from the data.
- According to the business rules, the delivery charges are calculated using a linear model which differs depending on the season. The model depends linearly (but in different ways for each season) on the distance between customer and nearest warehouse, whether the customer wants an expedited delivery, whether the customer was happy with his/her last purchase.
- So, the outliers have to be found with respect to the delivery_charges column only.

Part III: MISSING DATA

Sequence of Activities:


Read the missing data file
Check for missing values
Investigate the missing values for the columns
Fix the order_price column by finding the order price of the orders
Calculate the distance to nearest warehouse for the rows missing its value
Find the nearest warehouse using the distance to nearest warehouse for the rows missing its value
Impute missing values in the is_happy_customer column by calculating the polarity score using sentiment analysis classifier
Fix the order_total column by calculating its values for the rows having null order_total.
Compute the delivery charges for the rows having null delivery charges
Check if missing values have been imputed for all the columns and there are no null values in the data
Remove the extra columns made for the calculations
Write the dataframe to a csv file

Libraries used:

- pandas 1.0.3 (for data manipulation and analysis, included in Python 3.7.3)
- numpy 1.16.2 (for working with arrays, included in Python 3.7.3)
- re 2.2.1(for regular expression, included in Anaconda Python 3.7.3)
- matplotlib 3.0.3 (is a plotting library for the Python programming language and its numerical mathematics extension numpy)
- nltk 3.5 (the Natural Language Toolkit, is a suite of libraries and programs for symbolic and statistical natural language processing for English written in python)
- operator (it exports a set of efficient functions corresponding to the intrinsic operators of Python)
- functools (it is used for higher-order functions that act on or return other functions)

1. Introduction

Missing data occur when no data value is stored for a particular variable in a row. Missing data is common and is of high significance on the inferences that are drawn from the data.

In this task, the coverage anomalies i.e. the missing value problem is to be rectified, the entire data is checked for missing values and the columns having the missing values are investigated individually.

For the rows having missing value in a particular column, the missing values are then imputed.

2. Load the libraries

The required libraries are imported.

In [ ]:

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import re
import operator
import functools
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

3. Read the data

The first thing is to inpect the file and figure out the file format. To read the csv file the **read_csv()** function is used.

In [ ]:

```python
#load the data into a datframe
missing_data = pd.read_csv('30757924_missing_data.csv')
```

4. Explore the data

Now, as the data is loaded in a pandas dataframe, the overview of the data is taken. The columns are checked in order to see the data type of each of the attribute and the number of missing values in each of the columns.

In [ ]:

```python
#check the non-null entries and the data types of each column
missing_data.info()
```

In [ ]:

```python
#check for the number of null values in each column
missing_data.isnull().sum()
```

5. Fix the columns with missing values

In this section, the columns havings missing values will be fixed i.e. a value will be imputed at the place where the column is null.

5.1. Fix the order_price column

In this section, the `order_price` column is fixed by imputing the missing values.

Firstly, the products for each of the rows' shopping cart is determined by applying a regular expression on the shopping cart values. Secondly, another regular expression is applied on the shopping cart in order to get the quantitites of each of the product for that order. Then, a list of all the 10 branded items that the retail store sells them at competitive prices is made. Further, a list having the quantities of the item mapped to the products is created.

eg: products = ['Alcon 10', 'Candle Inferno', 'Lucent 330S', 'Olivia x460', 'Thunder line', 'Toshika 750', 'Universe Note', 'iAssist Line', 'iStream', 'pearTV']

product_quantity = [0, 1, 0, 0, 1, 2, 0, 0, 1, 0] So this list tells the quantity of the product according to the indices.

$$OrderPrice = \sum quantity * price$$

Finally, the order_price is calculated by mapping the price of each of the items(as calculated in Part 1) to their quantities, finding the product of quantity and price for each item and calculating the sum of those values for each of the orders.

In [ ]:

```python
#get the item names from the shopping cart
regex_for_items = r'(?:\(\')(.*?)\''
prods = []
for i in range(len(missing_data['shopping_cart'])):
    prods.append(re.findall(regex_for_items, missing_data['shopping_cart'][i]))

missing_data['products'] = prods
```

In [ ]:

```python
#get the product quantities from the shopping cart
regex_for_quantity = r', ([0-9]*?)\)'
quantity = []
for i in range(len(missing_data['shopping_cart'])):
    quantity.append(re.findall(regex_for_quantity, missing_data['shopping_cart'][i]))

missing_data['quantity'] = quantity
```

In [ ]:

```python
#get the list of products
prod_list = missing_data['products'].to_list()
prod = [prod_list[i][j] for i in range(len(prod_list)) for j in range(len(prod_list[i]
unique_prods = set(prod)
prod_final = sorted(list(unique_prods))

print(prod_final)
```

In [ ]:

```python
#function to find the quantity of the product
def product_quantity(value):
    prods=value['products']
    quants=value['quantity']
    list1=[]
    for p in prod_final:
        if p in prods:
            list1.append(int(quants[prods.index(p)]))
        else:
            list1.append(0)
    return list1
```

In [ ]:

```
# using this final price list that has been calculated in part 1
final_price=[8950.0,430.0,1230.0,1225.0,2180.0,4320.0,3450.0,2225.0,150.00000000000006
```

In [ ]:

```
#a new column to store the quantities of the products
missing_data['product_quantity'] = missing_data.apply(product_quantity,axis=1)
```

The reduce(fun,seq) function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along.

In [ ]:

```
#function to calculate the order price
def sum_of_products(value):
    list1=value['product_quantity']
    final_price_list = final_price
    if pd.isna(value['order_price']):
        return sum(functools.reduce(operator.mul, data) for data in zip(list1, final_pr
    else:
        return value['order_price']
```

In [ ]:

```
missing_data['order_price'] = missing_data.apply(sum_of_products, axis=1)
```

In [ ]:

```
#verify if the null values don't exist in the column after the imputations
missing_data.isnull().sum()
```

In [ ]:

```
#load the warehouses.csv file
warehouses = pd.read_csv('warehouses.csv')
```

In [ ]:

```
missing_data.nearest_warehouse.value_counts()
```

In [ ]:

```
warehouses
```

5.2. Fix the distance_to_nearest_warehouse column

In this section, the `distance_to_the_nearest_warehouse` attribute is fixed by imputing the missing values in this column.
Firstly, we calculate the haversine distance using the haversine function, this function calculates the distance between two points for which the latitude and the longitude is known. The radius of the earth is taken as 6378 km.

New columns,

`dist_from_N` which is the distance from Nickolson warehouse,
`dist_from_T` which is the distance from Thompson warehouse and
`dist_from_B` which is the distance from Bakers warehouse, are created.

These columns are made in order to find the distance between each customer and the warehouse using the latitudes and longitudes of both the customer and the warehouse.

Then the minimum of the three distances is calculated in order to find the distance to the nearest warehouse. Lastly, the missing values in the **distance_to_the_nearest_warehouse** column are imputed in the calculated values.

In [ ]:

```
#the haversine distance is calculated to find the distance between two geo-locations
def haversine_distance(lat1, lon1, lat2, lon2):
    r = 6378
    phi1 = np.radians(lat1)
    phi2 = np.radians(lat2)
    delta_phi = np.radians(lat2 - lat1)
    delta_lambda = np.radians(lon2 - lon1)
    a = np.sin(delta_phi / 2)**2 + np.cos(phi1) * np.cos(phi2) *   np.sin(delta_lambda
    res = r * (2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a)))

    return np.round(res, 4)
```

In [ ]:

```
dist_from_N=[]#list to store customer's distance from Nickolson
dist_from_T=[]#list to store customer's distance from Thompson
dist_from_B=[]#list to store customer's distance from Bakers
for i in missing_data.index:
    dist_from_N.append(haversine_distance(warehouses['lat'][0], warehouses['lon'][0],
    dist_from_T.append(haversine_distance(warehouses['lat'][1], warehouses['lon'][1],
    dist_from_B.append(haversine_distance(warehouses['lat'][2], warehouses['lon'][2],
```

In [ ]:

```
#column to store customer's distance from Nickolson
missing_data['dist_from_N']=dist_from_N
#column to store customer's distance from Thompson
missing_data['dist_from_T']=dist_from_T
#column to store customer's distance from Bakers
missing_data['dist_from_B']=dist_from_B
```

In [ ]:

```
#find the minimum of the three distances
new = missing_data[['dist_from_N', 'dist_from_T', 'dist_from_B']].min(axis=1)
#store the minimum distance in a new column

missing_data['minimum_dist'] = new
```

In [ ]:

```
#for the missing values, the correct values are being imputed using the above calculat
for i in range(len(missing_data['distance_to_nearest_warehouse'])):
    if pd.isna(missing_data['distance_to_nearest_warehouse'][i]):
        missing_data['distance_to_nearest_warehouse'][i] = missing_data['minimum_dist'
    else:
        missing_data['distance_to_nearest_warehouse'][i]=missing_data['distance_to_nea
```

5.3. Fix the nearest_warehouse column

In this section, the nearest warehouse attribute is fixed by imputing the missing values in this column.
Using the minimum distance that was calculated in the previous section, the nearest warehouse can be found by checking for the minimum distance.

If the minimum distance was 'dist_from_N', then the nearest warehouse is Nickolson.
If the minimum distance was 'dist_from_T', then the nearest warehouse is Thompson.
If the minimum distance was 'dist_from_B', then the nearest warehouse is Bakers.

The null empty values are then imputed in the **nearest_warehouse** column.

In [ ]:

```
#list to store the nearest warehouse names
nearest_retail=[]
for i in missing_data.index:
    if(missing_data['minimum_dist'][i]==missing_data['dist_from_N'][i]):
        nearest_retail.append('Nickolson')
    if(missing_data['minimum_dist'][i]==missing_data['dist_from_T'][i]):
        nearest_retail.append('Thompson')
    if(missing_data['minimum_dist'][i]==missing_data['dist_from_B'][i]):
        nearest_retail.append('Bakers')

#column to store nearest warehouse names
missing_data['nearest_retail'] = nearest_retail
```

In [ ]:

```
#for the missing values, the correct values are being imputed using the above calculat
for i in range(len(missing_data['nearest_warehouse'])):
    if pd.isna(missing_data['nearest_warehouse'][i]):
        missing_data['nearest_warehouse'][i] = missing_data['nearest_retail'][i]
    else:
        missing_data['nearest_warehouse'][i]=missing_data['nearest_warehouse'][i]
```

In [ ]:

```
missing_data[missing_data.nearest_warehouse != missing_data.nearest_retail]
```

In [ ]:

```
# verify if no null values exist in the nearest_warehouse column
missing_data.info()
```

5.4. Impute the missing values in is_happy_customer column

In this section, the `is_happy_customer` attribute is fixed by imputing the missing values in this column. To check whether a customer is happy with their last order, the customer's latest review is classified using a sentiment analysis classifier. `SentimentIntensityAnalyzer` from `nltk.sentiment.vader` is used to obtain the polarity score. A sentiment is considered positive if it has a 'compound' polarity score of 0.05 or higher and is considered negative otherwise.

The missing values are then imputed in the **is_happy_customer** column according to the above calculated polarity scores.

In [ ]:

```
#define the object
sentiment_analyser = SentimentIntensityAnalyzer()
sents=[]#initialize an empty list
for i in range(len(missing_data['latest_customer_review'])):
    if (sentiment_analyser.polarity_scores(missing_data['latest_customer_review'][i])[
        sents.append(True)#if the condition is true i.e. the review is positive so the
    else:
        sents.append(False)
#make a new column and put the values of the sents list in the column
missing_data['sents'] = sents
```

In [ ]:

```
#for the missing values, the correct values are being imputed using the above calculat
for i in range(len(missing_data['is_happy_customer'])):
    if pd.isna(missing_data['is_happy_customer'][i]):
        missing_data['is_happy_customer'][i] = missing_data['sents'][i]
    else:
        missing_data['is_happy_customer'][i]=missing_data['sents'][i]
```

In [ ]:

```
missing_data[missing_data.is_happy_customer != missing_data.sents]
```

In [ ]:

```
# verify if no null values exist in the is_happy_customer column
missing_data.info()
```

In [ ]:

```
#check that the delivery charges and order total are not null together in any row
missing_data[(pd.isna(missing_data['delivery_charges'])) & (pd.isna(missing_data['orde
```

In [ ]:

```
missing_data.isnull().sum()
```

5.5. Impute missing values in order_total column

In this section, the order_total attribute is fixed by imputing the missing values in this column.

$$OrderTotal = OrderPrice - (Discount * OrderPrice)/100 + DeliveryCharges$$

In this way the order total is calculated for the rows having null values in the order total column. The order total is rounded off to two decimal places in order to maintain the consistency of the data.

The missing values are then imputed in the **order_total** column according to the calculations performed.

In [ ]:

```
#get the rows with missing order_total
missing_total = missing_data[pd.isna(missing_data['order_total'])]
```

In [ ]:

```
orders=[] #initialize an empty list to store the order_id of the orders with missing or
for i in range(len(missing_total['order_id'])):
    orders.append(missing_total['order_id'].iloc[i])
```

In [ ]:

```
#impute the missing values with the order total that can be calculated using the calcu
for i in range(len(missing_data['order_total'])):

     if missing_data['order_id'][i] in orders:
        order_price = missing_data['order_price'][i]
        delivery_charge = missing_data['delivery_charges'][i]
        discount = missing_data['coupon_discount'][i]
        result = (order_price - ((discount*order_price)/100)) + delivery_charge

        missing_data['order_total'].fillna(round(result,2), inplace=True)
```

In [ ]:

```
# verify if no null values exist in the order_total column
missing_data.isnull().sum()
```

In [ ]:

```
missing_data.info()
```

5.6. Fix the delivery_charges column by imputing the missing values

In this section, the delivery_charges attribute is fixed by imputing the missing values in this column.

$$DeliveryCharges = OrderTotal + (Discount * OrderPrice)/100 - OrderPrice$$

In this way, delivery charges have been calculated and the missing values in the **'delivery_charges'** have been imputed based on this calculation.

In [ ]:

```
#calculate the delivery charges based on the formula shown above
def delivery_charges(value):

    order_price=value['order_price']
    discount=value['coupon_discount']
    order_total=value['order_total']
    delivery_charges = order_total+(order_price*(discount/100))-order_price

    return delivery_charges
```

In [ ]:

```
missing_data['new_delivery_charges'] = missing_data.apply(delivery_charges, axis=1)
```

In [ ]:

```
#impute the missing values in the delivery_charges column with the ones that have been
for i in range(len(missing_data['delivery_charges'])):
    if pd.isna(missing_data['delivery_charges'][i]):
        missing_data['delivery_charges'][i] = missing_data['new_delivery_charges'][i]
    else:
        missing_data['delivery_charges'][i]=missing_data['new_delivery_charges'][i]
```

In [ ]:

```
missing_data[missing_data.delivery_charges != missing_data.new_delivery_charges]
```

In [ ]:

```
# verify if no null values exist in the delivery_charges column
missing_data.isnull().sum()
```

6. Remove the extra columns

Some extra columns have been made in this task of imputing the missing values. These columns are to be removed in order to make the data consistent and in the same format as the original missing_data file.

In [ ]:

```
#columns to be removed
cols = ['products','quantity','product_quantity','dist_from_B','dist_from_N','dist_fro
missing_data = missing_data.drop(cols, 1)
```

7. Verification

Here, the final data is verified in order to check whether all the missing values of the attributes in the missing_data file have been imputed or not.

In [ ]:

```
missing_data.info()
```

In [ ]:

```
missing_data.isnull().sum()
```

8. Write to a csv file

The dataframe from which the outlier rows have been removed is then written to a csv file named "30757924_missing_data_solution.csv" using the **to_csv()** function of the pandas module.

In [ ]:

```
missing_data.to_csv('30757924_missing_data_solution.csv', index=False)
```

9. Summary of Part III

- In this part, the missing values, which cause a lot of problems in the data, are imputed by suitable values.
- For imputing the missing values, the columns had to be investigated individually in order to find the rows having missing values and then calculations have been made to replace the null value with a valid value.

CONCLUSION

- The data is explored graphically as well as non-graphically as it is one of the most important part of the data wrangling process.
- In **Part I:** The errors in the `30757924_dirty_data.csv` are detected and the errors are fixed and the error-fress data is stored in `30757924_dirty_data_solution.csv` file.
- In **Part II:** The outlier rows are detected from `30757924_outlier_data.csv` with respect to the delivery_charges and after deleting the outlier rows, the data is then stored in `30757924_outlier_data_solution.csv` file.
- In **Part III:** The missing values in the `30757924_missing_data.csv` are imputed with the correct ones and this data is then stored in `30757924_missing_data_solution.csv` file.

REFERENCES

1. Data cleansing. (2020). Retrieved 18 October 2020, from https://en.wikipedia.org/wiki/Data_cleansing (https://en.wikipedia.org/wiki/Data_cleansing)
2. Data Cleansing: What Is It and Why Is it Important?. (2020). Retrieved 18 October 2020, from https://www.blue-pencil.ca/data-cleansing-what-is-it-and-why-is-it-important/ (https://www.blue-pencil.ca/data-cleansing-what-is-it-and-why-is-it-important/)
3. Data Cleaning With Pandas. (2020). Retrieved 18 October 2020, from https://dribbble.com/shots/4215452-Data-Cleaning-With-Pandas (https://dribbble.com/shots/4215452-Data-Cleaning-With-Pandas)

4. Here’s How To Calculate Distance Between 2 Geolocations in Python. (2020). Retrieved 18 October 2020, from https://towardsdatascience.com/heres-how-to-calculate-distance-between-2-geolocations-in-python-93ecab5bbba4 (https://towardsdatascience.com/heres-how-to-calculate-distance-between-2-geolocations-in-python-93ecab5bbba4)

5. Haversine formula. (2020). Retrieved 18 October 2020, from https://en.wikipedia.org/wiki/Haversine_formula (https://en.wikipedia.org/wiki/Haversine_formula)

6. reduce() in Python - GeeksforGeeks. (2020). Retrieved 18 October 2020, from https://www.geeksforgeeks.org/reduce-in-python/ (https://www.geeksforgeeks.org/reduce-in-python/)

7. Understanding this line: list_of_tuples = [(x,y) for x, y, label in data_one]. (2020). Retrieved 18 October 2020, from https://stackoverflow.com/questions/38229059/understanding-this-line-list-of-tuples-x-y-for-x-y-label-in-data-one (https://stackoverflow.com/questions/38229059/understanding-this-line-list-of-tuples-x-y-for-x-y-label-in-data-one)

8. pandas.DataFrame.loc — pandas 1.1.3 documentation. (2020). Retrieved 18 October 2020, from https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html)

9. sklearn.linear_model.LinearRegression — scikit-learn 0.23.2 documentation. (2020). Retrieved 18 October 2020, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

10. sklearn.linear_model.LinearRegression — scikit-learn 0.23.2 documentation. (2020). Retrieved 18 October 2020, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

11. Enderlein, G. (1987). Hawkins, DM: Identification of Outliers. Chapman and Hall, London–New York 1980, 188 S.,£ 14, 50. Biometrical Journal, 29(2), 198-198.

12. Faculty of IT, Data Cleansing-3. Retrieved from URL https://d3cgwrxphz0fqu.cloudfront.net/ed/3d/ed3d8a0d2ec21337e9f1f94c6391b92efdc15b1c?response-content-disposition=inline%3Bfilename%3D%22week_8.pdf%22&response-content-type=application%2Fpdf&Expires=1602842505&Signature=NHXaBsbDXzZgL-X0IMm2yZJbyuqYKDmlvKp6DNqyims1L1Sp-Uo0NHFLvmX8LANr8UschRx3IwJUGcpu5vbl00pGbfc2sToozpECAjTJiaD3aSxAD7CqezeJR8jLCtQ10N9hg
IePcRG6hrfZHHJuoZP0NB5Kds8YQJj94z1g3u132X-DcQIhuiHt0jIaX1mD7laZLqGzGIRCf23kFg__&Key-Pair-Id=APKAJRIEZFHR4FGFTJHA (https://d3cgwrxphz0fqu.cloudfront.net/ed/3d/ed3d8a0d2ec21337e9f1f94c6391b92efdc15b1c?response-content-disposition=inline%3Bfilename%3D%22week_8.pdf%22&response-content-type=application%2Fpdf&Expires=1602842505&Signature=NHXaBsbDXzZgL-X0IMm2yZJbyuqYKDmlvKp6DNqyims1L1Sp-Uo0NHFLvmX8LANr8UschRx3IwJUGcpu5vbl00pGbfc2sToozpECAjTJiaD3aSxAD7CqezeJR8jLCtQ10N9hg
IePcRG6hrfZHHJuoZP0NB5Kds8YQJj94z1g3u132X-DcQIhuiHt0jIaX1mD7laZLqGzGIRCf23kFg__&Key-Pair-Id=APKAJRIEZFHR4FGFTJHA)

13. Code Faster with Line-of-Code Completions, Cloudless Processing. (2020). Retrieved 18 October 2020, from https://www.kite.com/python/docs/matplotlib.cbook.boxplot_stats (https://www.kite.com/python/docs/matplotlib.cbook.boxplot_stats)

14. pandas.DataFrame.apply — pandas 1.1.3 documentation. (2020). Retrieved 18 October 2020, from https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.apply.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.apply.html)

15. Faculty of IT, Data Cleansing-1. Retrieved from URL: https://d3cgwrxphz0fqu.cloudfront.net/cc/7a/cc7aa313068a1bb3a01c58f0bc1b24d69bb227a3?response-content-disposition=inline%3Bfilename%3D%22week_6_slides.pdf%22&response-content-type=application%2Fpdf&Expires=1602904908&Signature=L4sIRO8Lx~RmCZ1PBiOxaBivMWXxGrn1CzQ~dw6UL~tABqwmD3grKJFickACp2HMrp35KWsF0iApQe33-KllCoYT-EUvp7zeu~eTMkbnz5Q4l8Hm5UnTdWgtQvc7KaYjarJVDgLUBL0nIzULwwGW47csJ7kkTnUfJUBSflSDssa2
o-C02msFA__&Key-Pair-Id=APKAJRIEZFHR4FGFTJHA (https://d3cgwrxphz0fqu.cloudfront.net/cc/7a/cc7aa313068a1bb3a01c58f0bc1b24d69bb227a3?response-content-disposition=inline%3Bfilename%3D%22week_6_slides.pdf%22&response-content-type=application%2Fpdf&Expires=1602904908&Signature=L4sIRO8Lx~RmCZ1PBiOxaBivMWXxGrn1CzQ~dw6UL~tABqwmD3grKJFickACp2HMrp35KWsF0iApQe33-KllCoYT-EUvp7zeu~eTMkbnz5Q4l8Hm5UnTdWgtQvc7KaYjarJVDgLUBL0nIzULwwGW47csJ7kkTnUfJUBSflSDssa2
o-C02msFA__&Key-Pair-Id=APKAJRIEZFHR4FGFTJHA)

16. Generate all permutation of a set in Python - GeeksforGeeks. (2020). Retrieved 18 October 2020, from https://www.geeksforgeeks.org/generate-all-the-permutation-of-a-list-in-python/ (https://www.geeksforgeeks.org/generate-all-the-permutation-of-a-list-in-python/)

17. operator — Standard operators as functions — Python 3.9.0 documentation. (2020). Retrieved 18 October 2020, from https://docs.python.org/3/library/operator.html (https://docs.python.org/3/library/operator.html)

18. functools — Higher-order functions and operations on callable objects — Python 3.9.0 documentation. (2020). Retrieved 18 October 2020, from https://docs.python.org/3/library/functools.html (https://docs.python.org/3/library/functools.html)