FIT5196 Task in Assessment 2

Student Name: Gayatri Aniruddha

Student ID: 30945305

Date: 18/10/2020

Version: 1.0

Environment: Python 2.7.11 and Jupyter notebook

Libraries used: please include the main libraries you used in your assignment here, e.g.,:

- **pandas** : For loading and storing the data in a Panda DataFrame.
- **nltk** : For sentiment analysis
- **numpy** : For mathematical and matrix operations
- **math** : For calculating the geographical distances using math formulae
- **datetime** : For changing the date formats
- **matplotlib.pyplot** : To plot the boxplots in order to view the outliers
- **LinearRegression** : To create linear models while predicting the delivery charges

Table of Contents

1. Introduction

- We have been given with three files. The files contain:
    - Dirty data
    - Outlier data
    - Missing data
- This dataset contains the orders placed for an online electronics store in Melbourne.
- The dataset has around 500 rows and 16 columns.
- Every row in the dataset represents a single order that was made from the store.
- The different details that are made available to us are as follows:
    - order_id and customer_id
    - customer locations
    - date of the order
    - season of the order
    - warehouse located nearby
    - warehouse geographical locations
    - distance to the closest warehouse
    - items purchased, discounts offered, delivery charges
    - customer reviews and customer happiness index
- I have subdivided the question requirements into 3 sub-tasks as follows:
    - **Task 1** : Cleaning the dirty data
    - **Task 2** : Impute missing values in the missing data
    - **Task 3** : Remove outlier rows in the outlier data

2. Methodology

- This includes all the data preparation steps.

2.1. Task 1 : Cleaning the Dirty Data

- In general, there are a lot of iterations which are performed while doing data-cleaning. We first identify the anomalies while performing data cleaning. These anomalies decrease the quality of the data. Here, we apply many statistical tests.
- There are 3 main types of data anomalies that I have discovered while cleaning my dataset:
    - Syntatic
    - Semantic
    - Coverage
- Here, this task revolves around detecting and fixing the errors in the dirty dataset.
- I have followed the following steps in order to clean the data.

STEPS FOLLOWED:

- I have first loaded the dirty data into a dataframe.
- I have then examined the structure and shape of the dataframe.
- Syntactic Anomalies: I have then identified certain inconsistencies in the spellings of the warehouse names and season names and corrected the same.
- After the basic exploration of the dataset and correction of lexical anomalies, I have explored each column of the dirty data.
- As per the document specification, the following columns have no errors:
  - delivery_charges
  - coupon_discount
- After examination, I found that the following columns did not have any errors:
  - order_id
  - customer_id
  - shopping_cart
  - is_expedited_delivery
  - latest_customer_review
- I have identified the following mistakes and made the required corrections in every column as follows:
- **Date:**
  - Syntactic Anomaly
  - Total corrected rows: 39
  - All the dates have been converted to "YYYY-MM-DD" format as per the assignment specification.
  - After checking, I confirmed that there are no wrong values of days and months entered.
- **Season:**
  - Semantic Anomaly
  - Total corrected rows: 22
  - The month corresponding to every season has been provided in the assignment specification.
  - Certain months had incorrectly marked seasons and I have made the necessary corrections.
- **Shopping Cart:**
  - Semantic Anomaly
  - Here, I eaxmined and created a set of all the unique items ordered.
  - This came out to be a set of ten unique items.
  - Hence, there were no errors in this column and it was verfied that the retail store focuses on only 10 branded items.
- **Order Price:**
  - Semantic Anomaly
  - Total corrected rows: 54
  - Here, I used the linear algebra method in order to compute the prices for the individual items.
  - This was then used to calculate the order price for every order.
  - Certain order prices that were wrongly calculated were corrected.
- **Order Total:**
  - Semantic Anomaly
  - Total corrected rows: 54
  - This is the total price of the order, after adding the discounts and taking into consideration the delivery charges.

    - Certain incorrectly calculated order totals were corrected with the correctly calculated amounts.
- **Customer Locations:**
    - Syntactic Anomaly
    - Total corrected rows: 27
    - On examining certain customer locations, I discovered that some of the customer latitudes and longitudes were exchanged. These values were corrected.
- **Nearest Warehouse:**
    - Semantic Anomaly
    - Total corrected rows: 22
    - On calculating the distance to the nearest warehouse, I discovered that certain nearest warehouses were wrongly named.
    - These warehouses were correctly named.
- **Distance_to_nearest_warehouse:**
    - Semantic Anomaly
    - Total corrected rows: 43
    - Here, I observed that the nearest warehouses were correctly named but the distance was incorrectly calculated.
    - These incorrectly calculated distances to the nearest warehouses were calculated.
- **Customer Reviews:**
    - Semantic Anomaly
    - Total corrected rows: 28
    - Here, we use the sentiment analyzer to analyze the customer reviews in order to get the sentiment polarity.
    - On examining, I noticed that certain customer happiness indexes were wrongly marked when compared to the polarity values.

Importing Libraries

- This section contains the code to import the libraries we need in this assessment.

In [ ]:
```
# Importing the required libraries

# For storing and working with the files in a dataframe
import pandas as pd

# For Sentiment Analysis
import nltk
# For calculating the sentiment polarity values
from nltk.sentiment.vader import SentimentIntensityAnalyzer

# For mathemcatical and matrix operations
import numpy as np

# For calculating the geographical distances using math formulae
import math

# For changing the date formats
import datetime as dt

# For plotting boxplots to detect outliers
import matplotlib.pyplot as plt
%matplotlib inline

# For computing the delivery charges using a linear regression model
from sklearn.linear_model import LinearRegression

# For matrix evaluation
from scipy import linalg
```

Loading the data using Pandas library

- Here, the first thing we do is inspect the file and see the format of the file.
- Here, the dirty data is stored in a csv file.
- Hence, we use the read_csv() function

In [ ]:
```
# Loading the dirty data dataset
my_dirty_data = pd.read_csv("30945305_dirty_data.csv")
```

In [ ]:
```
# Loading the missing data dataset
my_missing_data = pd.read_csv("30945305_missing_data.csv")
```

In [ ]:
```
# Loading the outlier data dataset
my_outlier_data = pd.read_csv("30945305_outlier_data.csv")
```

In [ ]:
```
# Getting the number of rows and columns of the dataset
print (my_dirty_data.shape)
```

In [ ]:
```
# Displaying the top 5 elements of the dataset
my_dirty_data.head(5)
```

Categorical Variables:

- season
- is_expected_delivery
- is_happy_customer

In [ ]:
```
# Displaying the bottom 5 elements of the dataset
my_dirty_data.tail(5)
```

info()

- This gives us the total number of records present in the dataset.
- It also gives us the data-type of each column.
- Types of Data :
    - 'object' here means string
    - int and float
- It also tells us the number of Non-Null observations present in each column.

In [ ]:
```
my_dirty_data.info()
```

Numerical Data:

- order_price
- delivery_charges
- customer_lat
- customer_long
- coupon_discount
- order_total

Non-Numeric Data:

- order_id
- customer_id
- date
- nearest_warehouse
- shopping_cart

Boolean Data:

- season
- is_expected_delivery

In [ ]:
```
my_dirty_data.describe()
```

describe()

- By default, this gives us information about the numeric data.
- This gives us the descriptive statistics of the dataset.

- Here, count represents the total number of Non-Null observations.

```
In [ ]: # This gives us the decriptive statistics of all the non-numeric data
        my_dirty_data.describe(include = ['O'])
```

Observations: Possible Mistakes

Point 1:

- We can see that there are 6 unique values of the warehouses.
- But, in the question it is given that there are 3 unique warehouses.
- Thus, there must something wrong in the warehouse names.

Point 2:

- Similarly, we can see that there are 8 unique values of the seasons.
- But, we know that there are 4 seasons.
- Thus, there must something wrong in the warehouse names.

Non-Numeric Data:

- order_id
- customer_id
- date
- nearest_warehouse
- shopping_cart

Identify Syntactic Anomalies:

- Here, I have identified certain lexical errors of the dataset.

```
In [ ]: # Checking for inconsistent spellings in nearest_warehouse column
        my_dirty_data.nearest_warehouse.value_counts()
```

Inconsist spellings:

- Thompson and thompson
- Bakers and bakers
- Nickolson and nickolson

Assumption:

- Here, I have made the assumption that the spelling with more number of counts has a lesser chance to be wrong.
- Hence, I am going ahead with the spelling with the larger count.
  - **Thompson**
  - **Bakers**
  - **Nickolson**

- The spellings with the larger count will be set as the standard.

In [ ]:
```
# Correcting the spellings for nearest_warehouse
my_dirty_data.nearest_warehouse.replace({"thompson": "Thompson", "bakers": "Bake
```

In [ ]:
```
# Checking if the replacements have been made
my_dirty_data.nearest_warehouse.value_counts()
```

Clearly, the **nearest_warehouse** spellings have been corrected!

In [ ]:
```
my_dirty_data.season.value_counts()
```

Inconsist spellings:

- Summer and summer
- Autumn and autumn
- Winter and winter
- Spring and spring

Assumption:

- Here, I have made the assumption that the spelling with more number of counts has a lesser chance to be wrong.
- Hence, I am going ahead with the spelling with the larger count.
  - **Summer**
  - **Autumn**
  - **Winter**
  - **Spring**
- The spellings with the larger count will be set as the standard.

In [ ]:
```
# Correcting the spellings for season
my_dirty_data.season.replace({"summer": "Summer", "autumn": "Autumn", "winter":"
```

In [ ]:
```
# Checking if the replacements have been made or not
my_dirty_data.season.value_counts()
```

Clearly, the **season** spellings have been corrected!

Incorrect Spelling:

- Here, there are no inconsistencies in the spellings of True and False.

In [ ]:
```
#Checking for duplicate records
my_dirty_data[my_dirty_data.duplicated(["order_id"], keep=False)].head(2)
```

Clearly, we can see that there is no duplicate data!

In [ ]:
```
# Checking for consistency in data
my_dirty_data['customer_id'].groupby(my_dirty_data['nearest_warehouse']).describ
```

2.1.1 Exploratory Data Analysis : Date Values

Examining the Date Values:

- According to the assignment specification, all the dates should be in this format: "YYYY-MM-DD"
- Here, we examine the format of the dates and convert it into this required format!
- A total of 39 rows were corrected.
- I even checked if there were any incorrectly entered days values corresponding to the months.
- In order to perform the correctness of the dates, the new columns of days, months and years were finally removed.

In [ ]:
```
# https://stackoverflow.com/questions/51822956/change-dd-mm-yyyy-date-format-of-
# https://stackoverflow.com/questions/49435438/pandas-validate-date-format
# Changing the date-format to yyyy-mm-dd
my_dirty_data['new_date'] = pd.to_datetime(my_dirty_data['date'], format='%Y-%m-
my_dirty_data.new_date.isnull().sum()
# Thus there are 27 Null values
```

In [ ]:
```
# Handing incorrect dates
wrong_dates = my_dirty_data.new_date.isnull()
my_dirty_data.loc[wrong_dates,'new_date'] = pd.to_datetime(my_dirty_data[wrong_d
my_dirty_data.new_date.isnull().sum()
# Thus, the Null Values have been decreased to 12 now
```

In [ ]:
```
# Now, there are dates in the DD-MM-YYYY Format
my_dirty_data["date"].tail(15)
```

In [ ]:
```
wrong_dates = my_dirty_data.new_date.isnull()
my_dirty_data.loc[wrong_dates,'new_date'] = pd.to_datetime(my_dirty_data[wrong_d
```

In [ ]:
```
my_dirty_data.new_date.isnull().sum()
# Thus, all Null values have been deleted
```

In [ ]:
```
my_dirty_data["new_date"].tail(15)
```

In [ ]:
```
# Deleting the old date column
del my_dirty_data['date']
```

In [ ]:
```
# Renaming new_date column as the date column
my_dirty_data.rename(columns={'new_date': 'date'}, inplace = True)
```

Inference and Analysis:

- Now that all the formats of the dates have been corrected, I will now check for any incorrectly entered dates.

In [ ]:
```
# https://stackoverflow.com/questions/26105804/extract-month-from-date-in-python,
my_dirty_data['month'] = my_dirty_data['date'].dt.month
my_dirty_data['year'] = my_dirty_data['date'].dt.year
my_dirty_data['day'] = my_dirty_data['date'].dt.day
my_dirty_data[["date","year","month","day"]].head()
```

In [ ]:
```
# Checking for incorrect day values
days = dict(my_dirty_data.day.value_counts())
count = 0
for x in days.keys():
    if x > 31:
        count = count + 1
print(count)
# Hence, no incorrect days
```

In [ ]:
```
# Checking for incorrect month values
months = dict(my_dirty_data.month.value_counts())
count = 0
for x in months.keys():
    if x > 12:
        count = count + 1
print(count)
# Hence, no incorrect months
```

2.1.2 Exploratory Data Analysis : Season Analysis

Examining the seasons:

- As per the season documentation followed in Australia,
  - Summer Months : December, January, February
  - Autumn Months : March, April, May
  - Winter Months : June, July, August
  - Spring Months : September, October, November
- Thus, the seasons according to the month numbers are:
  - Summer Months : 12, 1, 2
  - Autumn Months : 3, 4, 5
  - Winter Months : 6, 7, 8
  - Spring Months : 9, 10, 11
- Here, while exploring the different seasons, I discovered that certain months have wronly marked seasons.
- A total of 22 rows had incorrectly marked seasons.
- These rows were then corrected.

In [ ]:
```
# http://www.bom.gov.au/climate/glossary/seasons.shtml
my_dirty_data[["season"]].describe()
```

In [ ]:
```
my_dirty_data.season.value_counts()
```

Summer Season : Correcting the incorrectly marked summer months

In [ ]:
```
summer = ( (my_dirty_data.month == 12) | (my_dirty_data.month == 1) | (my_dirty_

# Checking the wrong season : summer
wrong_summer = my_dirty_data[summer]
wrong_summer[["month","season"]]
```

In [ ]:
```
# Number of incorrectly marked Summer seasons
print(wrong_summer.shape)
```

In [ ]:
```
# Correcting the incorrectly marked Summer seasons
summer = ( (my_dirty_data.month == 12) | (my_dirty_data.month == 1) | (my_dirty_
my_dirty_data.loc[summer, 'season'] = "Summer"
```

In [ ]:
```
# Checking if the corrections have been made or not
wrong_summer = my_dirty_data[summer]
wrong_summer.shape
```

Autumn Season : Correcting the incorrectly marked autumn months

In [ ]:
```
autumn = ( (my_dirty_data.month == 3) | (my_dirty_data.month == 4) | (my_dirty_da

# Checking the wrong season : summer
wrong_autumn = my_dirty_data[autumn]
wrong_autumn[["month","season"]]
```

In [ ]:
```
# Number of incorrectly marked Summer seasons
print(wrong_autumn.shape)
```

In [ ]:
```
# Correcting the incorrectly marked Autumn seasons
autumn = ( (my_dirty_data.month == 3) | (my_dirty_data.month == 4) | (my_dirty_da
my_dirty_data.loc[autumn, 'season'] = "Autumn"
```

In [ ]:
```
# Checking if the corrections have been made or not
wrong_autumn = my_dirty_data[autumn]
wrong_autumn.shape
```

Winter Season : Correcting the incorrectly marked winter months

In [ ]:
```
winter = ( (my_dirty_data.month == 6) | (my_dirty_data.month == 7) | (my_dirty_da

# Checking the wrong season : summer
wrong_winter = my_dirty_data[winter]
wrong_winter[["month","season"]]
```

In [ ]:
```
# Number of incorrectly marked Winter seasons
print(wrong_winter.shape)
```

In [ ]:
```
# Correcting the incorrectly marked Winter seasons
winter = ( (my_dirty_data.month == 6) | (my_dirty_data.month == 7) | (my_dirty_da
my_dirty_data.loc[winter, 'season'] = "Winter"
```

In [ ]:
```
# Checking if the corrections have been made or not
wrong_winter = my_dirty_data[winter]
wrong_winter.shape
```

Spring Season : Correcting the incorrectly marked spring months

In [ ]:
```
spring = ( (my_dirty_data.month == 9) | (my_dirty_data.month == 10) | (my_dirty_

# Checking the wrong season : spring
wrong_spring = my_dirty_data[spring]
wrong_spring[["month","season"]]
```

In [ ]:
```
# Number of incorrectly marked Spring seasons
print(wrong_spring.shape)
```

In [ ]:
```
# Correcting the incorrectly marked Spring seasons
spring = ( (my_dirty_data.month == 9) | (my_dirty_data.month == 10) | (my_dirty_
my_dirty_data.loc[spring, 'season'] = "Spring"
```

In [ ]:
```
# Checking if the corrections have been made or not
wrong_spring = my_dirty_data[spring]
wrong_spring.shape
```

In [ ]:
```
my_dirty_data.info()
```

2.1.3 Exploratory Data Analysis : Shopping Cart

Examining the shopping cart

- Here, we first examine all the rows of the shopping cart column.
- Every row represents a shopping cart which has the following:
  - Item ordered
  - Quantity of the item ordered
- I have then used to comvert every row into a dictionary format, where :
  - Key : Item ordered
  - Value : Quantity of the item ordered

```
In [ ]: # Examining the top 2 elements of the shopping cart
        my_dirty_data["shopping_cart"].head(2)
```

```
In [ ]: # Comverting the values into a dictionary for easier evaulation
        my_dirty_data['shopping_cart'] = my_dirty_data['shopping_cart'].apply(lambda x: 
        # Examining the top 2 elements of the shopping cart
        my_dirty_data["shopping_cart"].head(2)
```

```
In [ ]: # List to store all the shopping items names
        items_name = []

        # Adding the items from every shopping cart into a list
        my_dirty_data['shopping_cart'].apply(lambda x:items_name.append(x.keys()))

        #https://stackoverflow.com/questions/952914/how-to-make-a-flat-list-out-of-list-
        import itertools
        final_list = list(itertools.chain(*items_name))

        # Creating an unique list
        unique_list = set(final_list)

        # Printing out all the items of the list
        count = 1
        for every_item in unique_list:
            print(count,":", every_item)
            count = count + 1
```

- In the assignment specification, it is mentioned that the store has it's sales focused on only 10 branded items.
- Thus, from this data exploration, we can clearly see that there are only **10 unique items** present in the shopping cart!

2.1.4 Exploratory Data Analysis : Order Price

Examining the following

- Now, as per the assignment specification, it is given that there are going to be only 10 unique items.
- We have been provided with the item quantities and the total order price.
- We now need to compute the individual item prices.

- After finding these items, I created a coefficient matrix that included the different item quantities.
- Then, I used the matrix multiplication method, in order to compute the item prices for each of the items.
- After finding the individual item prices, I calculated the order price for every order.
- I observed that certain rows had wrongly calculated order prices.
- A total of around 54 such rows were corrected.

Using this array, I have calculated the individual item prices.

In [ ]:
```
a=np.array([[0, 2, 0, 0, 0, 2, 0, 2, 0, 1],
 [0, 0, 0, 0, 0, 1, 1, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0, 1, 1, 2],
 [0, 0, 2, 2, 0, 0, 0, 1, 0, 2],
 [1, 0, 0, 0, 0, 2, 0, 0, 0, 1],
 [0, 1, 0, 1, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 2, 0, 0, 0],
 [2, 2, 1, 2, 0, 0, 0, 0, 0, 0],
 [0, 2, 0, 0, 0, 0, 0, 2, 0, 0],
 [0, 0, 1, 0, 0, 2, 0, 0, 2, 0]])
b = np.array([20260, 7770, 16225, 19755, 23900, 1655, 15850, 22440, 5310, 10170]
#x = np.linalg.solve(a,b)
```

- Now, I know the items and the price of the items in the dataset.
- Hence, for easier calculation, I have stored them into a dictionary.

In [ ]:
```
# Dictionary to store all the items and the prices
items_prices = {'Candle Inferno': 430.0,
                'pearTV': 6310.0,
                'Toshika 750': 4320.0,
                'Lucent 330S': 1230.0,
                'Thunder line': 2180.0,
                'Alcon 10': 8950.0,
                'iAssist Line': 2225.0,
                'iStream': 150.0,
                'Universe Note': 3450.0,
                'Olivia x460': 1225.0
                }
```

In [ ]:
```
items_prices_dict = dict(items_prices)
for each in items_prices:
    print(each,":",items_prices[each])
```

```
In [ ]: # Bar Plot of the various items and prices
        colors ='cm'

        keys = items_prices.keys()
        values = items_prices.values()
        plt.bar(keys, values, color = colors)
        plt.xticks(rotation = 90)

        plt.title('Item Prices')
        plt.xlabel('Items')
        plt.ylabel('Prices')
        plt.grid()
```

```
In [ ]: # List to store all the shopping items
        items = []

        # Adding the items from every shopping cart into a list
        my_dirty_data['shopping_cart'].apply(lambda x:items.append(x))
        items[0]
```

```
In [ ]: # For making a new figure
        plt.figure()
        # Plotting histogram for our order prices
        plt.hist(my_outlier_data['order_price'], color = 'blue')
        plt.title('Order Price Histogram')
        plt.xlabel('Order Price')
        plt.ylabel('Count')
        plt.grid()
```

```
In [ ]: # Function for calculating the order price
        def order_price(row):

            # Initial calculated order price
            calculated_order_price = 0

            # Iterating through the shopping cart
            for key, value in row.shopping_cart.items():
                calculated_order_price = calculated_order_price + items_prices[key]*valu

            # Returning the calculated total order price
            return calculated_order_price
```

```
In [ ]: # Calculating the order prices of the shopping cart manually
        my_dirty_data['calculated_order_price'] = my_dirty_data.apply(lambda x : order_pr
        #my_dirty_data[["order_price","calculated_order_price"]].head()
```

```
In [ ]: # Checking the correctness of the order price
        wrong_order_prices = my_dirty_data[(my_dirty_data.order_price != my_dirty_data.ca
        print(wrong_order_prices.shape)
        wrong_order_prices[["shopping_cart","order_price","calculated_order_price"]].hea
```

- Thus, we can see that there are 54 incorrect order prices.

In [ ]:
```
# Correcting the order_price
wrong_price = (my_dirty_data.order_price != my_dirty_data.calculated_order_price
my_dirty_data.loc[wrong_price,'order_price'] = my_dirty_data['calculated_order_p
```

In [ ]:
```
# Checking if the corrections have been made or not
my_dirty_data[["shopping_cart","order_price","calculated_order_price"]].head()
```

In [ ]:
```
# Checking if the corrections have been made or not
wrong_order_prices = my_dirty_data[(my_dirty_data.order_price != my_dirty_data.c
print(wrong_order_prices.shape)
```

2.1.5 Exploratory Data Analysis : Order Total

Examining the order total : Applying Discount + Delivery

- Here, I first defined defined a function in order to calculate the order total.
- First the discount is applied on the order price. Then, the delivery charges are added.
- Thus the order total = discounted order price + delivery charge
- I observed that there were some wrong order total calculations.
- A total of 54 such wrong order totals were corrected.

In [ ]:
```
# For making a new figure
plt.figure()
# Plotting histogram for our order totals
plt.hist(my_outlier_data['order_total'], color = 'cyan')
plt.title('Order Total Histogram')
plt.xlabel('Order Total')
plt.ylabel('Count')
plt.grid()
```

In [ ]:
```
# Function to check the order total values
def order_total(row):
    # Discount Value
    discount = (row["coupon_discount"])/100
    discount = 1 - discount

    # Initial calculated order price
    calculated_order_total = 0

    # Applying the discount
    temp_order_total = (row["order_price"]*discount)

    calculated_order_total = temp_order_total + row["delivery_charges"]
    # Rounding the result
    calculated_order_total = round(calculated_order_total,2)

    # Returning the calculated total order price
    return calculated_order_total
```

In [ ]:
```
# Calculating the order total of the shopping cart manually
my_dirty_data['calculated_order_total'] = my_dirty_data.apply(lambda x : order_t
my_dirty_data[["order_price","coupon_discount","delivery_charges","order_total",
```

In [ ]:
```
# Checking for incorrect order totals
incorrect_totals = my_dirty_data[((my_dirty_data.order_total != my_dirty_data.ca
print(incorrect_totals.shape)
incorrect_totals[["order_price","coupon_discount","delivery_charges","order_tota
```

In [ ]:
```
# Correcting the order_totals
wrong_totals = (my_dirty_data.order_total!=my_dirty_data.calculated_order_total)
my_dirty_data.loc[wrong_totals,'order_total'] = my_dirty_data['calculated_order_
```

In [ ]:
```
# Checking if the corrections have been made
my_dirty_data[["order_price","coupon_discount","delivery_charges","order_total",
```

In [ ]:
```
# Dropping all the extra columns which were used for order total
del my_dirty_data['calculated_order_total']
```

2.1.6 Exploratory Data Analysis : Customer locations

Examining the geographical locations of the customers:

Checking: Customer Coordinates

- First, I checked the geographical locations of the customers.
- Since Melbourne lies in the Southern Hemisphere:
    - The latitude values are lesser than zero
    - The longitude values are greater than zero
- Certain customer latitude and longitude values were exchanged.
- A total of around 27 such corrections were made.

In [ ]:
```
# Loading the warehouse locations
warehouse_data = pd.read_csv("warehouses.csv")
warehouse_data
```

In [ ]:
```
my_dirty_data[["customer_id","customer_lat","customer_long"]].head(10)
```

In [ ]:
```
# Checking for incorrect locations
swapped_locations = my_dirty_data[((my_dirty_data.customer_lat > 0) | (my_dirty_
print(swapped_locations.shape)
swapped_locations.head(2)
```

- Number of faulty rows: 27
- Here, we have 27 rows where the customer longitude and latitude values have been inter-changed.

In [ ]:
```
# https://www.thetopsites.net/article/58381333.shtml
# Correcting the customer locations
wrong_locations = ( (my_dirty_data.customer_lat > 0) | (my_dirty_data.customer_l
my_dirty_data.loc[wrong_locations, ['customer_lat', 'customer_long']] = my_dirty_
```

- We now swap the latitude and longitude locations in order to get the correct locations!

In [ ]:
```
swapped_locations = my_dirty_data[((my_dirty_data.customer_lat > 0) | (my_dirty_
print(swapped_locations.shape)
```

- Thus, the faulty customer locations have now been corrected!

2.1.7 Exploratory Data Analysis : Warehouse Names and locations

Examining the warehouse names and the distance to the nearest warehouse:

Checking : Warehouse names:

- Now, we need to find the warehouse that is closest to the customer.
- Hence, I have used the haversine distance formula to calculate the distance between two geographical locations.
- I have then calculated the distance to each of the warehouses for every customer.
- After getting these individual distances, I have then calculated the least distance.
- From this least distance, I have calculated the nearest warehouse.
- I observed that certain nearby warehouses have been wrongly named.
- A total of 22 such wrongly named nearby warehouses were corrected.

Checking : Distance to nearest warehouse

- After correcting the nearest warehouse names, I now checked if the distance to these nearest warehouses had been correctly calculated or not.
- I observed that a total of around 43 distances were incorrectly calculated.
- These distances were then corrected.

In [ ]:
```
w = dict( my_dirty_data.nearest_warehouse.value_counts() )

# Bar Plot of the various nearby warehouses
colors ='bgr'

keys = w.keys()
values = w.values()
plt.bar(keys, values, color = colors)
plt.xticks(rotation = 90)

plt.title('Various nearby warehouses')
plt.xlabel('warehouse names')
plt.ylabel('count')
plt.grid()
```

In [ ]:
```
# Viewing the data
my_dirty_data[["customer_id","customer_lat","customer_long","nearest_warehouse"]
```

Observations:

- Now, we have a table of customers and the nearest warehouse.

In [ ]:
```
# Merging the two dataframes in order to get the warehouse locations
# https://stackoverflow.com/questions/41463119/join-two-dataframes-on-common-colu
my_dirty_data['warehouse_lat'] = my_dirty_data.nearest_warehouse.map(warehouse_da
my_dirty_data['warehouse_long'] = my_dirty_data.nearest_warehouse.map(warehouse_
```

In [ ]:
```
# Viewing the contents
my_dirty_data[["customer_id","customer_lat","customer_long","nearest_warehouse",
```

- I have further merged the two tables : the dirty data table and the warehouse table.
- Now, we can view the following:
  - Customer
  - Customer location
  - Nearest warehouse
  - Nearesr warehouse location

In [ ]:
```
# Viewing the given distance to the nearest warehouse
my_dirty_data[["customer_id","customer_lat","customer_long","nearest_warehouse",
```

In [ ]:
```
# Viewing the warehouse locations
warehouse_data
```

In [ ]:
```
# Locations of Nickolson warehouse
Nickolson_lat = float(warehouse_data.loc[warehouse_data.names == 'Nickolson','la
Nickolson_lon = float(warehouse_data.loc[warehouse_data.names == 'Nickolson','lo

# Locations of Thompson warehouse
Thompson_lat = float(warehouse_data.loc[warehouse_data.names == 'Thompson','lat'
Thompson_lon = float(warehouse_data.loc[warehouse_data.names == 'Thompson','lon'

# Locations of Nickolson warehouse
Bakers_lat = float(warehouse_data.loc[warehouse_data.names == 'Bakers','lat'])
Bakers_lon = float(warehouse_data.loc[warehouse_data.names == 'Bakers','lon'])
```

HAVERSINE DISTANCE : Formula Used to calculate the distance

- I have used the Haversine Formula to calculate the distance between two geo-locations.
- As per the formula, when we have two points on a sphere along with their longitudes and latitudes, this formula determines the great circle distance between two given points.
- This formula is really helpful.
- We can use it to find the distance between the customer's location and the various warehouses.
- This further is used to determine the nearest warehouse to the customer.
- Finally, this formula can also be used to check the accuracy of the values of the calculated distances.

In [ ]:
```
# https://towardsdatascience.com/heres-how-to-calculate-distance-between-2-geoloc
def haversine_distance(lat1, lon1, lat2, lon2):
   # As per the specification provided
   r = 6378

   # Calculating the phi values
   phi1 = np.radians(lat1)
   phi2 = np.radians(lat2)

   # Calculating the delta and lambda values
   delta_phi = np.radians(lat2 - lat1)
   delta_lambda = np.radians(lon2 - lon1)

   # Distance calculated as per the haversine distance formula
   a = np.sin(delta_phi / 2)**2 + np.cos(phi1) * np.cos(phi2) *   np.sin(delta_la
   hav_distance = r * (2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a)))
   return np.round(hav_distance, 4)
```

In [ ]:
```
# Calculating the distance till the Nickolson warehouse
def nickolson_distance(row):

    cust_long = row.customer_long
    cust_lat = row.customer_lat

    distance_to_nickolson = haversine_distance(cust_lat, cust_long, Nickolson_la

    return distance_to_nickolson
```

In [ ]:
```
# Calculating the distance till the Thompson warehouse
def thompson_distance(row):

    cust_long = row.customer_long
    cust_lat = row.customer_lat

    distance_to_thompson = haversine_distance(cust_lat, cust_long, Thompson_lat,

    return distance_to_thompson
```

In [ ]:
```
# Calculating Bakers Distance
def bakers_distance(row):

    cust_long = row.customer_long
    cust_lat = row.customer_lat

    distance_to_bakers= haversine_distance(cust_lat, cust_long, Bakers_lat, Bake

    return distance_to_bakers
```

In [ ]:
```
# Calculating the distance to Nickolson
my_dirty_data['Nickolson_Distance'] = my_dirty_data.apply(lambda x: nickolson_di

# Calculating the distance to Thompson
my_dirty_data['Thompson_Distance'] = my_dirty_data.apply(lambda x: thompson_dist

# Calculating the distance to Nickolson
my_dirty_data['Bakers_Distance'] = my_dirty_data.apply(lambda x: bakers_distance
```

In [ ]:
```
# Calculating the least distance to the nearest warehouse
my_dirty_data['least_distance'] = my_dirty_data[['Nickolson_Distance','Thompson_
```

In [ ]:
```
# Assigning the nearest warehouse to every customer based on the least distance
def least_warehouse(row):
    if row.least_distance == row.Nickolson_Distance:
        return "Nickolson"
    elif row.least_distance == row.Thompson_Distance:
        return "Thompson"
    elif row.least_distance == row.Bakers_Distance:
        return "Bakers"
```

In [ ]:
```
# Calculating the nearest warehouse using the above function
my_dirty_data['least_warehouse'] = my_dirty_data.apply(lambda x: least_warehouse
```

In [ ]:
```
# Viewing
my_dirty_data[['least_warehouse','nearest_warehouse','Nickolson_Distance','Thomp
```

In [ ]:
```
# Identifying warehouses that have been wrongly named
wrong_warehouse = my_dirty_data[( my_dirty_data.least_warehouse !=  my_dirty_dat
print(wrong_warehouse.shape)

wrong_warehouse[['least_warehouse','nearest_warehouse','Nickolson_Distance','Tho
```

In [ ]:
```
# Correcting the Wrong Warehouse Names
wrong_warehouse_names = ( (my_dirty_data.least_warehouse !=  my_dirty_data.neare
my_dirty_data.loc[wrong_warehouse_names, ['nearest_warehouse']] = my_dirty_data.
```

In [ ]:
```
# Checking if the wrong warehouse names have been corrected or not
wrong_warehouse = my_dirty_data[( my_dirty_data.least_warehouse !=  my_dirty_dat
print(wrong_warehouse.shape)
```

In [ ]:
```
# Identifying the mistakes in distance to nearest warehouse
wrong_distance = my_dirty_data[( my_dirty_data.least_distance !=  my_dirty_data.
print(wrong_distance.shape)
wrong_distance[['least_warehouse','nearest_warehouse','Nickolson_Distance','Thom
```

In [ ]:
```
# Correcting the Wrong Warehouse Distances
wrong_warehouse_distance = ( (my_dirty_data.least_distance !=  my_dirty_data.dis
my_dirty_data.loc[wrong_warehouse_distance, ['distance_to_nearest_warehouse']] =
```

In [ ]:
```
# Checking if the wrong warehouse distances have been corrected or not
wrong_distance = my_dirty_data[( my_dirty_data.least_distance !=  my_dirty_data.
print(wrong_distance.shape)
```

2.1.8 Exploratory Data Analysis : Sentiment Analysis

Examining the customer reviews and happiness of the customer

- Here, I have examined the customer review and the customer happiness columns.
- I have used the sentiment analyzer in order to compute the sentiment polarity.
- 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.
  - **Positive Sentiment :** 'compound' polarity score >= 0.05

    - **Negative Sentiment :** 'compound' polarity score < 0.05
- If it's a positive sentiment, it means the customer is happy.
- If it's a negative sentiment, the customer is not happy.
- Now, I observed that there were some inconsistencies in the the customer reviews and the customer happiness.
- A total of 28 such errors were identified and corrected.

In [ ]:
```
# Are most of the customers happy?
c = dict( my_dirty_data.is_happy_customer.value_counts() )
print(c)

# Bar Plot of the various nearby warehouses
colors ='br'

keys = c.keys()
values = c.values()
plt.bar(keys, values, color = colors)
plt.xticks(rotation = 90)

plt.title('Customer Happiness')
plt.xlabel('Happy/Sad')
plt.ylabel('count')
plt.grid()
```

- Thus, we can say that in general, most of the customers are happy with their purchase.

In [ ]:
```
my_dirty_data[["latest_customer_review","is_happy_customer"]].head()
```

In [ ]:
```
my_dirty_data['latest_customer_review'] = my_dirty_data['latest_customer_review'
```

In [ ]:
```
# Function to calculate the compound polarity score
def sentiment_polarity(row):
    sid = SentimentIntensityAnalyzer()

    # Calculating polarity
    polarity = sid.polarity_scores(row)

    # Polarity score
    polarity_score = polarity['compound']

    # Returning the polarity score
    return polarity_score
```

In [ ]:
```
# Calculating the polarity scores for every order
my_dirty_data['polarity'] = my_dirty_data['latest_customer_review'].apply(lambda
```

In [ ]:
```
my_dirty_data[["latest_customer_review","is_happy_customer","polarity"]].head()
```

In [ ]:
```
# Identifying the mistakes in customer reviews
wrong_review = my_dirty_data[( (my_dirty_data['polarity']>0.05) & (my_dirty_data
print(wrong_review.shape)
wrong_review[["latest_customer_review","is_happy_customer","polarity"]].head()
```

In [ ]:
```
# Correcting the customer reviews
wrong_reviews = ( ( (my_dirty_data['polarity']>0.05) & (my_dirty_data['is_happy_
my_dirty_data.loc[wrong_reviews,'is_happy_customer'] = my_dirty_data['polarity']
```

In [ ]:
```
my_dirty_data[["latest_customer_review","is_happy_customer","polarity"]].head()
```

In [ ]:
```
wrong_review = my_dirty_data[( (my_dirty_data['polarity']>0.05) & (my_dirty_data
print(wrong_review.shape)
```

In [ ]:
```
# Identifying the mistakes in customer reviews where the reviews are no reviews
null_review = my_dirty_data[(my_dirty_data['latest_customer_review'].isnull())]
print(null_review.shape)
null_review[["latest_customer_review","is_happy_customer"]].head()
```

Task 2 : Handling Missing Values

- Here, we explore the missing data dataset to explore the different columns which have Null values.
- Clearly, there are null values in columns of warehouse names, distance to nearby warehouses, order price, delivery charges, order total and customer happiness index.
- I have explored each of these columns separately.
- ***Nearest Warehouse Names:***
  - Using the functions defined above, I calculated the distance to each of the warehouses, found the nearest warehouse and used these calculations to fill in the missing values.
- ***Distance to the nearest warehouses:***
  - Similarly, by using the functions defined above, I have even calculated the distance to the nearest warehouse and used these calculated values to fill in the missing values.
- ***Customer Happiness:***
  - There were around 40 rows with Null values.
  - Using the functions for calculating polarity, I have calculated the customer happiness values and have used these values to fill in the missing values.
- ***Delivery charges:***
  - Since the delivery charges change every season and are dependent on factors such as distance between the customer and the nearest warehouse, expedited delivery options and customer happiness, I have used a linear regression model to train the outlier dataset and used this model to predict the delivery charges.
- ***Order Price and Order Total:***
  - Here also, I have manually calculated the values using the functions defined above and have used these values to fill out the missing values.

In [ ]:
```python
# Loading the missing data dataset
my_missing_data = pd.read_csv("30945305_missing_data.csv")
my_missing_data.head(2)
```

In [ ]:
```python
# Understanding the missing data dataset
my_missing_data.shape
```

In [ ]:
```python
# Getting an idea about the descriptive statistics of the dataset
my_missing_data.describe()
```

In [ ]:
```python
# In order to get the number of records and data type of each column
my_missing_data.info()
```

Total Rows = 500

- Columns which have **Missing Values**
  - nearest_warehouse
  - order_price
  - delivery_charges
  - order_total
  - distance_to_nearest_warehouse
  - is_happy_customer

In [ ]:
```python
# In order to detect the number of missing values for every column
my_missing_data.isnull().sum()
```

In [ ]:
```python
# Storing the Null Values in a dictionary
null_values = dict( my_missing_data.isnull().sum() )

new = {k:v for k,v in null_values.items() if v != 0}

# Bar Plot of the various nearby warehouses
colors ='br'

keys = new.keys()
values = new.values()
plt.bar(keys, values, color = colors)
plt.xticks(rotation = 90)

plt.title('Customer Happiness')
plt.xlabel('Happy/Sad')
plt.ylabel('count')
plt.grid()
```

Thus, we can see that the following rows have the following number of missing values:

- nearest_warehouse : 55
- order_price : 15
- delivery_charges : 40

- order_total : 15
- distance_to_nearest_warehouse : 31
- is_happy_customer : 40

2.2.1 Missing Values : 1) Nearest Warehouse Names

In [ ]:
```
missing_warehouse = my_missing_data[my_missing_data['nearest_warehouse'].isnull(
missing_warehouse.head(2)
```

In [ ]:
```
# Number of rows with missing nearest warehouse
my_missing_data[my_missing_data['nearest_warehouse'].isnull()].shape
```

Observations:

- Thus, we can see that there are 55 rows with the nearest warehouse value missing.

Methodology to fix the values:

- I am using the functions defined above in order to calculate the distance to each warehouse.
- Further, these values are used to calculate the least distance and hence the nearest warehouse!

In [ ]:
```
# Calculating the distance to Nickolson
my_missing_data['Nickolson_Distance'] = my_missing_data.apply(lambda x: nickolso

# Calculating the distance to Thompson
my_missing_data['Thompson_Distance'] = my_missing_data.apply(lambda x: thompson_

# Calculating the distance to Nickolson
my_missing_data['Bakers_Distance'] = my_missing_data.apply(lambda x: bakers_dist

# Calculating the least distance to the nearest warehouse
my_missing_data['least_distance'] = my_missing_data[['Nickolson_Distance','Thomp

# Calculating the nearest warehouse using the above function
my_missing_data['least_warehouse'] = my_missing_data.apply(lambda x: least_wareh

my_missing_data[['least_warehouse','nearest_warehouse','Nickolson_Distance','Tho
```

In [ ]:
```
# Missing warehouse names
missing_warehouse = my_missing_data['nearest_warehouse'].isnull()

# Filling the missing values
my_missing_data.loc[missing_warehouse,'nearest_warehouse'] = my_missing_data.loc
```

In [ ]:
```
# Checking if the missing values have been added
my_missing_data[['least_warehouse','nearest_warehouse','Nickolson_Distance','Thom
```

In [ ]:
```
# Number of rows of missing warehouse names
my_missing_data[my_missing_data['nearest_warehouse'].isnull()].shape
```

Thus,

All the missing nearest warehouse names have been calculated and filled successfully!

2.2.2 Missing Values : 2) Distance to the Nearest Warehouse

In [ ]:
```
missing_distance = my_missing_data[my_missing_data['distance_to_nearest_warehouse
missing_distance.head(2)
```

In [ ]:
```
# Number of rows with missing distance to the nearest warehouse
my_missing_data[my_missing_data['distance_to_nearest_warehouse'].isnull()].shape
```

Observations:

- Thus, we can see that there are 31 rows with the distance to the nearest warehouse value missing.

Methodology to fix the values:

- I am using the functions defined above in order to calculate the distance to each warehouse.
- This later can be used to find the least distance

In [ ]:
```
my_missing_data[['least_warehouse','nearest_warehouse','Nickolson_Distance','Thom
```

In [ ]:
```
ng warehouse names
_distance = my_missing_data['distance_to_nearest_warehouse'].isnull()

ng the missing values
ing_data.loc[missing_distance,'distance_to_nearest_warehouse'] = my_missing_data.
```

In [ ]:
```
# Checking if the missing nearby distance values have been added or not
my_missing_data[['least_warehouse','nearest_warehouse','Nickolson_Distance','Thom
```

In [ ]:
```
# Number of rows with missing distance to the nearest warehouse
my_missing_data[my_missing_data['distance_to_nearest_warehouse'].isnull()].shape
```

Thus,

All the missing distance to the nearest warehouse values have been calculated and filled successfully!

2.2.3 Missing Values : 3) Customer Happiness

```
In [ ]: missing_review = my_missing_data[my_missing_data['is_happy_customer'].isnull()]
        missing_review.head(2)
```

```
In [ ]: # Number of rows with missing customer happiness review
        my_missing_data[my_missing_data['is_happy_customer'].isnull()].shape
```

Observations:

- Thus, we can see that there are 40 rows with the customer happiness values missing.

Methodology to fix the values:

- I am using the functions defined above in order to calculate the calculate the sentiment polarity.
- Based on these values, we can predict the customer happiness quotient.

```
In [ ]: my_missing_data['latest_customer_review'] = my_missing_data['latest_customer_revi
```

```
In [ ]: # Calculating the polarity scores for every order
        my_missing_data['polarity'] = my_missing_data['latest_customer_review'].apply(la
```

```
In [ ]: my_missing_data[["latest_customer_review","is_happy_customer","polarity"]].tail(
```

```
In [ ]: # Missing customer reviews
        missing_review = my_missing_data['is_happy_customer'].isnull()

        # New column for customer happiness
        my_missing_data['happy'] = my_missing_data['polarity'].apply(lambda x: 1 if x >=
        my_missing_data[["latest_customer_review","is_happy_customer","polarity","happy"
```

```
In [ ]: # Filling the missing customer happiness values
        my_missing_data.loc[missing_review,'is_happy_customer'] =  my_missing_data.loc[mi
        my_missing_data[["latest_customer_review","is_happy_customer","polarity","happy"
```

```
In [ ]: # Number of rows with missing customer happiness review
        my_missing_data[my_missing_data['is_happy_customer'].isnull()].shape
```

Thus,

All the missing customer happiness values have been calculated and filled successfully!

2.2.4 Missing Values : 4) Order Price

```
In [ ]: missing_price = my_missing_data[my_missing_data['order_price'].isnull()]
        missing_price.head(2)
```

```
In [ ]: # Number of rows with missing order price
        my_missing_data[my_missing_data['order_price'].isnull()].shape
```

Observations:

- Thus, we can see that there are 15 rows with the order price values missing.

Methodology to fix the values:

- I am using the functions defined above in order to calculate the order prices of the shopping cart.
- Based on these values, we can fill in the required missing values.

```
In [ ]: # Comverting the values of the shopping cart into a dictionary for easier evaluat
        my_missing_data['shopping_cart'] = my_missing_data['shopping_cart'].apply(lambda
        # Examining the top 2 elements of the shopping cart
        my_missing_data["shopping_cart"].head(2)
```

```
In [ ]: # Calculating the order prices of the shopping cart manually
        my_missing_data['calculated_order_price'] = my_missing_data.apply(lambda x : ord

        missing_price = my_missing_data[my_missing_data['order_price'].isnull()]
        missing_price[["shopping_cart","order_price","calculated_order_price"]].tail()
```

```
In [ ]: # Wrong order prices
        missing_price = ( my_missing_data['order_price'].isnull() )

        # Filling the missing order_price
        my_missing_data.loc[missing_price,'order_price'] = my_missing_data['calculated_or
```

```
In [ ]: my_missing_data[["shopping_cart","order_price","calculated_order_price"]].tail()
```

```
In [ ]: # Number of rows with missing order price
        my_missing_data[my_missing_data['order_price'].isnull()].shape
```

Thus,

All the missing order price values have been calculated and filled successfully!

2.2.5 Missing Values : 5) Delivery Charges

In [ ]:
```
delivery_charge = my_missing_data[my_missing_data['delivery_charges'].isnull()]
delivery_charge.head(2)
```

In [ ]:
```
# Number of rows with missing delivery charges
my_missing_data[my_missing_data['delivery_charges'].isnull()].shape
```

Observations:

- Thus, we can see that there are 40 rows with the delivery charges values missing.

Methodology to fix the values:

- We need to define functions that can predict the values of delivery charges.
- These values can then be used to fill in the missing delivery charges.

Calculation of delivery charge:

- The delivery charges are calculated using a linear model.
- The charge differs from season to season.
- It is dependent on the following factors:
    1. Distance between customer and nearest warehouse
    2. Whether or not the customer wants an expedited delivery
    3. Whether or not the customer was happy with his/her last purchase

Fixing the Summer Delivery Charges

In [ ]:
```
# Getting the data for the summer season
# Taking only the required columns on which the delivery charge is dependent on
Summer_data = my_outlier_data[ my_outlier_data['season']=='Summer'][['delivery_cl
Summer_data.shape
```

In [ ]:
```
# Reference : Taken from Wrangling Tutorial 8
# https://towardsdatascience.com/simple-and-multiple-linear-regression-in-python
# Instantiating the Linear Regression Model
lm = LinearRegression()
# Training the model based on the outlier dataset
lm.fit(Summer_data[[x for x in Summer_data.columns if x!= 'delivery_charges']], S
```

In [ ]:
```
summer_charge = (my_missing_data.season == 'Summer')&(my_missing_data.delivery_cl
# Applying the linear model to predict the delivery charges
lm.predict(my_missing_data[summer_charge][['delivery_charges','distance_to_neare
```

In [ ]:
```
null_index = list(my_missing_data['delivery_charges'][summer_charge].index)
# Filling in the missing values
my_missing_data.loc[null_index,'delivery_charges']= lm.predict(my_missing_data[su
```

Fixing the Autumn Delivery Charges

In [ ]:
```
# Getting the data for the autumn season
# Taking only the required columns on which the delivery charge is dependent on
Autumn_data = my_outlier_data[ my_outlier_data['season']=='Autumn'][['delivery_cl
Autumn_data.shape
```

In [ ]:
```
# Reference : Taken from Wrangling Tutorial 8
# Instantiating the Linear Regression Model
lm = LinearRegression()
# Training the model based on the outlier dataset
lm.fit(Autumn_data[[x for x in Autumn_data.columns if x!= 'delivery_charges']], A
```

In [ ]:
```
autumn_charge = (my_missing_data.season == 'Autumn')&(my_missing_data.delivery_cl
# Applying the linear model to predict the delivery charges
lm.predict(my_missing_data[autumn_charge][['delivery_charges','distance_to_neares
```

In [ ]:
```
null_index = list(my_missing_data['delivery_charges'][autumn_charge].index)
# Filling in the missing values
my_missing_data.loc[null_index,'delivery_charges']= lm.predict(my_missing_data[au
```

In [ ]:
```
#my_missing_data.isnull().sum()
```

Fixing the Winter Delivery Charges

In [ ]:
```
# Getting the data for the winter season
# Taking only the required columns on which the delivery charge is dependent on
Winter_data = my_outlier_data[ my_outlier_data['season']=='Winter'][['delivery_cl
Winter_data.shape
```

In [ ]:
```
# Reference : Taken from Wrangling Tutorial 8
# Instantiating the Linear Regression Model
lm = LinearRegression()
# Training the model based on the outlier dataset
lm.fit(Winter_data[[x for x in Winter_data.columns if x!= 'delivery_charges']], W
```

In [ ]:
```
winter_charge = (my_missing_data.season == 'Winter')&(my_missing_data.delivery_cl
# Applying the linear model to predict the delivery charges
lm.predict(my_missing_data[winter_charge][['delivery_charges','distance_to_neare:
```

In [ ]:
```
null_index = list(my_missing_data['delivery_charges'][winter_charge].index)
# Filling in the missing values
my_missing_data.loc[null_index,'delivery_charges'] = lm.predict(my_missing_data[w
```

In [ ]:
```
#my_missing_data.isnull().sum()
```

Fixing the Spring Delivery Charges

In [ ]:
```
# Getting the data for the spring season
# Taking only the required columns on which the delivery charge is dependent on
Spring_data = my_outlier_data[ my_outlier_data['season'] == 'Spring'][['delivery_
Spring_data.shape
```

In [ ]:
```
# Reference : Taken from Wrangling Tutorial 8
# Instantiating the Linear Regression Model
lm = LinearRegression()
# Training the model based on the outlier dataset
lm.fit(Spring_data[[x for x in Spring_data.columns if x!= 'delivery_charges']], S
```

In [ ]:
```
spring_charge = (my_missing_data.season == 'Spring')&(my_missing_data.delivery_cl
# Applying the linear model to predict the delivery charges
lm.predict(my_missing_data[spring_charge][['delivery_charges','distance_to_neare:
```

In [ ]:
```
null_index = list(my_missing_data['delivery_charges'][spring_charge].index)
# Filling in the missing values
my_missing_data.loc[null_index,'delivery_charges']= lm.predict(my_missing_data[s|
```

In [ ]:
```
#my_missing_data.isnull().sum()
```

Thus,

All the missing delivery charges have been calculated and filled successfully!

2.2.6 Missing Values : 6) Order Total

In [ ]:
```
missing_total = my_missing_data[my_missing_data['order_total'].isnull()]
missing_total.head()
```

In [ ]:
```
missing_total.shape
```

Observations:

- Thus, we can see that there are 15 rows with the order total values missing.

Methodology to fix the values:

- I am using the functions defined above in order to calculate the order totals.
- Based on these values, we can fill in the required missing values.

In [ ]:
```
# Calculating the order total of the shopping cart manually
my_missing_data['calculated_order_total'] = my_missing_data.apply(lambda x : ord
#my_missing_data[["order_price","coupon_discount","delivery_charges","order_tota
```

In [ ]:
```
# Missing order totals
missing_totals = ( my_missing_data['order_total'].isnull() )
# Filling in the Null Order totals
my_missing_data.loc[missing_totals,'order_total'] = my_missing_data['calculated_
```

In [ ]:
```
#my_missing_data.isnull().sum()
```

Thus,

All the missing order total values have been calculated and filled successfully!

Task 3 : Removing Outlier rows

- Here, I have used three major approaches to detect the outliers.
- The first approroch was by plotting the boxplots.
    - Here, just by observing the boxplot, I saw that there were values roughly below 30 and roughly above 105 which were the possible outliers.
    - A total of 40 outliers were detected.
    - However, I wanted to further ensure the correctness of these values.
- The second approach was by calculating the quantiles and thhe inter-quantile range.
    - Here, around 17 outliers were detected.
    - In this method, we were missing out certain outliers.
    - Hence, I decided to go for a method which would provide me a greater accuracy
- The third approach was by using a linear model.
    - According to the assignment specification, the delivery charges varied for every season. In every season, it was linearly dependent on three factors : expedited delivery, customer happiness, and distance to nearest warehouse.
    - Here, I came up with a linear model for every season and used this model to predict the delivery charges.
    - The accuracy of these models were around 0.99
    - I then calculated the charge differences of the actual and predicted values.
    - Finally, I calculated the quantiles of these differences in order to detect the outliers.

    - A total of 40 outliers were detected!
- These outliers were then removed from the dataset.

In [ ]:
```
# Loading the outlier data dataset
my_outlier_data = pd.read_csv("30945305_outlier_data.csv")
my_outlier_data.head(2)
```

In [ ]:
```
my_outlier_data.shape
```

2.3.1 Outlier Detection Approach 1 : Using Boxplots

- Here, in order to view the different outliers, I have plotted a box plot for the data.
- From the box plot, I could detect a total of 25 possible outliers.

In [ ]:
```
# # Plotting box-plots to view possible outliers
bp = my_outlier_data.boxplot()
plt.xticks(rotation = 45)
```

- We can see that there are outliers for a lot of attributes.
- However, our focus will be on the column of delivery charges.

In [ ]:
```
# Box-plot to view outliers only for the delivery charge
bp = my_outlier_data.boxplot(column = 'delivery_charges')
plt.xticks(rotation = 0)
```

In [ ]:
```
# Viewing the outliers grouped by season as the delivery charge is season depend
bp = my_outlier_data.boxplot(column = 'delivery_charges', by = "season")
plt.xticks(rotation = 0)
```

- Thus, we can see that there are many extreme values for the delivery charges.
- Outliers:
    - Approximately greater than 105
    - Approximately lesser than 40

In [ ]:
```
# Filtering out the data to view the possible outliers
outliers_plot = my_outlier_data[( my_outlier_data['delivery_charges'] > 105) | (
outliers_plot.shape
```

- Thus, roughly we have to detect around 40 outliers in this dataset.

2.3.2 Outlier Detection Approach 2 : Using Quantiles

- Here, I have calculated the lower and upper quantiles.
- Then, using the Inter Quantile range, I have calculated the lower and upper limits.

- I have then used these values to detect the outliers.

In [ ]:
```
# Generating the quantiles
# https://stackoverflow.com/questions/45926230/how-to-calculate-1st-and-3rd-quar
overall_quantiles = list(my_outlier_data.delivery_charges.quantile([0.25,0.5,0.7

# Vewing the quantiles
print("The Quantiles are:")
print("\n Q1 :",overall_quantiles[0], " \n Q2 :",overall_quantiles[1], " \n Q3 :
```

In [ ]:
```
# Calculating the IQ
IQR = round( overall_quantiles[2] - overall_quantiles[0], 2)

# Calculating the lower limit : Q1 -(1.5*IQR)
lower_range = round( overall_quantiles[0]-(1.5*IQR), 2)

# Calculating the Upper limit : Q3 +(1.5*IQR)
upper_range = round( overall_quantiles[2]+(1.5*IQR), 2)
```

In [ ]:
```
print("IQR:",IQR)
print("Lower Limit:",lower_range)
print("Upper Limit:",upper_range)
```

In [ ]:
```
# Filtering out the data in terms of quantiles
outliers_quantiles = my_outlier_data[( my_outlier_data['delivery_charges'] > upp
print("Outliers detected through quantiles:",outliers_quantiles.shape[0])
```

- Using this method, we can see that we have been able to detect only 17 outliers.
- We have may have missed out a few possible outliers.

In [ ]:
```
# Checking the possible in between outliers which we may have missed out
in_between = my_outlier_data[ ( (my_outlier_data['delivery_charges'] > 105) & (m
                              |( (my_outlier_data['delivery_charges'] > lower_ran
print("Possible number of in-between outliers:",in_between.shape[0])
```

2.3.3 Outlier Detection Approach 3 : Fitting a Linear Model

- Clearly, from the above method of quantiles, we can see that we are missing out on some possible outliers.
- Thus, we need a different method which is able to accurately detect the outliers.
- Hence, I have created linear models for every season.
- Since the linearity of our delivery charge changes every season, I have separated out the data for each season.
- I have then trained the model using the imputed missing dataset.
- I have then used this model to generate predicted values.

- Then, using the difference between the actual and predicted values, I have calculated quantiles.
- These quantiles are then used to calculate the lower and upper limits and calculate the possible outliers.

In [ ]:
```
# Creating a new figure
plt.figure()

# Plotting histogram for our delivery charges
plt.hist(my_outlier_data['delivery_charges'], color = 'orange')
plt.title('Delivery Charges Histogram')
plt.xlabel('Delivery Charges')
plt.ylabel('Count of the delivery charges')
plt.grid()
```

Observations from the Histogram:

- This histogram gives us a general overview of the distribution of the data.
- Thus, we can see that most of the values of the delivery charges lies between 60 and 100.
- There are a few values which appear roughly above 105 and some below 40.
- These are the outliers that we need to identify and remove!

Predicting Values for the Summer Season:

- Separate out the Summer data.
- Fit a linear model using the imputed missing dataset.
- Use this model to predict the delivery charge.

In [ ]:
```
# Getting the data for the summer season
# Taking only the required columns on which the delivery charge is dependent on
summer = my_missing_data[ my_missing_data['season'] == 'Summer'][['delivery_char

# Reference : Taken from Wrangling Tutorial 8
# Instantiating the Linear Regression Model
lm = LinearRegression()
# Training the model based on the imputed missing data dataset
lm.fit(summer[[x for x in summer.columns if x!= 'delivery_charges']], summer['de

summer_charge = (my_outlier_data.season == 'Summer')
# Applying the linear model to predict the delivery charges
summer_values = lm.predict(my_outlier_data[summer_charge][['delivery_charges','d

# Creating a new column for all the predicted values
my_outlier_data["predicted_delivery_charge"] = np.nan

# For inserting values only for the summer season
summer_index = list(my_outlier_data['delivery_charges'][summer_charge].index)
my_outlier_data.loc[summer_index,'predicted_delivery_charge'] = np.around(summer_
```

In [ ]:
```
summer_test = my_outlier_data[ my_outlier_data['season'] == 'Summer'][['delivery_

X = summer_test[[x for x in summer_test.columns if x!= 'delivery_charges']]
y = summer_test['delivery_charges']

summer_test_score = lm.score(X,y)
print("R-Squared for the Summer Model:",round(summer_test_score, 4))
```

In [ ]:
```
X = summer[[x for x in summer.columns if x!= 'delivery_charges']]
y = summer['delivery_charges']

summer_train_score = lm.score(X,y)
print("R-Squared for the Testing Autumn Model:",round(summer_train_score, 4))
```

R-Squared Values in general:

- R-Squared in general gives an indication of how close the data is to the fitted regression line.
- R-Squared value is the % of the variation of the response variable, which is determined by the linear model.
- Thus, R-squared Value = (Explained variation) / (Total variation)
- It is between 0% and 100%.
- Greater the value of R-Squared, greater is the accuracy of the model.

R-Squared Value here:

- Here, we have an high value of R-Squared Value.
- This indicates that the high accuracy of the linear model.

Predicting Values for the Autumn Season:

- Using the steps mentioned above, I have done the same for predicting the delivery charges for Autumn!

```
In [ ]: # Getting the data for the autumn season
        # Taking only the required columns on which the delivery charge is dependent on
        autumn = my_missing_data[ my_missing_data['season'] == 'Autumn'][['delivery_char
                                                                          'is_expedited_


        # Reference : Taken from Wrangling Tutorial 8
        # Instantiating the Linear Regression Model
        lm = LinearRegression()
        # Training the model based on the imputed missing data dataset
        lm.fit(autumn[[x for x in autumn.columns if x!= 'delivery_charges']], autumn['de

        autumn_charge = (my_outlier_data.season == 'Autumn')
        # Applying the linear model to predict the delivery charges
        autumn_values = lm.predict(my_outlier_data[autumn_charge][['delivery_charges','d
                                                                      'is_expedited_deli
        # For inserting values only for the autumn season
        autumn_index = list(my_outlier_data['delivery_charges'][autumn_charge].index)
        my_outlier_data.loc[autumn_index,'predicted_delivery_charge'] = np.around(autumn_
```

```
In [ ]: autumn_test = my_outlier_data[ my_outlier_data['season'] == 'Autumn'][['delivery_

        X = autumn_test[[x for x in autumn_test.columns if x!= 'delivery_charges']]
        y = autumn_test['delivery_charges']

        autumn_test_score = lm.score(X,y)
        print("R-Squared for the Training Autumn Model:",round(autumn_test_score, 4))
```

```
In [ ]: X = autumn[[x for x in autumn.columns if x!= 'delivery_charges']]
        y = autumn['delivery_charges']

        autumn_train_score = lm.score(X,y)
        print("R-Squared for the Testing Autumn Model:",round(autumn_train_score, 4))
```

R-Squared Value here:

- Value = 0.9898
- Here, we have an high value of R-Squared.
- This indicates that the high accuracy of the linear model.

Predicting Values for the Winter Season:

- Using the steps mentioned above, I have done the same for predicting the delivery charges for Winter!

In [ ]:
```
# Getting the data for the winter season
# Taking only the required columns on which the delivery charge is dependent on
winter = my_missing_data[ my_missing_data['season'] == 'Winter'][['delivery_charg
                                                                   'is_expedited_d


# Reference : Taken from Wrangling Tutorial 8
# Instantiating the Linear Regression Model
lm = LinearRegression()
# Training the model based on the imputed missing data dataset
lm.fit(winter[[x for x in winter.columns if x!= 'delivery_charges']], winter['de

winter_charge = (my_outlier_data.season == 'Winter')
# Applying the linear model to predict the delivery charges
winter_values = lm.predict(my_outlier_data[winter_charge][['delivery_charges','di
                                                               'is_expedited_deliv
# For inserting values only for the winter season
winter_index = list(my_outlier_data['delivery_charges'][winter_charge].index)
my_outlier_data.loc[winter_index,'predicted_delivery_charge'] = np.around(winter_
```

In [ ]:
```
winter_test = my_outlier_data[ my_outlier_data['season'] == 'Winter'][['delivery_
                                                                        'is_expedi

X = winter_test[[x for x in autumn_test.columns if x!= 'delivery_charges']]
y = winter_test['delivery_charges']

winter_test_score = lm.score(X,y)
print("R-Squared for the Testing Winter Model:",round(winter_test_score, 4))
```

In [ ]:
```
winter_train_score = lm.score(winter[[x for x in winter.columns if x!= 'delivery_
print("R-Squared for the Training Winter Model:",round(winter_train_score,4))
```

R-Squared Value here:

- Value = 0.9898
- Here, we have an high value of R-Squared.
- This indicates that the high accuracy of the linear model.

Predicting Values for the Spring Season:

- Using the steps mentioned above, I have done the same for predicting the delivery charges for Spring!

```
In [ ]: # Getting the data for the Spring season
        # Taking only the required columns on which the delivery charge is dependent on
        spring = my_missing_data[ my_missing_data['season'] == 'Spring'][['delivery_char
                                                                          'is_expedited_

        # Reference : Taken from Wrangling Tutorial 8
        # Instantiating the Linear Regression Model
        lm = LinearRegression()
        # Training the model based on the imputed missing data dataset
        lm.fit(spring[[x for x in spring.columns if x!= 'delivery_charges']], spring['de

        spring_charge = (my_outlier_data.season == 'Spring')
        # Applying the linear model to predict the delivery charges
        spring_values = lm.predict(my_outlier_data[spring_charge][['delivery_charges','d
                                                                      'is_expedited_deli
        # For inserting values only for the winter season
        spring_index = list(my_outlier_data['delivery_charges'][spring_charge].index)
        my_outlier_data.loc[spring_index,'predicted_delivery_charge'] = np.around(spring_
```

```
In [ ]: spring_test = my_outlier_data[ my_outlier_data['season'] == 'Spring'][['delivery_
                                                                                'is_exped
        X = spring_test[[x for x in spring_test.columns if x!= 'delivery_charges']]
        y = spring_test['delivery_charges']

        spring_test_score = lm.score(X,y)
        print("R-Squared for the Testing Spring Model:",round(spring_test_score, 4))
```

```
In [ ]: spring_test_score = lm.score(spring[[x for x in spring.columns if x!= 'delivery_
        print("R-Squared for the Training Spring Model:",round(spring_test_score,4))
```

R-Squared Value here:

- Value = 0.9947
- Here, we have an high value of R-Squared.
- This indicates that the high accuracy of the linear model.

```
In [ ]: # Viewing the predicted values
        my_outlier_data[['delivery_charges','predicted_delivery_charge','season']].head(
```

```
In [ ]: # Calculating the charge differences
        my_outlier_data['charge_difference'] = abs(my_outlier_data['delivery_charges'] -
        my_outlier_data[['delivery_charges','predicted_delivery_charge','charge_differen
```

In [ ]:
```
final_quantiles = list(my_outlier_data.charge_difference.quantile([0.25,0.5,0.75

# Vewing the quantiles
print("The Quantiles are:")
print(" \n Q1 :", round(final_quantiles[0],2),\
      " \n Q2 :", round(final_quantiles[1],2),\
      " \n Q3 :", round(final_quantiles[2],2) )
```

In [ ]:
```
IQR_final = round( final_quantiles[2] - final_quantiles[0], 2)
lower = round( final_quantiles[0]-(1.5*IQR_final), 2)
upper = round( final_quantiles[2]+(1.5*IQR_final), 2)

print("IQR", IQR_final)
print("Lower Limit:",lower)
print("Upper Limit:",upper)

outliers_final = my_outlier_data[( my_outlier_data['charge_difference'] > upper)
print("Outliers detected using the linear model:", outliers_final.shape[0])
```

- Thus, we have been able to successfully detect all the outliers using the linear model.

In [ ]:
```
my_outlier_data = my_outlier_data.drop(outliers_final.index, axis=0)
print("Number of rows after deleting the outliers:", my_outlier_data.shape[0])
```

In [ ]:
```
# Box-plot to view outliers only for the delivery charge
bp = my_outlier_data.boxplot(column = 'delivery_charges', by = 'season')
plt.xticks(rotation = 0)
```

Summary:

- Thus, we can see that a total of around 40 Outliers were eliminated.
- It can be verified from the boxplot as well!

Generating : Output Files

- Firstly while generating the output files, I have first deleted all the extra columns that I had created for analysing my dataset.
- This is because it's mentioned in the assignment specification that the output file must have the same number of columns as the input files.

In [ ]:
```
required_columns = ['order_id','customer_id','date','nearest_warehouse','shoppin
       'coupon_discount', 'order_total', 'season', 'is_expedited_delivery',
       'distance_to_nearest_warehouse', 'latest_customer_review',
       'is_happy_customer']
```

In [ ]:
```
# Generating the required output files as per the specification
# <student_id>_dirty_data_solution.csv
# <student_id>_outlier_data_solution.csv
# <student_id>_missing_data_solution.csv

my_dirty_data.to_csv('30945305_dirty_data_solution.csv', columns = required_colu
my_outlier_data.to_csv('30945305_outlier_data_solution.csv', columns = required_
my_missing_data.to_csv('30945305_missing_data_solution.csv', columns = required_
```

3. Discussion and Analysis

- **Importance of Cleaning Dirty Data:**
  - Here, we were able to find dirty data in most of the columns.
  - After performing the task of cleaning this data, I realised the importance of data cleaning.
  - This was a process of detecting the errors in an iterative manner.
    - The inconsistencies in the data were removed.
    - The quality of the data was improved.
  - This is the reason Data Scientists spend 60% of their time in Data Cleansing.
  - Because of the steps I followed to clean my data, I was able to correctly determine the nearest warehouse, order prices, order totals and an indication of customer's happiness.

- **Importance of correctly imputing missing values:**
  - Here, I was able to identify the missing data and impute values into it accordingly.
  - After performing this task, I realised the importance of the problem of missing data while doing data analysis.
  - If we ignore the problem of missing data :
    - We can have either an over/under estimation of the sample mean and variance values.
    - This will then result in incorrect data references and results.
  - Thus, the best solution is to ensure that there is no missing data!
- **Importance of removing outliers:**
  - Here, I was able to identify and remove all the outliers.
  - Outliers are observations which deviate a lot from other observations of the dataset.
  - I realised the importance of outliers after performing this task.
  - If we ignore the problem of outliers, it can lead to the following problems:
    - Outliers can cause an increase in the error variance.
    - Outliers can also decrease the normality, if they are distribued in an non-random manner.
    - Finally, they can also affect the statistical model assumptions.

4. Conclusion

- Thus, we can conclude that we have performed through data exploration of the dirty dataset. We have discovered and corrected various syntactical and semantic errors which were found in the various columns.
- The different errors such as inconsisties in spellings of seasons and warehouse names, date formats, customer locations, incorrectly marked seasons, wrongly marked nearby

warehouses, faulty nearest warehouse locations, inconsistencies in order price, order total and differences in the customer reviews and happiness were identified and successfully fixed in the dirty data.
- In conclusion, the different columns with missing values such as nearest warehouse names, distance, order price, delivery and customer happiness quotiensts were identified in the missing dataset. Using the functions defined, the values were then calculated and used to fill in the missing data.
- Thus, while going through the outlier data, the boxplots for the data wa plotted, outliers were first computed by calculating the lower and upper quantiles. Then, to model was further improved by generating a linear regression model. Finally, the outliers were then removed successfully from the model.

5. References

Michael Kramer (2020, June 28). Trending YouTube Video Statistics and Comments.
Retrieved from: https://stackoverflow.com/questions/26105804/extract-month-from-date-in-python/26105888 (https://stackoverflow.com/questions/26105804/extract-month-from-date-in-python/26105888)

Shawn Chin (2009, June 4). How to make a flat list out of list of lists?
Retrieved from: https://stackoverflow.com/questions/952914/how-to-make-a-flat-list-out-of-list-of-lists (https://stackoverflow.com/questions/952914/how-to-make-a-flat-list-out-of-list-of-lists)

Jezrael (2017, January 14). JOIN two dataframes on common column in python
Retrieved from: https://stackoverflow.com/questions/41463119/join-two-dataframes-on-common-column-in-python (https://stackoverflow.com/questions/41463119/join-two-dataframes-on-common-column-in-python)

Dario Radečić (2020, April 14). Here's How To Calculate Distance Between 2 Geolocations in Python
Retrived 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)

The SciPy community(2020, June 29). numpy.linalg.solve
Retrieved from:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html#numpy.linalg.solve (https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html#numpy.linalg.solve)

Adi Bronshtein (2017, May 9). Simple and Multiple Linear Regression in Python
Retrieved from: https://towardsdatascience.com/simple-and-multiple-linear-regression-in-python-c928425168f9 (https://towardsdatascience.com/simple-and-multiple-linear-regression-in-python-c928425168f9)

Ben Yo(2017, August 28). How to calculate 1st and 3rd quartiles?
Retrived from: https://stackoverflow.com/questions/45926230/how-to-calculate-1st-and-3rd-quartiles (https://stackoverflow.com/questions/45926230/how-to-calculate-1st-and-3rd-quartiles)

Gandreadis(2018, December 6). How to round each item in a list of floats to 2 decimal places? Retrieved from: https://stackoverflow.com/questions/5326112/how-to-round-each-item-in-a-list-of-floats-to-2-decimal-places (https://stackoverflow.com/questions/5326112/how-to-round-each-item-in-a-list-of-floats-to-2-decimal-places)

Ashwini Chaudhary(2013, June 13). Remove a dictionary key that has a certain value [duplicate] Retrieved from: https://stackoverflow.com/questions/17095163/remove-a-dictionary-key-that-has-a-certain-value (https://stackoverflow.com/questions/17095163/remove-a-dictionary-key-that-has-a-certain-value)