FIT5196 Assessment 2: Exploratory Data Analysis and Data Cleansing

Student Name: Md. Saadman Hossain

Student ID: 31043313

Introduction

This assessment involves fixing the anomalies in the dirty data, imputing the missing values for the missing data file as well as removing the outlier rows in the outlier data file.

More details for each task will be given in the following sections.

Importing required libraries and configuring some settings

In [ ]:

```
import pandas as pd

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.pylab as pylab
import datetime as dt
import numpy as np
from nltk.sentiment.vader import SentimentIntensityAnalyzer
from numpy.linalg import solve
from sklearn.linear_model import LinearRegression

%matplotlib inline
mpl.style.use( 'ggplot' )

from IPython.core.display import HTML
css = open('style/style-table.css').read() + open('style/style-notebook.css').read()
HTML('<style>{}</style>'.format(css))
```

First, load the dirty data using Pandas library

In [ ]:

```
dirty_data = pd.read_csv("31043313_dirty_data.csv")
```

In [ ]:

```
print (dirty_data.shape)
dirty_data.head(10)
```

In [ ]:

```
dirty_data.info()
```

In [ ]:

```
dirty_data.describe()
```

Obsevations

```
* 500 rows
* mistakes in cumtomer lat and long. It shows maximum latitude is 145 which is n
ot possible
*
```

In [ ]:

```
# Here we can see that problem is that for the below columns customer_long should be cu
stomer_lat as
#latitute cannot be more that 90. so just need to swap the column values for these rows

dirty_data[dirty_data['customer_lat']>90]
```

Categorical variables

In [ ]:

```
dirty_data.describe(include=['O'])
```

Observations

```
- customer_id not unique, has duplicates. one customer_id repeated twice. Here w
e cannot change anything as customer id may not be an unique identifier due to t
he possibility that a single customer can place multiple orders.
- date not unique, means multiple orders in same day.
- nearest warehouse has 6 uniques, should be 3. Here some values are capitalize
d, others are not.
- 8 unique season should be 4. Here some values are capitalized, others are not.
- latest_customer_review has all unique.
- shopping cart doesnt have all unique as 2 customers can have the same order (s
hopping cart items)
```

fixing nearest warehouse column

In [ ]:

```
dirty_data.nearest_warehouse.unique()
```

In [ ]:

```
dirty_data['nearest_warehouse'] = dirty_data['nearest_warehouse'].replace(['nickolson'
],'Nickolson')
dirty_data['nearest_warehouse'] = dirty_data['nearest_warehouse'].replace(['bakers'],'B
akers')
dirty_data['nearest_warehouse'] = dirty_data['nearest_warehouse'].replace(['thompson'],
'Thompson')
```

Here i replaced the lower case warehouse names with capitalized versions (eg. nickolson Nickolson) to fix the syntactical error.

In [ ]:

```
dirty_data.nearest_warehouse.unique()
```

In [ ]:

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

In [ ]:

```
dirty_data['season'] = dirty_data['season'].replace(['summer'],'Summer')
dirty_data['season'] = dirty_data['season'].replace(['winter'],'Winter')
dirty_data['season'] = dirty_data['season'].replace(['autumn'],'Autumn')
dirty_data['season'] = dirty_data['season'].replace(['spring'],'Spring')
```

Here i replaced the lower case season names with capitalized versions (eg. summer to Summer) to fix the syntactical error.

In [ ]:

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

Fixing date

In [ ]:

```
#dirty_data['date'].str.findall(r'(\d{4}-\d{2}-\d{2})')

dirty_data['date'].str.findall(r'^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-
9]|3[01])$')
```

In [ ]:

```
#s = dirty_data.date.str.contains(r'(\d{4}-\d{2}-\d{2})')


# finding the correct dates
s = dirty_data.date.str.contains(r'^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12]
[0-9]|3[01])$')

s[s != True] #the mixtakes in the date column
```

In [ ]:

```
# filtering out the data frame containing only the error dates

date_mistakes = dirty_data[~dirty_data.date.str.contains(r'^(19|20)\d\d[- /.](0[1-9]|1
[012])[- /.](0[1-9]|[12][0-9]|3[01])$')]

date_mistakes
```

For this part i am using a regex which matches dates in the correct format YYYY-MM-DD. I am outputting the rows where the dates dont match into a mistakes dataframe for correction of the error.

In [ ]:

```
# dates with year errors

year_errors = date_mistakes[date_mistakes.date.str.contains(r'-2019')]
year_errors
```

Here only taking the rows where we have year errors using a regex for matching date string ending with -2019

Fixing year errors

In [ ]:

```
# using to date time function to set format as
year_errors['date'] = pd.to_datetime(year_errors['date'],format='%d-%m-%Y')

year_errors

year_errors['date'] = year_errors['date'].dt.strftime('%Y-%m-%d')
```

Using the to_datetime pandas function to convert the date column then changing the format of the date to YYYY-MM-DD

In [ ]:

```
year_errors
```

In [ ]:

```
# replacing year errors in original data frame
new_dirty = year_errors.combine_first(dirty_data)
new_dirty.head(3)
```

Replacing year errors in original data frame using pandas combine_first method

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```
# dates with month errors

month_errors = date_mistakes[~date_mistakes.date.str.contains(r'-2019')]
month_errors
```

I am using regex from earlier to exclude the year mistakes and only take the month errors which i put into a month_error data frame for correction

In [ ]:

```
month_errors['date'] = pd.to_datetime(month_errors['date'],format='%Y-%d-%m')

month_errors['date'] = month_errors['date'].dt.strftime('%Y-%m-%d')

month_errors
```

Similar method used to correct the year errors, used to correct the month mistakes. Just converted the dates from YYYY-DD-MM to YYYY-MM-DD which is the desired format.

In [ ]:

```
# replacing month errors in original data frame
new_dirty = month_errors.combine_first(new_dirty)
new_dirty.head(30)
```

Updating the new_dirty dataset with the fixed dates for month errors.

Fixing customer_lat

In [ ]:

```
new_dirty[new_dirty['customer_lat']>90]
```

We can observe that some customer latitudes are more than 90 which int possible. we also observe that the longitudes are negative for these rows. This is probably a lexical error where someone swapped the lats and longs when inserting the data. Its a quick fix by just swapping the column values for these rows.

In [ ]:

```
# using loc function to use a condidition to swap the column values for the error latit
ude and longitude errors
condition = new_dirty['customer_lat']>90

new_dirty.loc[condition, ['customer_lat', 'customer_long']] = new_dirty.loc[condition,
['customer_long', 'customer_lat']].values
```

Using loc function to use a condidition to swap the column values for the customer latitude and longitude errors.

In [ ]:

```
# displaying a few fixed errors
new_dirty.iloc[[38,47,56,63], :]
```

Displaying a few fixed errors

fixing season

In [ ]:

```
# extracting month and adding a new column named month

new_dirty['month'] = pd.DatetimeIndex(new_dirty['date']).month
```

Extracting month and adding a new column named month

In [ ]:

```
new_dirty.columns
```

In [ ]:

```
# replacing new_season column with season names based on months of a season

for i in range(len(new_dirty)):
    if new_dirty.iloc[i,16] in [9,10,11]:
        new_dirty.iloc[i,11] = 'Spring'

for i in range(len(new_dirty)):
    if new_dirty.iloc[i,16] in [12,1,2]:
        new_dirty.iloc[i,11] = 'Summer'


for i in range(len(new_dirty)):
    if new_dirty.iloc[i,16] in [3,4,5]:
        new_dirty.iloc[i,11] = 'Autumn'


for i in range(len(new_dirty)):
    if new_dirty.iloc[i,16] in [6,7,8]:
        new_dirty.iloc[i,11] = 'Winter'
```

For this part i am using the month column created earlier to classify the seasons based on the provided link in the requirements. for instance, if month is 9,10 or 11 (september,october or november) that means that season needs to be Spring. This piece of code goes through all the rows where the month fullfills a certain criteria to classify the season.

In [ ]:

```
# dropping the month column as its not needed anymore
new_dirty = new_dirty.drop(columns=['month'])
```

In [ ]:

```
new_dirty.groupby('season').count()
```

Fixing is_happy_customer

In [ ]:

```
new_dirty.head(2)
```

In [ ]:

```
analyzer = SentimentIntensityAnalyzer()

# storing the compounds scores into a new column compound_score
new_dirty['compound_score'] = 0 # creating a new column and setting all values to 0
for i in range(len(new_dirty)):
    z = analyzer.polarity_scores(new_dirty.iloc[i,14])['compound']

    new_dirty.iloc[i,16] = z


# making a new is_happy_customer_2 column to store the boolean values resulting from cl
assifying the compound scores based on
# given parameter (>0.05 is True else false)
#new_dirty['is_happy_customer_2'] = ' '
for i in range(len(new_dirty)):
    if new_dirty.iloc[i,16]>=0.05:
        new_dirty.iloc[i,15] = True
    else:
        new_dirty.iloc[i,15] = False




# dropping the compound_score column as its not needed anymore
new_dirty = new_dirty.drop(columns=['compound_score'])
# k = analyzer.polarity_scores(new_dirty.iloc[0,14])

# k
```

Using the SentimentIntensityAnalyzer from nltk.sentiment.vader to claculate polarity scores for the reviews form customers. Here i go through each row, getting the compund polarity score and adding to the column compound_score. Next code takes compund scores and if the score is greater than equal to 0.05 it imputes True into is_happy_customer, else it imputes false. then i just dropped the compound_score column.

In [ ]:

```
new_dirty.head(2)
```

In [ ]:

```
new_dirty.groupby('is_happy_customer').count()
```

Fixing distance to nearest warehouse

In [ ]:

```

```

In [ ]:

```
warehouse = pd.read_csv("warehouses.csv")
warehouse
```

Reading the warehouse dataset from the csv file into a pandas dataframe

In [ ]:

```
from math import sin, cos, sqrt, atan2, radians
```

In [ ]:

```
# https://gist.github.com/rochacbruno/2883505?fbclid=IwAR1YyJJdaF625MjH1YtymEdHVXw_kW5Z
k4fTMMeOKKBirS-vVx32XiBLPxk

import math

def distance(origin, destination):
    lat1, lon1 = origin
    lat2, lon2 = destination
    radius = 6378 # kilometers

    dlat = math.radians(lat2-lat1)
    dlon = math.radians(lon2-lon1)
    a = math.sin(dlat/2) * math.sin(dlat/2) + math.cos(math.radians(lat1)) \
        * math.cos(math.radians(lat2)) * math.sin(dlon/2) * math.sin(dlon/2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
    d = radius * c

    return d
```

This is a function to calculater the distance between 2 coordinates using the latitude and longitude values of 2 coordinates.

In [ ]:

```
new_dirty.shape
```

In [ ]:

```
new_dirty.head(2)
```

In [ ]:

```
# here i am going through the dataset and filtering the dataset based on nearest_wareho
use by name.
# then i am applying the distance function taking the lat and long form the warehouse d
f and the lat and long for the customer.
# finallt for each step its replacing the 0 with distance values calculated in previous
steps

for i in range(len(new_dirty)):
    if new_dirty.iloc[i,3] == 'Nickolson':
        new_dirty.iloc[i,13] = distance((warehouse.iloc[0,1],warehouse.iloc[0,2]),(new_
dirty.iloc[i,7],new_dirty.iloc[i,8]))

for i in range(len(new_dirty)):
    if new_dirty.iloc[i,3] == 'Thompson':
        new_dirty.iloc[i,13] = distance((warehouse.iloc[1,1],warehouse.iloc[1,2]),(new_
dirty.iloc[i,7],new_dirty.iloc[i,8]))


for i in range(len(new_dirty)):
    if new_dirty.iloc[i,3] == 'Bakers':
        new_dirty.iloc[i,13] = distance((warehouse.iloc[2,1],warehouse.iloc[2,2]),(new_
dirty.iloc[i,7],new_dirty.iloc[i,8]))
```

Here i am going through the dataset and filtering the dataset based on nearest_warehouse by name. Then i am applying the distance function taking the lat and long form the warehouse df and the lat and long for the customer. Finally for each step its replacing distance_to_nearest_warehouse values with distance values calculated in previous steps.

In [ ]:

```
new_dirty.head(2)
```

In [ ]:

```
new_dirty.shape
```

Fixing order_price and order_total

In [ ]:

```
outlier_data = pd.read_csv("31043313_outlier_data.csv")
```

Reading the outlier dataset from the csv file into a dataframe

In [ ]:

```
# using the outlier dataset in this case as order price is not wrong in that dataset.


cart = outlier_data['shopping_cart'].head(10) # taking first 10 rows


cart
```

cart contains the first 10 shopping cart items.

In [ ]:

```
# order prices of those indeces

order_prices = outlier_data['order_price'].head(10) # taking first 10 rows

order_prices
```

order_prices contains the first 10 order prices which are the totals of the shopping cart before discounts are applied and delivery charge is added.

In [ ]:

```
type(order_prices)
```

In [ ]:

```
#cart.reset_index(drop=True)
```

In [ ]:

```
# converting string of tuples to a list of tuples
newl = [eval(cart[0]),eval(cart[1]),eval(cart[2]),eval(cart[3]),
        eval(cart[4]),eval(cart[5]),eval(cart[6]),eval(cart[7]),eval(cart[8]),eval(cart
[9])]


newl  # This is now a nested list containing tuples of product and order quantity
```

The shopping cart contains string data which needs to be converted into a nested list containing tuples of product and order quantity. I achieved this by using the eval function. newl is a list contating lists (shopping cart of a customer) which contains tuples (product and order quantity).

In [ ]:

```
newl[0][0][0] # i will be able to iterate through this nested list now
```

In [ ]:

```
newll = [] # an empty list

for i in newl:
    for z in i:
        newll.append(z[0])



unique_products = list(set(newll))

unique_products # unique products list
```

Appending only the products into a list then getting the unique product list by using the set function.

In [ ]:

```
# creating empty dataframe
new_cart = pd.DataFrame(0, index=np.arange(len(newl)), columns=unique_products) # colum
ns are the products of the store
new_cart
```

Creating an empty dataframe which has the products as column names and 10 empty rows filled with 0s.

In [ ]:

```
new_cart.columns
```

In [ ]:

```
newl
```

In [ ]:

```
# entering the order amount of each product

# Here each row represents an order which is each nested list in the newl list

first = newl[0]
for i in first:
    new_cart.iloc[0,new_cart.columns.get_loc(i[0])] = i[1]

second = newl[1]
for i in second:
    new_cart.iloc[1,new_cart.columns.get_loc(i[0])] = i[1]

third = newl[2]
for i in third:
    new_cart.iloc[2,new_cart.columns.get_loc(i[0])] = i[1]

fourth = newl[3]
for i in fourth:
    new_cart.iloc[3,new_cart.columns.get_loc(i[0])] = i[1]

fifth = newl[4]
for i in fifth:
    new_cart.iloc[4,new_cart.columns.get_loc(i[0])] = i[1]

sixth = newl[5]
for i in sixth:
    new_cart.iloc[5,new_cart.columns.get_loc(i[0])] = i[1]

seventh = newl[6]
for i in seventh:
    new_cart.iloc[6,new_cart.columns.get_loc(i[0])] = i[1]

eighth = newl[7]
for i in eighth:
    new_cart.iloc[7,new_cart.columns.get_loc(i[0])] = i[1]


ninth = newl[8]
for i in ninth:
    new_cart.iloc[8,new_cart.columns.get_loc(i[0])] = i[1]

tenth = newl[9]
for i in tenth:
    new_cart.iloc[9,new_cart.columns.get_loc(i[0])] = i[1]





new_cart
```

For this step, i am going through each shopping cart item and inputting the order quantity into the correct row and columns of the empty dataframe created earlier.

In [ ]:

```
# using linear algebra package to calculate prices for each product

from numpy.linalg import solve

vals = new_cart.values # to get numpy.ndarray square matrix

vals
```

Converting new_cart into a numpy.ndarray square matrix.

In [ ]:

```
type(vals)
```

In [ ]:

```
correct_prices = solve(vals,order_prices) # using solve function to calculate the produ
ct prices

correct_prices
```

Using linear algebra package to calculate prices for each product

In [ ]:

```
# converting numpy array to a list

correct_prices = correct_prices.tolist()
correct_prices
```

Converting numpy array to a list

In [ ]:

```
# dictionary for products as keys and product prices as values

dictionary = dict(zip(new_cart.columns, correct_prices))

dictionary
```

Dictionary for products as keys and product prices as values. This dictionary contains all the products and their corresponding prices.

In [ ]:

```
dirty_shop_cart = new_dirty['shopping_cart']

# converting string of tuples to a list of tuples

for i in range(len(dirty_shop_cart)):
    dirty_shop_cart[i] = eval(dirty_shop_cart[i])
```

Converting string of tuples to a list of tuples

In [ ]:

```
dirty_shop_cart[0]
```

In [ ]:

```
# creating empty list for the order totals
value_list = []

for i in dirty_shop_cart:
    product_list=[]
    for x in range(len(i)):

        product_list.append(i[x][0])
    print(product_list)

    order_quantities=[]
    for x in range(len(i)):

        order_quantities.append(i[x][1])
    print(order_quantities)

    value=0
    for y in range (len(product_list)):
        value+=(dictionary[product_list[y]]*order_quantities[y])
    print(value)

    value_list.append(value)
```

steps:

```
Firstly creating an empty list called value_list.
Then iterating through the dirty_shop_cart, creating another empty list inside t
he for loop.
next up i append each poduct in a shopping cart into a list and also append the
 quantities into a separate list.
value is a variable holds the calculated order_price for each row.
Then just append all the values into a list called value_list.
```

In [ ]:

```

```

In [ ]:

```
new_dirty.head(2)
```

In [ ]:

```
# updating the order price column

for i in range(len(new_dirty)):
    new_dirty.iloc[i,5] = value_list[i]
```

value_list contains all the order_price values. Here used a for loop to insert the calculated order prices for each row.

In [ ]:

```
new_dirty.head(2)
```

In [ ]:

```
new_dirty.columns
```

In [ ]:

```
# updating the order total column

# for each row remove the discount from the order_price then add the delivery charge
for i in range(len(new_dirty)):
    new_dirty.iloc[i,10] = new_dirty.iloc[i,5]*((100-new_dirty.iloc[i,9])/100)+new_dirt
y.iloc[i,6]

```

To calculate the order_total column values, for each row I removed the discount from the order_price then added the delivery charge.

In [ ]:

```
new_dirty.head(2)
```

In [ ]:

```
### output dirty data into a csv file

new_dirty.to_csv('31043313_dirty_data_solution.csv',index=False)
```

output dirty data into a csv file

In [ ]:

```

```

Working with the missing_data file

Imputing missing values

In [ ]:

```
missing_data = pd.read_csv("31043313_missing_data.csv")
```

Reading the missing data file from a csv into dataframe

In [ ]:

```
missing_data.info()
```

imputing missing values in order_price

In [ ]:

```
missing_order_price = missing_data[missing_data['order_price'].isnull()]
missing_order_price
```

Here i will use the dictionary of products and prices created earlier to impute the missing order prices column

In [ ]:

```
m_cart = []
for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,5]):
        m_cart.append(missing_data.iloc[i,4])

print(m_cart)

# converting string of tuples to a list of tuples

for i in range(len(m_cart)):
    m_cart[i] = eval(m_cart[i])
```

Here i am appending the shopping cart values (where order_price is missing) into a list by iterating through each shopping cart row in the dataframe.

Then converting the strings into nested list of tuples similar to what i did for previous tasks.

In [ ]:

```
# creating empty list for the order totals
value_list_1 = []

for i in m_cart:
    product_list=[]
    for x in range(len(i)):

        product_list.append(i[x][0])
    print(product_list)

    order_quantities=[]
    for x in range(len(i)):

        order_quantities.append(i[x][1])
    print(order_quantities)

    value=0
    for y in range (len(product_list)):
        value+=(dictionary[product_list[y]]*order_quantities[y])
    print(value)

    value_list_1.append(value) # this is a list containing all the order_price for the
 missing values
```

steps:

```
- Firstly creating an empty list called value_list_1.
- Then iterating through the dirty_shop_cart, creating another empty list inside
the for loop.
- next up i append each poduct in a shopping cart into a list and also append th
e quantities into a separate list.
- value is a variable holds the calculated order_price for each row.
- Then just append all the values into a list called value_list_1 created earlie
r.
```

In [ ]:

```
#indices for the missing order_prices

indexx = []
for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,5]):
        indexx.append(i)


indexx

# creating a dictionary with the indices for missing values as key and order_price as v
alue
index_dictionary = dict(zip(indexx, value_list_1))

index_dictionary
```

For this part im inserting the indices of the rows with missing order_price into a list. Then creating a dictionary using the indices as key and order_price values as values.

In [ ]:

```
index_dictionary[18]
```

In [ ]:

```
# imputing the missing values in order_price column using the dictionary created above

for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,5]):
        missing_data.iloc[i,5] = index_dictionary[i]
```

For this step i am just imputing the missing values in order_price column (of missing_data dataframe) using the dictionary created above.

In [ ]:

```
missing_data.info()
```

imputing the missing values in is_happy_customer

In [ ]:

```
missing_data[missing_data['is_happy_customer'].isnull()]
```

In [ ]:

```
#using SentimentIntensityAnalyzer to get compound polarity scores to classify is_happy_
customer as either 1 or 0.

analyzer = SentimentIntensityAnalyzer()


for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,15]):
        z = analyzer.polarity_scores(missing_data.iloc[i,14])['compound']
        if z>=0.05:
            missing_data.iloc[i,15] = 1
        else:
            missing_data.iloc[i,15] = 0
```

Using SentimentIntensityAnalyzer to get compound polarity scores to classify is_happy_customer as either 1 or 0. This is similar code to tasks done earlier.

In [ ]:

```
missing_data.info()
```

Imputing missing values in distance_to_nearest_warehouse and nearest_warehouse

In [ ]:

```
m = missing_data[missing_data['nearest_warehouse'].isnull()]

m
```

In [ ]:

```
# imputing missing values in the distance to nearest warehouse

for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,13]):
        distances = [] # empty list
        # distances from nicholson,thompson,bakers to customer
        distances.append(distance((warehouse.iloc[0,1],warehouse.iloc[0,2]),(missing_da
ta.iloc[i,7],missing_data.iloc[i,8])))
        distances.append(distance((warehouse.iloc[1,1],warehouse.iloc[1,2]),(missing_da
ta.iloc[i,7],missing_data.iloc[i,8])))
        distances.append(distance((warehouse.iloc[2,1],warehouse.iloc[2,2]),(missing_da
ta.iloc[i,7],missing_data.iloc[i,8])))

        missing_data.iloc[i,13] = min(distances) # minimum of the 3 distances imputed i
nto the column
```

Steps:

```
- iterating through the whole dataset using a for loop
- if distance_to_nearest_warehouse is null, then first create a empty list
- append Nicholsons, Thompson and bakers distances into the list
- get minimum of those 3 distances using min function on the list
- impute the distance_to_nearest_warehouse column with the minimum distance
- the lowest distance out of the 3 is the distance_to_nearest_warehouse for that
customer.
```

In [ ]:

```




# imputing missing values in nearest warehouse

for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,3]):
        distances = []
        # distances from nicholson,thompson,bakers to customer
        distances.append(distance((warehouse.iloc[0,1],warehouse.iloc[0,2]),(missing_da
ta.iloc[i,7],missing_data.iloc[i,8])))
        distances.append(distance((warehouse.iloc[1,1],warehouse.iloc[1,2]),(missing_da
ta.iloc[i,7],missing_data.iloc[i,8])))
        distances.append(distance((warehouse.iloc[2,1],warehouse.iloc[2,2]),(missing_da
ta.iloc[i,7],missing_data.iloc[i,8])))


        # a list containing the 3 warehouses in order, Nicholas, Thompson, Bakers.
        warehouse_names = warehouse.names.tolist() # list of warehouse names in order

        list_index = distances.index(min(distances)) # index of the minimum distance fr
om the distances list after one iteration
        # using the index of the distance to find the warehouse name from the warehouse
_names list which has the same index
        missing_data.iloc[i,3] = warehouse_names[list_index]
```

Steps:

```
- iterating through the whole dataset using a for loop
- if nearest_warehouse is null, then first create a empty list
- append Nicholsons, Thompson and bakers distances into the list
- create a list with the 3 warehouse names in order
- get minimum of those 3 distances using min function on the distances list. get
the index of that distance in the list
- get the warehouse name from the warehouse list with the same index (as warehou
se list is in order i will be able to get the name out using the index of the di
stance in the distance list)
- impute the nearest_warehouse column with the nearest warehouse name
- the lowest distance out of the 3 is the distance_to_nearest_warehouse for that
customer. And the name from the warehouse list is the nearest_warehouse.
```

In [ ]:

```
missing_data.info()
```

In [ ]:

```
missing_data[missing_data['delivery_charges'].isnull()]
```

Using linear regression to calculate delivery charge which will be used in order_total calculation

In [ ]:

```
missing_data['delivery_charges'].isnull().sum()
```

In [ ]:

```
# a dataframe containing rows where delivery charge is not null
not_null_deliv = missing_data[~missing_data['delivery_charges'].isnull()]
not_null_deliv.count()
```

not_null_deliv is a dataframe containing rows where delivery charge is not null.

In [ ]:

```
not_null_deliv['season'].unique().tolist()
```

In [ ]:

```
not_null_deliv.head(2)
```

In [ ]:

```
# stringindexing season
for i in range(len(not_null_deliv)):
    if not_null_deliv.iloc[i,11] == 'Autumn':
        not_null_deliv.iloc[i,11] = 0
    elif not_null_deliv.iloc[i,11] == 'Winter':
        not_null_deliv.iloc[i,11] = 1

    elif not_null_deliv.iloc[i,11] == 'Spring':
        not_null_deliv.iloc[i,11] = 2

    elif not_null_deliv.iloc[i,11] == 'Summer':
        not_null_deliv.iloc[i,11] = 3



# stringindexing is_expidited_delivery

for i in range(len(not_null_deliv)):
    if not_null_deliv.iloc[i,12] == True:
        not_null_deliv.iloc[i,12] = 1
    elif not_null_deliv.iloc[i,12] == False:
        not_null_deliv.iloc[i,12] = 0
```

Stringindexing season and is_expidited_delivery columns. this needs to be done for the linear regression.

In [ ]:

```
#Instantiate LinearRegression()

lm_for_imputing = LinearRegression()
```

Instantiate LinearRegression()

In [ ]:

```
lm_for_imputing.fit(not_null_deliv[[x for x in not_null_deliv.columns
                                   if x in ['season', 'distance_to_nearest_warehouse',
                                           'is_expedited_delivery' ,'is_happy_custome
r']]],
                    not_null_deliv['delivery_charges']) #fitting the data
```

Fitting the model with the relevant columns which are the predictors and the delivery charges column is the target variable which needs to be predicted.

In [ ]:

```
# dataframe containing only nulls for delivery charge
null_deliv = missing_data[missing_data['delivery_charges'].isnull()]
```

null_deliv is a dataframe containing rows where delivery charge is null.

In [ ]:

```
# stringindexing season
for i in range(len(null_deliv)):
    if null_deliv.iloc[i,11] == 'Autumn':
        null_deliv.iloc[i,11] = 0
    elif null_deliv.iloc[i,11] == 'Winter':
        null_deliv.iloc[i,11] = 1

    elif null_deliv.iloc[i,11] == 'Spring':
        null_deliv.iloc[i,11] = 2

    elif null_deliv.iloc[i,11] == 'Summer':
        null_deliv.iloc[i,11] = 3



# stringindexing is_expidited_delivery

for i in range(len(null_deliv)):
    if null_deliv.iloc[i,12] == True:
        null_deliv.iloc[i,12] = 1
    elif null_deliv.iloc[i,12] == False:
        null_deliv.iloc[i,12] = 0
```

Stringindexing season and is_expidited_delivery columns. this needs to be done for the linear regression.

In [ ]:

```
predicted_delivery_charges = lm_for_imputing.predict(null_deliv[[x for x in null_deliv.
columns
                                    if x in ['season', 'distance_to_nearest_warehouse',
                                            'is_expedited_delivery' ,'is_happy_custome
r']]])

predicted_delivery_charges
```

Using the predict function to predict the delivery charges for the missing data.

In [ ]:

```
delivery_list = predicted_delivery_charges.tolist()
```

Converting predictions into a list

In [ ]:

```
missing_data.head(2)
```

In [ ]:

```
#indices for the missing delivery_charges

indexx1 = []
for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,6]):
        indexx1.append(i)


indexx1
```

A list containing the indices of nulls in deliver_charge

In [ ]:

```
# creating a dictionary with the indices for missing values as key and predicted delive
ry charges as value
indexx1_dictionary = dict(zip(indexx1, delivery_list))

indexx1_dictionary
```

Creating a dictionary with the indices for missing values as key and predicted delivery charges as value

In [ ]:

```
# imputing the missing values in delivery_charges column using the dictionary created a
bove

for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,6]):
        missing_data.iloc[i,6] = indexx1_dictionary[i]
```

Imputing the missing values in delivery_charges column using the dictionary created above

In [ ]:

```
missing_data.head(6)
```

Imputing the missing values in order total column

In [ ]:

```
# updating the order total column

# for each row remove the discount from the order_price then add the delivery charge
for i in range(len(missing_data)):
    if pd.isnull(missing_data.iloc[i,10]):
        missing_data.iloc[i,10] = missing_data.iloc[i,5]*((100-missing_data.iloc[i,9])/
100)+missing_data.iloc[i,6]
```

For each row (if order_total is missing) remove the discount from the order_price then add the delivery charge. then impute the value into order total column.

In [ ]:

```
missing_data.head()
```

In [ ]:

```
### output missing data into a csv file

missing_data.to_csv('31043313_missing_data_solution.csv',index=False)
```

output missing data into a csv file

Fixing outlier data

In [ ]:

```
outlier_data = pd.read_csv('31043313_outlier_data.csv')
```

Reading outlier data into a dataframe

In [ ]:

```
outlier_data.describe()
```

Delivery charge column has quite a big range, 24.47 minimum to 164.31 maximum. As per the question, there are outliers present here. I will produce some boxplots to see which outliers need to be removed.

In [ ]:

```
outlier_data.head(2)
```

In [ ]:

```
from pylab import rcParams
rcParams['figure.figsize'] = 20, 10
```

Setting config for figures

In [ ]:

```
# boxplot of entire dataframe

boxplot = outlier_data.boxplot()
```

In [ ]:

```
# boxplot for delivery charges

boxplot = outlier_data.boxplot(column = ['delivery_charges'])

# Here can see that there are a decent few outliers for the delivery charges data

# Outliers are mainly starting to occur past the $120 delivery charge mark
```

Boxplot of delivery_charges

In [ ]:

```
# Checking outliers by is_expedited_delivery. Maybe this will give me some indication.

bp = outlier_data.boxplot(column='delivery_charges', by = 'is_expedited_delivery')
```

Boxplot of deliver charges by is_expected_delivery

We can see here that generally, non expedited deliveries have lower delivery charge costs which is to be expected. But we can also observe that these boxplots have more outliers. especially for the False category we have quite a few more outliers. There are probably other factors impacting the outliers.

In [ ]:

```
plt.scatter(data = outlier_data, x = 'distance_to_nearest_warehouse', y = 'delivery_cha
rges')
```

In [ ]:

```
def outlier_calc(col): # a function to calculate the lower and upper whisker of a boxpl
ot to find outliers
 sorted(col)
 Q1,Q3 = np.percentile(col , [25,75]) #Q1 and Q3 are lower and upper quartiles respecti
vely
 IQR = Q3 - Q1 # interquartile range
 lower_whisker = Q1 - (1.5 * IQR) # below this threshold lies the outliers
 upper_whisker = Q3 + (1.5 * IQR) # above this threshold lies the outliers
 return lower_whisker,upper_whisker


lower_whisker,upper_whisker = outlier_calc(outlier_data.delivery_charges)


outlier_data[(outlier_data.delivery_charges < lower_whisker) | (outlier_data.delivery_c
harges > upper_whisker)]
```

In [ ]:

```
print(lower_whisker)
print(upper_whisker)
```

DigiCO has different business rules depending on the season to match the different demand of each season. As per the question delivery charge varies depending on the season. So, i though about creating delivery charge boxplots for every season.

In [ ]:

```
outlier_data.boxplot(column='delivery_charges', by = 'season')
```

Here we can see that, delivery charges do vary as per season. Delivery charges are higher on average in spring and summer, and lower for autumn and winter. Thus when assesing if there are any outliers in the data, i am considering the season the order was placed in. this will allow me to eliminate the outliers based on the particular season's lower and upper whisker in the boxplot.

For Autumn season

In [ ]:

```
# For Autumn season

autumn = outlier_data[outlier_data['season'] == 'Autumn']


lower_whisker,upper_whisker = outlier_calc(autumn.delivery_charges)

a = autumn[(autumn.delivery_charges < lower_whisker) | (autumn.delivery_charges > upper
_whisker)]

a = a.index.tolist()

a

outlier_data = outlier_data.drop(a) # dropping the outlier rows
```

Steps:

```
- taking only the rows where season is autumn
- calculating lower and upper whisker using the funciton created above
- filtering out the outlier rows
- finding the index of those rows and converting it to a list
- dropping those rows form the outlier dataset using the drop function
```

In [ ]:

```
outlier_data
```

For Spring season

In [ ]:

```
# For Spring season

Spring = outlier_data[outlier_data['season'] == 'Spring']


lower_whisker,upper_whisker = outlier_calc(Spring.delivery_charges)

s = Spring[(Spring.delivery_charges < lower_whisker) | (Spring.delivery_charges > upper
_whisker)]


s = s.index.tolist()

s

outlier_data = outlier_data.drop(s) # dropping the outlier rows
```

Steps:

```
- taking only the rows where season is Spring
- calculating lower and upper whisker using the funciton created above
- filtering out the outlier rows
- finding the index of those rows and converting it to a list
- dropping those rows form the outlier dataset using the drop function
```

In [ ]:

```
outlier_data
```

For Summer season

In [ ]:

```
# For Summer season

Summer = outlier_data[outlier_data['season'] == 'Summer']


lower_whisker,upper_whisker = outlier_calc(Summer.delivery_charges)

ss = Summer[(Summer.delivery_charges < lower_whisker) | (Summer.delivery_charges > uppe
r_whisker)]


ss = ss.index.tolist()

ss

outlier_data = outlier_data.drop(ss) # dropping the outlier rows
```

Steps:

```
- taking only the rows where season is Summer
- calculating lower and upper whisker using the funciton created above
- filtering out the outlier rows
- finding the index of those rows and converting it to a list
- dropping those rows form the outlier dataset using the drop function
```

In [ ]:

```
outlier_data
```

For Winter season

In [ ]:

```
# For Winter season

Winter = outlier_data[outlier_data['season'] == 'Winter']


lower_whisker,upper_whisker = outlier_calc(Winter.delivery_charges)
w = Winter[(Winter.delivery_charges < lower_whisker) | (Winter.delivery_charges > upper
_whisker)]


w = w.index.tolist()

w

outlier_data = outlier_data.drop(w) # dropping the outlier rows
```

Steps:

```
- taking only the rows where season is Winter
- calculating lower and upper whisker using the funciton created above
- filtering out the outlier rows
- finding the index of those rows and converting it to a list
- dropping those rows form the outlier dataset using the drop function
```

In [ ]:

```
outlier_data
```

In [ ]:

```
### output outlier data into a csv

outlier_data.to_csv('31043313_outlier_data_solution.csv',index=False)
```

Output outlier data into a csv file

In [ ]:

```

```