31187366_ass2

October 18, 2020

1 FIT5196 Assessment 2

Student Name: PRASHANT JAJORIA

Student ID: 31187366 Date: 18/10/2020

Environment: Python 3.7.4 and Anaconda 4.8.4 (64-bit)

Libraries used: 1. **pandas** - for reading and writing CSV file, and manipulating Datafarme, included in Anaconda Python 3.7.4 2. **re** - for regular expression, included in Anaconda Python 3.7.4 3. **numpy** - for arithmetic operations and calculations on arrays, included in Anaconda Python 3.7.4 4. **matplotlib** - for visualizing data, included in Anaconda Python 3.7.4 5. **datetime** - for validating date, included in Anaconda Python 3.7.4 6. **math** - for using predefined mathematical functions, included in Anaconda Python 3.7.4 7. **nltk.sentiment.vader** - for identifying customer sentiment from the product review. 8. **sklearn.linear_model** - for making Regression model and prediction.

1.1 Introduction

The main goal of this assessment is Data cleansing, before it can be used for any Data analysis process. Data quality problems like Missing data, Inconsistent and faulty data, Outliers and Duplicates are to be identified, analysed and recified.

Syntactical Anomalies involving format and issues in values; Semantic Anomalies involving comprehensiveness and redundancy; Coverage Anomalies involving missing values are to be effectively handled for the given data.

Following are the requirement of the task: 1. Clean the Dirty data file by removing Syntactical Anomalies and Semantic Anomalies. 2. Fill in the missing data for Missing data file by using various techiques, sentiment analysis, appropriate Multiple Linear Regression models, etc. 3. Remove the outliers for the Delivery charges from the Outlier data file.

A step by step explanation of completing the requirements will be explained in the following code cells.

2 TASK 1 : Dirty Data

2.0.1 Import the Library

[1]:
```
import pandas as pd
import numpy as np
import re
import matplotlib.pyplot as plt
import datetime
import math

from nltk.sentiment.vader import SentimentIntensityAnalyzer
from sklearn.linear_model import LinearRegression
```

2.0.2 Read CSV

Read the Dirty data file using the `read_csv` file from Pandas library

[2]:
```
df1 = pd.read_csv('31187366_dirty_data.csv')
warehouses = pd.read_csv('warehouses.csv')
```

2.0.3 Inspect the data

Lets check the first 5 rows using the `head` function to get a hang of the data provided.

[184]:
```
df1.head()
```

Lets look at the warehouse details given to us.

[185]:
```
warehouses
```

2.0.4 Look at the shape of Dataframe

[186]:
```
df1.shape
```

The shape of the Dataframe tell us that there are 500 rows i.e. orders and 16 columns i.e. order details

2.0.5 Statistics for Numerical variables

Using the `describe` function to view some basic statistical details like percentile, mean, std etc. of the read CSV file.

[189]:
```
df1.describe()
```

2.0.6 Look at Datatypes

Lets have a look at the data type of the columns of the CSV file

[190]:
```
df1.dtypes
```

2.0.7 Statistics for Categorical variables

As decribe by default shows stastics for only numerical data. To see the statistics of Categorical data we use `include` parameter of `describe` to include Datatype of type Object i.e. categorical data.

[191]:
```
df1.describe(include = ['O'])
```

2.1 Do we have duplicate customer_id ?

As the number of unique **customer_id** is 494 out of the total 500 orders, we need to check if the customer order is getting repeated. To check for Duplicate, we look at the **customer_id**, **shopping_cart** and **latest_customer_review**.

[192]:
```
df1[df1.duplicated(["customer_id", "shopping_cart","latest_customer_review"],␣
↪keep=False)]
```

As we get no data for this query, we can say that the orders are not duplicate and are different orders.

2.2 More than 3 warehouses ?

The count of unique warehouses returend was 6, so we need to check the warehouses values. As we only have 3 warehouses, there seems to be a problem here.

[193]:
```
df1.nearest_warehouse.unique()
```

We can see that the values of warehouses have been lower cased in some orders, giving us 6 warehouses.

2.2.1 Function to check if warehouse name is incorrect

We know the correct value of the 3 warehouses, so we can define a check to see the values that have name of Warehouses other than the defined warehouses.

[11]:
```
def check_incorrect_warehouses_name(warehouse_name):
    '''
    Function to check if warehouse name is from the 3 predefined warehouses.
    '''
    if warehouse_name not in ['Nickolson', 'Bakers', 'Thompson']:
        return True
    else:
        return False
```

[194]:
```
df_incorrect_nearest_warehouse = df1[df1['nearest_warehouse'].
↪apply(check_incorrect_warehouses_name)]
```

2.2.2 Replace the lower case warehouse

Correcting the warehouses names, where the values are in lower case

[13]:
```
df1.nearest_warehouse.replace({'nickolson':'Nickolson', 'bakers':'Bakers',␣
↪'thompson' : 'Thompson'}, inplace=True)
```

2.2.3 Check after replace

[195]:
```
df1.nearest_warehouse.unique()
```

2.3 Check the season

From the describe() function we got the number of unique values for season as 8, this tells us that there is a anamoly in some orders.

[196]:
```
df1.season.unique()
```

We can see that some of the orders have been lower cased giving us 8 different values for season.

2.3.1 Function to check if Season is incorrect

As we know the 4 seasons, we can define a check for finding flaws in season column.

[16]:
```
def check_incorrect_season_name(season_name):
    '''
    Function to check if the season name is incorrect.
    '''
    if season_name in ['summer', 'spring', 'winter']:
        return True
    else:
        return False
```

[197]:
```
df1[df1['season'].apply(check_incorrect_season_name)].shape
```

There are 12 order that have season value that is in lower case.

[18]:
```
df_incorrect_season = df1[df1['season'].apply(check_incorrect_season_name)]
```

2.3.2 Replace with correct Season

Correcting the lower cased season name to proper values.

[19]:
```
df1.season.replace({'summer':'Summer', 'autumn':'Autumn', 'spring' : 'Spring',␣
↪'winter' : 'Winter'}, inplace=True)
```

2.3.3 Check after replace

[198]:
```
df1.season.unique()
```

After correcting the values we get the correct number of seasons i.e. 4

2.4 Check Date format

We know the Date format for order is **YYYY-MM-DD**.

2.4.1 Check if Year is correct

As know the date format is **YYYY-MM-DD**, we can define a check for year, to see if first 4 numbers represent year or not.

[21]:
```
def check_year(order_date):
    '''
    Function to check if the Year is the first part of YYYY-MM-DDD format or␣
↪not.
    '''
    if len(order_date.split('-')[0]) != 4:
        return True
    else:
        return False
```

2.4.2 Number of rows which have the DD-MM-YYYY format

Looking at the data which do not pass the check for year, we can see that they are in **DD-MM-YYYY** format.

[199]:
```
df1[df1['date'].apply(check_year)].shape
```

There are 14 row that are in **DD-MM-YYYY** format.

[23]:
```
df_dd_mm_yyyy = df1[df1['date'].apply(check_year)]
```

2.4.3 Check if Month is correct

As know the date format is **YYYY-MM-DD**, we can define a check for month, to see if first 6 and 7 digit represent month or not.

[24]:
```
def check_month(order_date):
    '''
    Function to check if the Month is the second part of YYYY-MM-DDD format or␣
↪not.
    '''
    if int(order_date.split('-')[1]) > 12:
        return True
    else:
```

```
        return False
```

2.4.4 Number of rows which have the YYYY-DD-MM format

Looking at the data which do not pass the check for month, we can see that the date here is in **YYYY-DD-MM** format.

[200]:
```
df1[df1['date'].apply(check_month)].shape
```

There are 13 orders having date in **YYYY-DD-MM** format.

[201]:
```
df1['date'].apply(check_month).describe()
```

[27]:
```
df_yyyy_dd_mm = df1[df1['date'].apply(check_month)]
```

2.4.5 Fixing the Date Format issue

As we know the check conditions for both year and month, we can impute the order date that are not in **YYYY-MM-DD** format

[28]:
```
def impute_date(order_date):
    '''
    Function to impute the incorrect date and make it in the YYYY-MM-DDD format.
    '''
    if len(order_date.split('-')[0]) != 4:
        dd,mm,yyyy = order_date.split('-')
        return yyyy + '-' + mm + '-' + dd

    elif int(order_date.split('-')[1]) > 12:
        yyyy,dd,mm = order_date.split('-')
        return yyyy + '-' + mm + '-' + dd
    else:
        return order_date
```

[29]:
```
df1['date'] = df1['date'].apply(impute_date)
```

2.4.6 Check the new Date column

2.4.7 Check the year

Checking for errors in year after imputation.

[202]:
```
df1[df1['date'].apply(check_year)]
```

2.4.8 Check the month

Checking for errors in month after imputation.

[203]:
```
df1[df1['date'].apply(check_month)]
```

2.4.9 Validate the complete date

After correcting the year and month of the orders, we must check the complete validity of the Date.

This can be done using **datetime** python module

Defining a function to check if date is valid.

[32]:
```
def is_valid_date(date):
    '''
    Validate the order_date after imputations.
    '''
    year = date.split('-')[0]
    month = date.split('-')[1]
    day = date.split('-')[2]

    try :
        datetime.datetime(int(year),int(month),int(day))
        return False
    except ValueError :
        return True
```

[204]:
```
df1[df1['date'].astype(str).apply(is_valid_date)]
```

As we get no data for above query, we can be sure that all the dates are valid.

2.5 Checking the Shopping Cart

2.5.1 Find the names of all the Items of the shopping cart

We know that the store sells only 10 items, so we can check the shopping cart to see if the unique number of items is 10 or not.

Using regular expression to capture all the item names. The expression captures all the values that are in between single quotes.

Storing all the item names in a List.

[34]:
```
list_items = []

def get_item_names(cart):
    '''
    Function to get item name of a shopping cart.
    '''
    for item in re.findall(r'\'(.*?)\'',cart):
        list_items.append(item)

    return re.findall(r'\'(.*?)\'',cart)
```

2.5.2 Store the name of all the Itmes of shopping cart

Storing the shopping cart items as new column in dataframe.

[35]:
```
df1['shopping_items'] = df1['shopping_cart'].apply(get_item_names)
```

2.5.3 Names of 10 items

As we get the number of unique items as 10, we can be sure that the item name in the shopping cart are correct.

[205]:
```
len(set(list_items))
```

2.5.4 Store the item quantity ordered as new column

Defing a function to capture the item quantity of each shopping cart.

[37]:
```
def get_item_qty(cart):
    '''
    Function to get item quantity of each of the items of a shopping cart.
    '''
    return list(re.findall(r'\(\'\w+\s?\d*\w*\',\s(.*?)\)',cart))
```

[38]:
```
df1['shopping_item_qty'] = df1['shopping_cart'].apply(get_item_qty)
```

2.5.5 Finding the price of each of the 10 items

The **order_total** column has the total price for an order. This price can be incorrect for some orders. To ensure we have correct value for each order, we need to get the individual price of each Item.

To get the price of each Item, I use Linear algebra.

In this I am solving Linear equations in Two variables to get the price of Two items at a time.

[39]:
```
lst = {}

dict_item_price = {}

lst_items_names = set(list_items)

# make a dict to store the price of the items
for item_name in list(lst_items_names):
    dict_item_price[item_name] = []

# Solve linear equation for each of the 2 combinations of the orders.
# first order
for index, row in df1.iterrows():

    # first eq coefficients
```

```
    a_coeff =  re.findall(r'\(\'\w+\s?\d*\w*\',\s(.*?)\)',row['shopping_cart'])
    #print(a_coeff)

    # first equation RHS
    a_rhs = row['order_price']

    # make a adictionary to store the order_id of rows that have same type of␣
↪order
    lst[row['order_id']] = []

    # get the name of all the items for current order
    lst_items = re.findall(r'\'(.*?)\'',str(row['shopping_items']))

    # second order
    for index2, row2 in df1.iterrows():

        # we check only for cart with two items
        if ((row['shopping_items'] == row2['shopping_items']) and␣
↪len(row['shopping_items']) == 2 ):

            lst[row['order_id']] = lst[row['order_id']] + [row2['order_id']]

            # second equation coefficients
            b_coeff =  re.findall(r'\(\'\w+\s?\d*\w*\',\s(.*?
↪)\)',row2['shopping_cart'])

            # second equation RHS
            b_rhs = row2['order_price']

            # go ahead if the 2 equations have same number of coefficients
            if (len(a_coeff) == len(b_coeff)):

                # 2 coefficients of the linear equation
                coeff = np.array([a_coeff, b_coeff],dtype='float')

                # constant on the RHS of the linear equation.
                rhs = np.array([a_rhs,b_rhs])

                # using try as we will get exceptions in case the coefficients
                # are same for the two linear equations
                try:
                    # equation result
                    x = np.linalg.solve(coeff, rhs)
                except np.linalg.LinAlgError:
                    continue
                else:
                    # storing the price of the 2 items in a dictionary
```

```
                        dict_item_price[lst_items[0]].append(x[0])
                        dict_item_price[lst_items[1]].append(x[1])
```

2.5.6 Count of Prices of Each Items

After solving the Linear eauations, the Price of items we get is as a Dictionary storing the list of price for each of the 10 items.

Looking at the count of Price for each Item, we can find the correct item price of each of the 10 items.

The item price which has the maximum count will be the correct price for that item.

[206]:
```
# Dictionary to store the final item prices
final_item_prices = {}

# traverse all the items
for item_name,values in dict_item_price.items():

    # creating dict of count of prices for each item
    data = dict((x,dict_item_price[item_name].count(x)) for x in␣
↪set(dict_item_price[item_name]))
    print(item_name,data)

    # get the maximum count of prices of each item stored in the dictionary.
    max_count = max(data.values())

    # storing the correct item price in a new dictionary with key as Item name␣
↪and value as the item price
    for key2,value in data.items():
        if value == max_count:
            final_item_prices[item_name] = key2
```

2.5.7 Final price of the 10 items

Correct item price of each of the 10 items sold by the Store

[207]:
```
final_item_prices
```

2.6 Calculate the order_price

Now we can calculate the **order_price** for each item as we have the price of each Item of the Shopping cart

[42]:
```
def get_order_price(cart):
    '''
    Function to calculate the order_price
    '''
```

```
    # get the name of all the items in the shopping cart for an order
    list_items = re.findall(r'\'(.*?)\'',cart)

    # get the corresponding item quantity of each item for the order
    list_quantity = re.findall(r'\(\'\w+\s?\d*\w*\',\s(.*?)\)',cart)

    # as we know the item names and quantity is for the same order
    # we can combine them together to form a tuple
    item_order = zip(list_items,list_quantity)

    order_price = 0

    # calculate the order_price
    for order in item_order:
        order_price = order_price + float(final_item_prices[order[0]]) *␣
↪float(order[1])

    return int(order_price)
```

Store the calculated order_price as a new column

[43]:
```
df1['check_order_price'] = df1['shopping_cart'].apply(lambda x :␣
↪get_order_price(x) )
```

Lets inspect the orders that have different order_price than calcluated

[44]:
```
df_incorrect_order_price = df1[ df1['check_order_price'] != df1['order_price'] ]
```

There are **53 rows** that have incorrect **order_price**

Other possiblity is that the **shopping items** can be wrong for these 53 order.

[208]:
```
df_incorrect_order_price.shape
```

2.7 Shopping Cart items CORRECT - order_price not calculated properly

The order_total and shopping_cart is correct for these orders.

Out of the 53 rows, for the **29 rows** calculation of **order_price is not correct**

We need to calculate the order_price using the individual item prices

[209]:
```
df_corr_cart_incorr_order_price =␣
↪df_incorrect_order_price[df_incorrect_order_price['order_price'] !=␣
↪((df_incorrect_order_price['order_total'] -␣
↪df_incorrect_order_price['delivery_charges'])/
↪(100-df_incorrect_order_price['coupon_discount']))*100]
df_corr_cart_incorr_order_price.shape
```

2.7.1 Impute order_price for the 29 rows where order_price is incorrect

As we know that the shopping cart items are correct for these 29 orders, we can calculate the correct order_price by using the indicidual item price and quantity of the order.

Imputing the order_price for the 29 orders in the original dataframe.

[47]:
```
lst_imputed_prices = []

# for the 29 orders impute the order_price
for x in list(df_corr_cart_incorr_order_price.index):

    # get the item name for each order
    item_name = get_item_names(df1.loc[x,'shopping_cart'])

    # get item qty for each order
    item_qty = get_item_qty(df1.loc[x,'shopping_cart'])

    orders = zip(item_name,item_qty)

    order_price = 0

    # calculate the total price of a order
    for order in tuple(orders):
        order_price = order_price + final_item_prices[order[0]] *␣
↪float(order[1])

    # replace in the original dataframe
    df1.loc[x,'order_price'] = order_price

    lst_imputed_prices.append(order_price)
```

2.7.2 Check after imputation

Checking the order price after imputatuion using a Scatter plot.

As we know that the order_price and order_total have a linear relation, we can use the Scatter plot to verify whether our imputation is correct or not.

[210]:
```
# plot of old order_total v/s newly imputed order_price

plt.xlabel('order_total')
plt.ylabel('order_price')
plt.
↪plot(list(df_corr_cart_incorr_order_price['order_total']),lst_imputed_prices,'.
↪r')
```

The order_price and order_total has a straight line which means the imputation is correct for order_price.

2.7.3 Reverse calculate the order_price

2.7.4 Shopping cart items WRONG - order_price correct

As we can reverse calculate the **order_price** from order_total and delivery_charges, we can be certain that the order_price is correct, and the items are wrong in the shopping cart.

[49]:
```
df_rel_holds = df_incorrect_order_price[df_incorrect_order_price['order_price']␣
↪== ((df_incorrect_order_price['order_total'] -␣
↪df_incorrect_order_price['delivery_charges'])/
↪(100-df_incorrect_order_price['coupon_discount']))*100]
```

Out of the 53 orders, **24 orders** have the correct order_price.

This means that the **shoppping cart items are wrong** for these orders

2.7.5 Function to find the correct cart item for these 24 orders

As we know the correct order_price for the order, we can us this as a condition to identify the incorrect item of the cart.

The find the correct item of the order, I substitute each of the item of the shopping cart with the 10 items the store sells.

The item for which the true order_price matches the calculated order_price after substitution, is the correct item for that order.

[211]:
```
# traverse each of the rows
for index, row in df_rel_holds.iterrows():

    # flag
    correct_item_found = False

    # substitute each of the 10 items in shopping cart
    for item,price in final_item_prices.items():

        # get the items of this order
        items = get_item_names(row['shopping_cart'])

        # get the qty of this order
        qty = get_item_qty(row['shopping_cart'])

        # flag for items in the cart
        i = 0

        # do the check for each of the items of the cart
        while i < len(items):

            # get the items again as the items have been modified
```

```
            items = get_item_names(row['shopping_cart'])

            # substitute each of the item of the cart
            items[i] = item

            # zip the item and qty to form a tuple
            items_tuple = zip(items,qty)

            # intialize
            check_order_price = 0

            # calculating the total order_price
            for order in items_tuple:

                order = tuple(order)

                # value of calculated order_price after substitution
                check_order_price = check_order_price +␣
↪final_item_prices[order[0]] * float(order[1])

            # check with the true order_price
            if check_order_price == row['order_price']:

                correct_item_found = True

                # get out of this loop - we found the correct item name
                break
            i = i + 1

        if correct_item_found:
            qty = map(int,qty)

            # modified shopping cart
            new_cart = list(zip(items,qty))
            print(str(new_cart))

            # replace with the modified cart
            df1.loc[index,'shopping_cart'] = str(new_cart)
            break
```

2.7.6 Check the Shopping cart after replacing the incorrect cart item

[51]:
```
df1['check_order_price'] = df1['shopping_cart'].apply(lambda x :␣
↪get_order_price(x) )
```

As we get no rows for the below query, the shopping cart has been fixed

[212]:
```
df1[df1['check_order_price'] != df1['order_price']]
```

2.7.7 Check the order_total column

As we already know the **order_price**, **delivery_charges** and the **coupon_discount**; we can calculate the order_total using the below equation:

order_total = (order_price*(1 - discount/100)) + delivery_price

Storing the correct **order_total** value in a new column

[215]:
```
df1['check_order_total'] = df1['order_price']*( 1 - df1['coupon_discount'] /␣
↪100 )  + df1['delivery_charges']
```

2.7.8 Rows with WRONG value of order_total

[217]:
```
df_incorrect_order_total = df1[ df1['check_order_total'] != df1['order_total'] ]
df_incorrect_order_total.shape
```

There are 32 orders that have incorrect value of order_total

2.7.9 Impute the order_total

Replace the order_total with the correct values

[55]:
```
for x in df_incorrect_order_total.index:
    df1.loc[x,'order_total'] = df_incorrect_order_total.
↪loc[x,'check_order_total']
```

2.7.10 Check after impute order_total

[218]:
```
df1[ df1['check_order_total'] != df1['order_total'] ]
```

As we get no result for the above query, we can be sure that the order_total column is fixed

2.8 Check for Latitude and Longitude

Looking at the below scatter plot of Latitude and Longitude, we can see that the values are mirror image of each other. This suggests that the values of Latitude and Longitude have been interchaged by mistake.

[219]:
```
plt.xlabel('Customer latitude')
plt.ylabel('Customer longitude')
plt.plot(df1.customer_lat,  df1.customer_long, '.r')
```

[220]:
```
df1.customer_lat.describe()
```

2.8.1 Looks like the Latitude and Langitude have been swapped by mistake

[59]:
```python
df_incorrect_lat_long = df1[df1.customer_lat > -37 ]
```

2.8.2 Swap the Latitude and Longitude

Correcting the Latitude and Longitude columns.

[60]:
```python
df_incorrect_lat_long.rename(columns = {'customer_lat': 'customer_long',␣
↪'customer_long' : 'customer_lat'}, inplace = True)

# change the original dataframe
for x in df_incorrect_lat_long.index:
    df1.loc[x] = df_incorrect_lat_long.loc[x]
```

```
C:\Users\prash\Anaconda3\lib\site-packages\pandas\core\frame.py:4223:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  return super().rename(**kwargs)
```

2.8.3 Check after replace

[221]:
```python
df1[df1.customer_lat > -37 ]
```

2.9 Check the is_happy_customer column

Using the SentimentIntensityAnalyzer class from nltk to do the sentiment analysis of the customer review

[62]:
```python
senti = SentimentIntensityAnalyzer()
```

2.9.1 Generate a column for latest_customer_review

If the compound polarity score is more than 0.05 we say that the customer was happy with the purchase

[63]:
```python
def check_latest_customer_review(sentence):
    '''
    Function to check the sentiment of customer based on the product review
    '''

    if senti.polarity_scores(sentence)['compound'] >= 0.05:
        return True
    else:
        return False
```

[64]:
```
df1['check_is_customer_happy'] = df1['latest_customer_review'].
↪apply(check_latest_customer_review)
```

We perform a **XOR** operation to check if the **is_happy_customer** has correct value

XOR give the rows where the calculated customer sentiment is different from the given customer sentiment

These rows have inncorrect value for **is_happy_customer**

[65]:
```
df_incorrect_is_happy_customer = df1[df1.is_happy_customer ^ df1.
↪check_is_customer_happy]
```

[222]:
```
df_incorrect_is_happy_customer.shape
```

28 orders that have incorrect value for **is_happy_customer**

[67]:
```
df_incorrect_is_happy_customer['is_happy_customer'] =␣
↪df_incorrect_is_happy_customer['check_is_customer_happy']
```

```
C:\Users\prash\Anaconda3\lib\site-packages\ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

2.9.2 Impute is_happy_customer

Changing the value for these incorrect orders

[68]:
```
for x in df_incorrect_is_happy_customer.index:
    df1.loc[x] = df_incorrect_is_happy_customer.loc[x]
```

2.9.3 Check is_happy_customer after imputation

[223]:
```
df1[df1.is_happy_customer ^ df1.check_is_customer_happy]
```

As we get no result for the above query, we can be sure that the imputation for is_happy_customer is correct

2.10 Lets find the nearest_warehouse

Using the Havershine formula we can calculate the distance between two location provided we have the latitude and longitude of the 2 locations and the Radius of Earth.

```
[70]: '''
Calculate distance using the Haversine Formula.

Reference:
https://community.esri.com/groups/coordinate-reference-systems/blog/2017/10/05/
 ↪haversine-formula
'''

def haversine(coord1: object, coord2: object):

    # Coordinates in decimal degrees (e.g. 2.89078, 12.79797)
    lon1, lat1 = coord1
    lon2, lat2 = coord2

    R = 6378000  # radius of Earth in meters
    phi_1 = math.radians(lat1)
    phi_2 = math.radians(lat2)

    delta_phi = math.radians(lat2 - lat1)
    delta_lambda = math.radians(lon2 - lon1)

    a = math.sin(delta_phi / 2.0) ** 2 + math.cos(phi_1) * math.cos(phi_2) *␣
 ↪math.sin(delta_lambda / 2.0) ** 2

    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    meters = R * c  # output distance in meters
    km = meters / 1000.0  # output distance in kilometers

    meters = round(meters, 3)
    km = round(km, 4)

    return km
```

Making a new Dataframe of Latitude and Longitude.

```
[71]: # prepare the data in Lat,Long form
cust_lat_long = df1[['customer_lat', 'customer_long']].apply(lambda x: ','.
 ↪join(x.astype(str)), axis = 1)

# store the prepared data in a new dataframe
df_cust_lat_long = pd.DataFrame(index = range(len(cust_lat_long)), data =␣
 ↪cust_lat_long, columns = ['lat_long'])
```

Store the Location of each Warehouse in Dictionary, with key as the warehouse name and value as the co-ordinates.

[72]:
```
dict_warehouses = {'Nickolson': (-37.818595,144.969551),
                   'Thompson' : (-37.8126732,144.9470689),
                    'Bakers' : (-37.8099961,144.995232)}
```

2.10.1 Distances from all 3 warehouese

Here we calculate the Distance of the Customer from all the 3 warehouse.

[73]:
```
def cal_warehouses_dist(lat_long):
    '''
    Function to calculate the distance of the customer from all the 3␣
↪warehouses.
    Return : A dictionary for value of distance of customer from each of the 3␣
↪warehouses
    '''

    # customer location
    lat_long = str(lat_long)
    re_results = re.findall(r'(-?\d+\.-?\d+)',lat_long)

    cust_lat = float(re_results[0])
    cust_long = float(re_results[1])

    dict_distance_from_warehouses = {}

    # calculate the distance from each of the 3 warehouses
    for warehouse,location in dict_warehouses.items():

        # haversine() function needs 2 list containing Lat, Long
        distance_from_warehouse = haversine([location[0],location[1]],␣
↪[cust_lat,cust_long])

        # store results in a dict
        dict_distance_from_warehouses[warehouse] = distance_from_warehouse

    return dict_distance_from_warehouses
```

2.10.2 Nearest warehouse data

As we already calculated the distance of the customer from all the 3 warehouses, we can get the name and distance of the warehouse that is nearest to the customer.

[74]:
```
def get_nearest_warehouse(dict_distance_from_warehouses):
    '''
    Using the result from cal_warehouses_dist() fucntion, return dictionary␣
↪having the warehouse closest to the customer
    '''
```

```python
    # get the min distance
    min_dist = min(dict_distance_from_warehouses.values())

    # get the name of warehouse corresponding to min distance
    nearest_warehouse = [ warehouse for warehouse, distance in␣
↪dict_distance_from_warehouses.items() if distance == min_dist ]

    dict_nearest_warehouse = {}

    # dict of nearest warehouse name and distance
    dict_nearest_warehouse[nearest_warehouse[0]] = min_dist

    return dict_nearest_warehouse
```

2.10.3 Nearest warehouse name

Getting the name of nearest warehouse.

[75]:
```python
def get_nearest_warehouse_name(dict_nearest_warehouse):
    '''
    Using the result from get_nearest_warehouse() fucntion, return the name of␣
↪warehouse closest to the customer
    '''
    return list(dict_nearest_warehouse.keys())[0]
```

2.10.4 Nearest warehoues distance

Getting the distance of the nearest warehouse.

[76]:
```python
def get_nearest_warehouse_dist(dict_nearest_warehouse):
    '''
    Using the result from get_nearest_warehouse() fucntion, return the distance␣
↪of warehouse closest to the customer
    '''
    return list(dict_nearest_warehouse.values())[0]
```

2.10.5 Storing the calculated nearest_warehouse and nearest_warehouse_dist in new column.

Applying the defined function for getting the nearest warehouse on our dataframe.

[77]:
```python
df1['check_nearest_warehouse'] = df_cust_lat_long['lat_long'].apply(lambda x :␣
↪get_nearest_warehouse_name(get_nearest_warehouse(cal_warehouses_dist(x))))
```

Applying the defined function for getting the nearest warehouse distance on our dataframe.

[78]:
```
df1['check_nearest_warehouse_dist'] = round(df_cust_lat_long['lat_long'].
↪apply(lambda x :␣
↪get_nearest_warehouse_dist(get_nearest_warehouse(cal_warehouses_dist(x)))␣
↪),4)
```

Below 55 orders have incorrect value of **nearest_warehouse**. The variation in the distance of the nearest warehouse is due to the rounding of the distance. As the variation is very minor of a single decimal place, we can ignore this and impute the **nearest_warehouse** column.

[224]:
```
df1[df1['nearest_warehouse'] != df1['check_nearest_warehouse']].loc[:
↪,['nearest_warehouse','distance_to_nearest_warehouse','check_nearest_warehouse','check_near
↪shape
```

2.10.6 Impute nearest_warehouse

Imputing the above identified 55 order having incorrect value of nearest_warehouse.

[80]:
```
df_incorrect_nearest_warehouse = df1[df1['nearest_warehouse'] !=␣
↪df1['check_nearest_warehouse']]
```

[81]:
```
for x in df_incorrect_nearest_warehouse.index:
    df1.loc[x,'nearest_warehouse'] = df1.loc[x,'check_nearest_warehouse']
```

2.10.7 Check after replace

As we have imuted the nearest warehouse value, we don not get any result for the below query.

[225]:
```
df1[df1['nearest_warehouse'] != df1['check_nearest_warehouse']]
```

2.11 Check the value of distance_to_nearest_warehouse

We calculate the absolute difference between the distance given in the dataseta and the calculated distance to the nearest warehouse.

We can ignore the minor variation arising due to rounding of the distances

Following are the orders for which the values of **distance_to_nearest_warehouse** is incorrect

[226]:
```
df1[abs((round(df1['distance_to_nearest_warehouse'],3) -␣
↪round(df1['check_nearest_warehouse_dist'],3) )) > 0.2 ].shape
```

2.11.1 Replacing the distance_to_nearest_warehouse with correct values

Replacing the distance for orders having incorrect value of distance_to_nearest_warehouse.

[84]:
```
df_incorrect_dist_nearest_warehouse =␣
↪df1[abs((round(df1['distance_to_nearest_warehouse'],3) -␣
↪round(df1['check_nearest_warehouse_dist'],3) )) > 0.2 ]
```

[85]:
```
for x in df_incorrect_dist_nearest_warehouse.index:
    df1.loc[x,'distance_to_nearest_warehouse'] =␣
 ↪df_incorrect_dist_nearest_warehouse.loc[x,'check_nearest_warehouse_dist']
```

Check the values after imputation

[227]:
```
df1[abs((round(df1['distance_to_nearest_warehouse'],3) -␣
 ↪round(df1['check_nearest_warehouse_dist'],3) )) > 0.2 ].loc[:
 ↪,['nearest_warehouse','distance_to_nearest_warehouse','check_nearest_warehouse','check_near
```

2.12 Check the season column

As we know the months for which each of the season happens, we can do as simple mapping and check the season column.

As the date has already been fixed we can be sure that the season is incorrect for the orders failing this check.

[87]:
```
def get_season(order_date):
    '''
    Function to check if the date is mapped to the correct season of the year.
    '''

    if int(order_date.split('-')[1]) in [12,1,2]:
        return 'Summer'
    elif int(order_date.split('-')[1]) in [3,4,5]:
        return 'Autumn'
    elif int(order_date.split('-')[1]) in [6,7,8]:
        return 'Winter'
    elif int(order_date.split('-')[1]) in [9,10,11]:
        return 'Spring'
```

[88]:
```
df1['check_season'] = df1['date'].apply(get_season)
```

Following rows have incorrect value of season

[228]:
```
df1[df1['season'] != df1['check_season']].shape
```

2.12.1 Impute the season

[90]:
```
df_incorrect_season = df1[df1['season'] != df1['check_season']]
```

[91]:
```
# change the original dataframe
for x in df_incorrect_season.index:
    df1.loc[x,'season'] = df_incorrect_season.loc[x,'check_season']
```

2.12.2 Check after impute

[229]:
```
df1[df1['season'] != df1['check_season']]
```

2.13 Checking the values of is_expedited_delivery

By visualizing Boxplot of Delivery charges when is_customer is True/False, can tell us whether the value of delivery charges is an outlier or not.

As we have already fixed the is_happy_customer, the only value to impute here is the is_expedite_delivery.

[230]:
```
df1.
↪boxplot('delivery_charges',by=['is_expedited_delivery','is_happy_customer'],sym='k.
↪', figsize=(20, 10))
```

Above Boxplot shows that the is_expedite_delivery is incorrect for 8 orders.

When is_expedited_delivery is False and is_happy_customer is True

[94]:
```
df_f_t = df1[(df1['is_expedited_delivery'] == False) &␣
↪(df1['is_happy_customer'] == True)]

q1 = np.quantile(df1[ (df1['is_expedited_delivery'] == False) &␣
↪(df1['is_happy_customer'] == True) ]['delivery_charges'], .25)
q3 = np.quantile(df1[ (df1['is_expedited_delivery'] == False) &␣
↪(df1['is_happy_customer'] == True) ]['delivery_charges'], .75)

iqr = q3 - q1

list_index_to_impute = list(df_f_t[df_f_t['delivery_charges'] > (q3 + 1.5*␣
↪iqr)].index)
```

When is_expedited_delivery is False and is_happy_customer is False

[95]:
```
df_f_f = df1[(df1['is_expedited_delivery'] == False) &␣
↪(df1['is_happy_customer'] == False)]

q1 = np.quantile(df1[ (df1['is_expedited_delivery'] == False) &␣
↪(df1['is_happy_customer'] == False) ]['delivery_charges'], .25)
q3 = np.quantile(df1[ (df1['is_expedited_delivery'] == False) &␣
↪(df1['is_happy_customer'] == False) ]['delivery_charges'], .75)

iqr = q3 - q1

list_index_to_impute.append(list(df_f_f[df_f_f['delivery_charges'] > (q3 + 1.5*␣
↪iqr)].index))
```

Making a single list of index to impute

```
[96]: lst_indexes = []

for x in list_index_to_impute:

    if type(x) != list:
        lst_indexes.append(x)
    elif type(x)== list:
        for y in x:
            lst_indexes.append(y)
```

2.13.1 Impute is_expedite_delivery

We have the indexes of the order to impute the is_expedite_delivery. Using these indexes to change the original dataframe.

```
[231]: lst_indexes
```

```
[98]: # changing the values in the original datafarme
for index in lst_indexes:
    df1.loc[index,'is_expedited_delivery'] = True
```

2.13.2 Checking the Boxplot after impute

We can see that we no longer have outliers for the is_expedited_delivery.

```
[232]: df1.
 ↪boxplot('delivery_charges',by=['is_expedited_delivery','is_happy_customer'],sym='k.
 ↪', figsize=(20, 10))
```

2.13.3 Export the CSV for Task 1 - Dirty data

Before we export the data, we get rid of the unnecessary columns added for analysis.

```
[100]: columns_to_drop =␣
 ↪['shopping_items','shopping_item_qty','check_order_price','check_order_total','check_is_cus
df1.drop(columns_to_drop, 1, inplace=True)
```

```
[101]: df1.to_csv('31187366_dirty_data_solution.csv', index=False)
```

3 TASK 2 - MISSING DATA

Reading the data file.

```
[102]: #load the data
df2 = pd.read_csv('31187366_missing_data.csv')
```

Using the `describe` function to view some basic statistical details like percentile, mean, std etc. of the read CSV file.

[233]:
```
df2.describe()
```

As `decribe()` by default shows stastics for only numerical data.

To see the statistics of Categorical data we use `include` parameter of `describe` to include Datatype of type Object i.e. categorical data.

[234]:
```
df2.describe(include=['O'])
```

3.1 Missing is_happy_customer

Below 40 orders have missing value of `is_happy_customer`

[235]:
```
df2_missing_is_happy_customer = df2[df2['is_happy_customer'].isna()]
df2_missing_is_happy_customer.shape
```

3.1.1 Fill the missing is_happy_customer

We make use of `SentimentIntensityAnalyzer` to decide the sentiment of the customer based on the customer review and fill the value of is_happy_customer column.

[106]:
```
senti = SentimentIntensityAnalyzer()
```

[107]:
```
def fill_is_happy_customer(review):
    '''
    Function to check the sentiment of customer using the customer review.
    '''

    if senti.polarity_scores(review)['compound'] >= 0.05:
        return 1.0
    else:
        return 0.0
```

[108]:
```
df2_missing_is_happy_customer['is_happy_customer'] =␣
↪df2_missing_is_happy_customer['latest_customer_review'].
↪apply(fill_is_happy_customer)
```

```
C:\Users\prash\Anaconda3\lib\site-packages\ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

3.1.2 Check after filling the missing data

[236]:
```
df2_missing_is_happy_customer[df2_missing_is_happy_customer['is_happy_customer'].
↪isna()]
```

3.1.3 Fill the original Dataframe

Making the changes in the original dataframe.

[110]:
```
for x in df2_missing_is_happy_customer.index:
    df2.loc[x,'is_happy_customer'] = df2_missing_is_happy_customer.
↪loc[x,'is_happy_customer']
```

We dont get any result for below query, as we have filled the is_happy_customer column.

[237]:
```
df2[df2['is_happy_customer'].isna()]
```

3.2 Checking Distance and Warehouse

3.2.1 Both distance and warehouse missing

Below 31 orders have missing value for `nearest_warehouse` and `distance_to_nearest_warehouse` column.

[238]:
```
df2[df2['nearest_warehouse'].isna() & df2['distance_to_nearest_warehouse'].
↪isna() ].shape
```

3.2.2 Warehouse name missing

Below 55 orders have missing value of `nearest_warehouse` column.

[239]:
```
df2[df2['nearest_warehouse'].isna() ].shape
```

3.2.3 Distance missing

Below 31 orders have missing value of `distance_to_nearest_warehouse` column.

[240]:
```
df2[df2['distance_to_nearest_warehouse'].isna()].shape
```

3.2.4 Find the Distance and Warehouse

3.2.5 Lets find the nearest_warehouse

Make a new Dataframe of missing warehouses.

[115]:
```
df2_missing_warehouse = df2[df2['nearest_warehouse'].isna()]
```

Prepare data to calcuate the distance and the nearest warehouse.

[116]:
```
cust_lat_long = df2_missing_warehouse['customer_lat'].astype(str) + ',' +␣
↪df2_missing_warehouse['customer_long'].astype(str)
df_cust_lat_long = pd.DataFrame(index = range(len(cust_lat_long)), data =␣
↪list(cust_lat_long), columns = ['lat_long'])
```

Applying the functions on the Dataframe.

[117]:
```
nearest_warehouse = df_cust_lat_long['lat_long'].apply(lambda x :␣
↪get_nearest_warehouse_name(get_nearest_warehouse(cal_warehouses_dist(x))))
nearst_warehouse_dist = round(df_cust_lat_long['lat_long'].apply(lambda x :␣
↪get_nearest_warehouse_dist(get_nearest_warehouse(cal_warehouses_dist(x)))␣
↪),4)
```

[118]:
```
filling_missing_warehouses = tuple(zip(nearest_warehouse,nearst_warehouse_dist))
```

3.2.6 Replace the missing warehouse

Imputing the original dataframe

[119]:
```
i = 0

lst_missing_warehouses_index = list(df2_missing_warehouse.index)

for x in filling_missing_warehouses:

    df2.loc[lst_missing_warehouses_index[i],'nearest_warehouse'] = str(x[0])
    df2.loc[lst_missing_warehouses_index[i],'distance_to_nearest_warehouse'] =␣
↪round(x[1],4)

    i += 1
```

As we have imputed the distance and warehouse name, we get no result for below queries.

[241]:
```
df2[df2['distance_to_nearest_warehouse'].isna()]
```

[242]:
```
df2[df2['nearest_warehouse'].isna()]
```

3.3 Fill the order_price

Seperating the item name and quantity using the predefined functions, namely `get_item_names` and `get_item_qty`.

[122]:
```
df2['shopping_items'] = df2['shopping_cart'].apply(get_item_names)
df2['shopping_item_quantity'] = df2['shopping_cart'].apply(get_item_qty)
```

Make a new dataframe of missing order_price

[123]:
```
df2_missing_order_price = df2[df2['order_price'].isna()]
```

Using the predefined `get_order_price` to calculate the order_price.

[124]:
```
df2_missing_order_price['order_price'] =␣
↪df2_missing_order_price['shopping_cart'].apply(lambda x : get_order_price(x)␣
↪)
```

```
C:\Users\prash\Anaconda3\lib\site-packages\ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

3.3.1 Impute order_price

Imputing the order_price in the original dataframe.

[125]:
```
for x in df2_missing_order_price.index:
    df2.loc[x,'order_price'] = df2_missing_order_price.loc[x,'order_price']
```

3.3.2 Check after replace

As we have imputed the value of order_price, we get no result for below query.

[243]:
```
df2[df2['order_price'].isna()]
```

3.4 Check order_total

Make a new dataframe for dataframe of missing order_total

[127]:
```
df2_missing_order_total = df2[df2['order_total'].isna()]
```

As we know the discount, delivery_charges and the order_price; we can calculate the order_total.

[128]:
```
def get_order_total(df):
    '''
    Function to calculate the order_total
    '''

    order_price = df['order_price']
    discount = df['coupon_discount']
    delivery_charges = df['delivery_charges']

    return order_price*( 1 - (discount/100)) + delivery_charges
```

[129]:
```
df2_missing_order_total['order_total'] = df2_missing_order_total.
↪apply(get_order_total,axis=1)
```

```
C:\Users\prash\Anaconda3\lib\site-packages\ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """Entry point for launching an IPython kernel.
```

3.4.1 Fill missing order_total

Imputing the original dataframe.

[130]:
```
for x in df2_missing_order_total.index:
    df2.loc[x,'order_total'] = df2_missing_order_total.loc[x,'order_total']
```

3.4.2 Check after replace

As we have imputed the order_total, we get no data for below query

[244]:
```
df2[df2['order_total'].isna()]
```

3.5 Delivery charges

3.5.1 METHOD 1. USING LINEAR REGRESSION MODEL

3.5.2 Making Regression model for each of the three seasons

As the data given in Missing Data file is correct, so we can use this data to build Linear Regression model.

3.5.3 1. Regression model for Spring

[132]:
```
df_spring = df2[df2['season'] == 'Spring'].dropna(how="any")
```

Preparing the input and output data to train the Linear regression model

[133]:
```
lst_tuple = list(zip(df_spring['distance_to_nearest_warehouse'],␣
 ↪df_spring['is_expedited_delivery'],df_spring['is_happy_customer']))
lst_input = [ list(x) for x in lst_tuple ]
lst_output = list(df_spring['delivery_charges'].astype(int))
```

Fitting a model for delivery charges for Spring

[134]:
```
model_spring = LinearRegression().fit(lst_input, lst_output)
```

Preparing data of Spring to predict the delivery charges

[135]:

```
df2_spring = df2[(df2['distance_to_nearest_warehouse'].notnull()) &␣
↪(df2['season'] == 'Spring') & (df2['is_expedited_delivery'].notnull()) &␣
↪(df2['is_happy_customer'].notnull())]

lst_tuple_input_to_check =␣
↪list(zip(df2_spring['distance_to_nearest_warehouse'],␣
↪df2_spring['is_expedited_delivery'], df2_spring['is_happy_customer']))
lst_input_to_check = [ list(x) for x in lst_tuple_input_to_check ]
```

Predicted Delivery charges using Regression model for Spring

[136]:
```
delivery_charges_using_model_spring = model_spring.predict(lst_input_to_check)
```

3.5.4 Compare given Delivery charges with predicted Delivery charges for Spring

[245]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression model')
plt.scatter(df2_spring['delivery_charges'], delivery_charges_using_model_spring)
```

3.5.5 2. Regression model for Autumn

[138]:
```
df_autumn = df2[df2['season'] == 'Autumn'].dropna(how="any")
```

Preparing the input and output data to train the Linear regression model for Autumn

[139]:
```
lst_tuple = list(zip(df_autumn['distance_to_nearest_warehouse'],␣
↪df_autumn['is_expedited_delivery'],df_autumn['is_happy_customer']))
lst_input = [ list(x) for x in lst_tuple ]
lst_output = list(df_autumn['delivery_charges'].astype(int))
```

Fitting a model for delivery charges for Autumn

[140]:
```
model_autumn = LinearRegression().fit(lst_input, lst_output)
```

Preparing data of Autumn to predict the delivery charges

[141]:
```
df2_autumn = df2[(df2['distance_to_nearest_warehouse'].notnull()) &␣
↪(df2['season'] == 'Autumn') & (df2['is_expedited_delivery'].notnull()) &␣
↪(df2['is_happy_customer'].notnull())]

lst_tuple_input_to_check =␣
↪list(zip(df2_autumn['distance_to_nearest_warehouse'],␣
↪df2_autumn['is_expedited_delivery'], df2_autumn['is_happy_customer']))
lst_input_to_check = [ list(x) for x in lst_tuple_input_to_check ]
```

Predicted Delivery charges using Regression model for Autumn

[142]:
```
delivery_charges_using_model_autumn = model_autumn.predict(lst_input_to_check)
```

3.5.6 Compare given Delivery charges with predicted Delivery charges for Autumn

[246]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression model')
plt.scatter(df2_autumn['delivery_charges'], delivery_charges_using_model_autumn)
```

3.5.7 3. Regression model for Winter

[144]:
```
df_winter = df2[df2['season'] == 'Winter'].dropna(how="any")
```

Preparing the input and output data to train the Linear regression model for Winter

[145]:
```
lst_tuple = list(zip(df_winter['distance_to_nearest_warehouse'],␣
 ↪df_winter['is_expedited_delivery'],df_winter['is_happy_customer']))
lst_input = [ list(x) for x in lst_tuple ]
lst_output = list(df_winter['delivery_charges'].astype(int))
```

Fitting a model for delivery charges for Winter

[146]:
```
model_winter = LinearRegression().fit(lst_input, lst_output)
```

Preparing data of Winter to predict the delivery charges

[147]:
```
df2_winter = df2[(df2['distance_to_nearest_warehouse'].notnull()) &␣
 ↪(df2['season'] == 'Winter') & (df2['is_expedited_delivery'].notnull()) &␣
 ↪(df2['is_happy_customer'].notnull())]

lst_tuple_input_to_check = list(zip(df_winter['distance_to_nearest_warehouse'],␣
 ↪df_winter['is_expedited_delivery'], df_winter['is_happy_customer']))
lst_input_to_check = [ list(x) for x in lst_tuple_input_to_check ]
```

Predicted Delivery charges using Regression model for Winter

[148]:
```
delivery_charges_using_model_winter = model_winter.predict(lst_input_to_check)
```

3.5.8 Compare given Delivery charges with predicted Delivery charges for Winter

[247]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression model')
plt.scatter(df_winter['delivery_charges'], delivery_charges_using_model_winter)
```

3.5.9 4. Regression model for Summer

[150]:
```
df_summer = df2[df2['season'] == 'Summer'].dropna(how="any")
```

Preparing the input and output data to train the Linear regression model for Summer

[151]:
```
lst_tuple = list(zip(df_summer['distance_to_nearest_warehouse'],␣
↪df_summer['is_expedited_delivery'],df_summer['is_happy_customer']))
lst_input = [ list(x) for x in lst_tuple ]
lst_output = list(df_summer['delivery_charges'].astype(int))
```

[152]:
```
model_summer = LinearRegression().fit(lst_input, lst_output)
```

Preparing data of Autumn to predict the delivery charges for Summer

[153]:
```
df2_summer = df2[(df2['distance_to_nearest_warehouse'].notnull()) &␣
↪(df2['season'] == 'Summer') & (df2['is_expedited_delivery'].notnull()) &␣
↪(df2['is_happy_customer'].notnull())]

lst_tuple_input_to_check =␣
↪list(zip(df2_summer['distance_to_nearest_warehouse'],␣
↪df2_summer['is_expedited_delivery'], df2_summer['is_happy_customer']))
lst_input_to_check = [ list(x) for x in lst_tuple_input_to_check ]
```

Predicted Delivery charges using Regression model for Summer

[154]:
```
delivery_charges_using_model_summer = model_summer.predict(lst_input_to_check)
```

3.5.10 Compare given Delivery charges with predicted Delivery charges for Summer

[248]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression model')
plt.scatter(df2_summer['delivery_charges'], delivery_charges_using_model_summer)
```

3.5.11 METHOD 2. Calculate Delivery charges using EQUATION

As we know that order_total is the sum of discounted order_price and the delivery charges, we can calculate the order_total as follows:

order_total = delivery_price + ((1 - discount/100) * order_price)

Rewriting the equation, we get the delivery_charges for each order.

delivery_price = order_total - ((1 - discount/100) * order_price)

[249]:
```
df2[df2['delivery_charges'].notnull()].shape
```

Verify the equation - works for all the 460 rows for which we have the delivery charges given.

[250]:
```
df2[ df2['delivery_charges'] == round((df2['order_total'] - ( ( 1 -␣
↪df2['coupon_discount']/100) * df2['order_price'] )),2) ].shape
```

Calculating Delivery charges for each of the seasons using the equation.

[158]:
```
delivery_charges_using_eq_autumn = list(round((df2_autumn['order_total'] - ( (␣
↪1 - df2_autumn['coupon_discount']/100) * df2_autumn['order_price'] )),2))
delivery_charges_using_eq_spring = list(round((df2_spring['order_total'] - ( (␣
↪1 - df2_spring['coupon_discount']/100) * df2_spring['order_price'] )),2))
delivery_charges_using_eq_winter = list(round((df_winter['order_total'] - ( ( 1␣
↪- df_winter['coupon_discount']/100) * df_winter['order_price'] )),2))
delivery_charges_using_eq_summer = list(round((df2_summer['order_total'] - ( (␣
↪1 - df2_summer['coupon_discount']/100) * df2_summer['order_price'] )),2))
```

3.5.12 Scatter plot of Delivery charges obtained using the equation for Autumn

[251]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Equation')
plt.scatter(delivery_charges_using_eq_autumn, df2_autumn['delivery_charges'])
```

3.5.13 Scatter plot of Delivery charges obtained using the equation for Spring

[252]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Equation')
plt.scatter(delivery_charges_using_eq_spring, df2_spring['delivery_charges'])
```

3.5.14 Scatter plot of Delivery charges obtained using the equation for Winter

[253]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Equation')
plt.scatter(delivery_charges_using_eq_winter, df_winter['delivery_charges'])
```

3.5.15 Scatter plot of Delivery charges obtained using the equation for Summer

[254]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Equation')
plt.scatter(delivery_charges_using_eq_summer, df2_summer['delivery_charges'])
```

3.5.16 Comapring the 2 approach: Equation v/s Regression model

3.5.17 Fill the delivery_charges - using equation

We can see from the above plots that the deilvery charges obtained using the equation, do not deviate much from the true delivery charges.

On the other hand, the delivery charges obtained using the Regression model, has high variance from the true delivery charges.

This means using the Equation to fill the missing Delivery charges is a good approach.

Also, the Equation is independent of the season and works in all the cases.

[163]:
```
df_missing_deliver_charges = df2[( df2['delivery_charges'].isna() )]
```

[255]:
```
df_missing_deliver_charges.shape
```

[165]:
```
def get_delivery_charge(order):
    '''
    Function to calculate the delivery charges
    '''

    order_total = order['order_total']
    order_price = order['order_price']
    coupon_discount = order['coupon_discount']

    delivery_charges = order_total - (order_price * (1 - coupon_discount/100))

    return round(delivery_charges,2)
```

Impute the original dataframe

[166]:
```
for index,row in df_missing_deliver_charges.iterrows():
    df2.loc[index,'delivery_charges'] = get_delivery_charge(row)
```

3.5.18 Check after filling the missing delivery_charges

As we have filled the missing delivery_charges, we get no result for below query.

[256]:
```
df2[df2['delivery_charges'].isna()]
```

3.5.19 Export the CSV for Task 2 - Missing data

Before we export the data, we get rid of the unnecessary columns added for analysis.

[168]:
```
columns_to_drop = ['shopping_items','shopping_item_quantity']
df2.drop(columns_to_drop, 1, inplace=True)
```

[169]:
```
df2.to_csv('31187366_missing_data_solution.csv', index=False)
```

4 TASK 3 - OUTLIER ANALYSIS

Read the data for outlier detection

[170]:
```
df3 = pd.read_csv('31187366_outlier_data.csv')
```

As the deliver charges follow a linear model;

We have to do seperate Box plot analysis for each season, is_happy_customer, is_expedited_delivery

[257]:
```
df3.
↪boxplot('delivery_charges',by=['season','is_expedited_delivery','is_happy_customer'],sym='k
↪', figsize=(30, 20))
```

To see whether the delivery charges are indeed Outliers, I have compared them to the predicted delivery charges for that season using the Regression model.

If the difference between the predicted delivery charges from the regression model and the given delivery charge is greater than 4, we declare them as outliers.

[172]:
```
# list to store the index of final outliers
index_outliers = []

# list to store the delivery charges of the outliers
lst_delivery_charges = []

# dictionary to store the outlier for each season
dict_predicted_delivery_charges = {}
dict_predicted_delivery_charges['Spring'] = list()
dict_predicted_delivery_charges['Autumn'] = list()
dict_predicted_delivery_charges['Winter'] = list()
dict_predicted_delivery_charges['Summer'] = list()


# repeat for each season
for season in ['Autumn', 'Summer', 'Spring', 'Winter']:
    for is_expedite in [True, False]:
        for is_happy in [True, False]:

            # filter the datframe before we proceed
            df_season = df3[ (df3['season'] == season) &␣
↪(df3['is_expedited_delivery'] == is_expedite) & (df3['is_happy_customer'] ==␣
↪is_happy)]

            # get the first and third quartile
            q1 = np.quantile(df_season['delivery_charges'], .25)
            q3 = np.quantile(df_season['delivery_charges'], .75)

            # cal iqr
            iqr = q3-q1

            # add index to the list of outliers
            index_outliers = index_outliers + ␣
↪list(df_season[df_season['delivery_charges'] > (q3 + 1.5*iqr)].index)
```

```
            index_outliers = index_outliers + ␣
↪list(df_season[df_season['delivery_charges'] < (q1 - 1.5*iqr)].index)

            # make a dataframe of outliers
            df_outliers1 = df3.
↪iloc[list(df_season[df_season['delivery_charges'] > (q3 + 1.5*iqr)].index)]
            df_outliers2 = df3.
↪iloc[list(df_season[df_season['delivery_charges'] < (q1 - 1.5*iqr)].index)]

            # merge the 2 df
            df_outliers = pd.concat([df_outliers1, df_outliers2],␣
↪ignore_index=True)

            # prepare the data to be used for Regression model
            lst_tuple_input_to_check =␣
↪list(zip(df_outliers['distance_to_nearest_warehouse'],␣
↪df_outliers['is_expedited_delivery'], df_season['is_happy_customer']))
            lst_input_to_check = [ list(x) for x in lst_tuple_input_to_check ]

            # predict the delivery_charges of outliers
            if len(lst_input_to_check) != 0:
                if season == 'Summer':
                    dict_predicted_delivery_charges[season].append(model_summer.
↪predict(lst_input_to_check))
                    predicted_charges = model_summer.predict(lst_input_to_check)

                elif season == 'Winter':
                    dict_predicted_delivery_charges[season].append(model_winter.
↪predict(lst_input_to_check))
                    predicted_charges = model_winter.predict(lst_input_to_check)

                elif season == 'Spring':
                    dict_predicted_delivery_charges[season].append(model_spring.
↪predict(lst_input_to_check))
                    predicted_charges = model_spring.predict(lst_input_to_check)

                elif season == 'Autumn':
                    dict_predicted_delivery_charges[season].append(model_autumn.
↪predict(lst_input_to_check))
                    predicted_charges = model_autumn.predict(lst_input_to_check)

                # add new column with the predicted delivery_charges
                df_outliers['predicted_delivery_charges'] =␣
↪list(predicted_charges)
```

```
                # if the difference is greater than 4, we declare the␣
↪delivery_charge as OUTLIER
                lst_delivery_charges.append(list(df_outliers[␣
↪abs(df_outliers['predicted_delivery_charges'] -␣
↪df_outliers['delivery_charges']) > 4].delivery_charges))
```

[173]:
```
lst_outlier_delivery_charges = []

# making a single list of ints as the data returend was list of list
for x in lst_delivery_charges:
    for y in x:
        lst_outlier_delivery_charges.append(y)
```

[174]:
```
dict_delivery_charges = {}
dict_delivery_charges['Spring'] = list()
dict_delivery_charges['Summer'] = []
dict_delivery_charges['Winter'] = []
dict_delivery_charges['Autumn'] = []

# gathering all the data for same season
# will be used for comparision
for season in ['Summer','Spring','Winter','Autumn']:
    for x in dict_predicted_delivery_charges[season]:
        for y in x:
            dict_delivery_charges[season].append(y)
```

4.1 Visualising the Predicted delivery charges and the given Delivery charges.

[175]:
```
df3_outliers = df3.iloc[index_outliers]
```

4.1.1 Outlier for Summer

[258]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression')
plt.scatter(df3_outliers[df3_outliers['season']␣
↪=='Summer']['delivery_charges'], dict_delivery_charges['Summer'])
```

4.1.2 Outlier for Winter

[259]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression')
plt.scatter(df3_outliers[df3_outliers['season']␣
↪=='Winter']['delivery_charges'], dict_delivery_charges['Winter'])
```

4.1.3 Outlier for Spring

[260]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression')
plt.scatter(df3_outliers[df3_outliers['season']␣
↪=='Spring']['delivery_charges'], dict_delivery_charges['Spring'])
```

4.1.4 Outlier for Autumn

[261]:
```
plt.xlabel('delivery_charge from CSV')
plt.ylabel('delivery_charge using Regression')
plt.scatter(df3_outliers[df3_outliers['season']␣
↪=='Autumn']['delivery_charges'], dict_delivery_charges['Autumn'])
```

4.2 Remove the Outilers

We can see from above plots that the value of delivery_charges predicted from the Linear Regression model and the given delivery charges vary a lot.

This justifies that these delivery charges are outliers.

[180]:
```
def is_outlier(delivery_charge):
    '''
    Function to check if the delievry charge belongs to a the outliers list of␣
↪delivery charges
    '''

    if delivery_charge in lst_outlier_delivery_charges:
        return True
    else:
        return False
```

[262]:
```
len(df3[df3['delivery_charges'].apply(is_outlier)])
```

There are eventually 31 outliers in the given file

Delete the orders having the outlier delivery_charges

[182]:
```
df3.drop(list(df3[df3['delivery_charges'].apply(is_outlier)].index),␣
↪inplace=True)
```

4.2.1 Export the CSV for Task 3 - Outliers

[183]:
```
df3.to_csv('31187366_outlier_data_solution.csv', index=False)
```

5 Summary

This assessment helps to build up the knowledge of Data cleansing to handle and remove various anamolies like Semantic anamoly, Coverage anamoly and Syntactic anamoly. The main objective achieved from this assessment are as follows:

- **Manipulating Python Data structures and Pandas DataFrames**: For successful completion of this task knowledge of manipulating and operating on Dataframes was fundamental. The main functions used for data frame manipulation were `apply`, `zip`, `filter`, `dropna`, `isna`, `notna`, `replace`, `unique` etc.
- **Developing Regression model**: Using the `LinearRegression` class, it was possible to make Linear regression model for Delivery charges for various seasons. This model was then used to predict and fill the missing delivery_charges.
- **Sentiment Analysis**: Using the `SentimentIntensityAnalyzer` class it was possible to predict of sentiment based on the last customer purchase. The compound polarity score was used to classify the happy and unhappy customers.
- **Visualizing Data**: For detecting outliers and verifying results making use of proper plots to visualize the data was important. The visualization was done using `matplotlib` library to plot Scatter plot and Histograms.
- **Formulating Regular Expressions** : Developing Regular expression to capture Item name and Quantity of item order for a Shopping cart, helped to build upon the knowledge of forming Regular expressions. Using `findall` method from `re` package helped to capture a list of data using the defined regular expression.
- **Using Date module**: To help validate the order date, the python `date` module was helpful.
- **Export CSV files** : Using the pandas `to_csv` function, it was possible to export the rectified Dataframe to CSV files.

6 References

Distance on a sphere: The Haversine Formula | GeoNet, The Esri Community | GIS and Geospatial Professional Community. (2020). Retrieved 27 September 2020, from https://community.esri.com/groups/coordinate-reference-systems/blog/2017/10/05/haversine-formula

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

nltk.sentiment.vader — NLTK 3.5 documentation. (2020). Retrieved 18 October 2020, from https://www.nltk.org/_modules/nltk/sentiment/vader.html

Pyplot tutorial — Matplotlib 2.0.2 documentation. (2020). Retrieved 18 October 2020, from https://matplotlib.org/users/pyplot_tutorial.html

7.1.6. What are outliers in the data?. (2020). Retrieved 18 October 2020, from https://www.itl.nist.gov/div898/handbook/prc/section1/prc16.htm