Order Brushing

June 14, 2020

1 Detecting abnormal user behaviour

1.1 Shopee Code League 2020: Order Brushing

People think of creative way to cheat the system.
For e-commerce sellers, one possible way is to create fake accounts to boost their rankings and reviews. We had previously solved this using network analysis.
Order brushing is another method sellers used. Some buyers employ third party to create fake purchases and post fake reviews, others simply use strangers' identifications. Since there might not be a clear relationship between sellers and buyers, relationship investigation between buyers and sellers become redundant.
This year, we will attempt to investigate fake orders without assuming relationship between buyer and seller and rely only on the transaction details.

The original competition link and dataset can be found in:
https://www.kaggle.com/c/students-order-brushing-1/overview

More information about order brushing:
https://blogs.wsj.com/chinarealtime/2015/03/03/they-call-it-brushing-the-dark-art-of-alibaba-sales-fakery/

1.2 What constitutes order brushing?

For Shopee, a concentrate rate (CR) metric is used to determine if there is an abnormal purchasing behaviour for the given hour. Concentrate rate measures the number of orders / number of unique buyers in an hour

Specifically, $CR = \frac{Orders_{1hr}}{UniqueBuyers_{1hr}}$

A concentrate rate >= 3 will be deem as order brushing

A buyer is considered as suspicious if the buyer buys the most products in all the fraud orders, called Order proportion (OP).

Specifically, $OP = \frac{Orders_{user1}}{TotalFraudOrders}$

A user with max order proportion among all users will be considered as fraud user

1.2.1 Let's explore the data

[1]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from datetime import timedelta
```

[2]:
```python
orders = pd.read_csv('order_brush_order.csv')
```

We observe that event_time is not correctly formatted as dates

[3]:
```python
orders['event_time'] = pd.to_datetime(orders['event_time'])
```

[4]:
```python
orders.info()
orders.head()
```

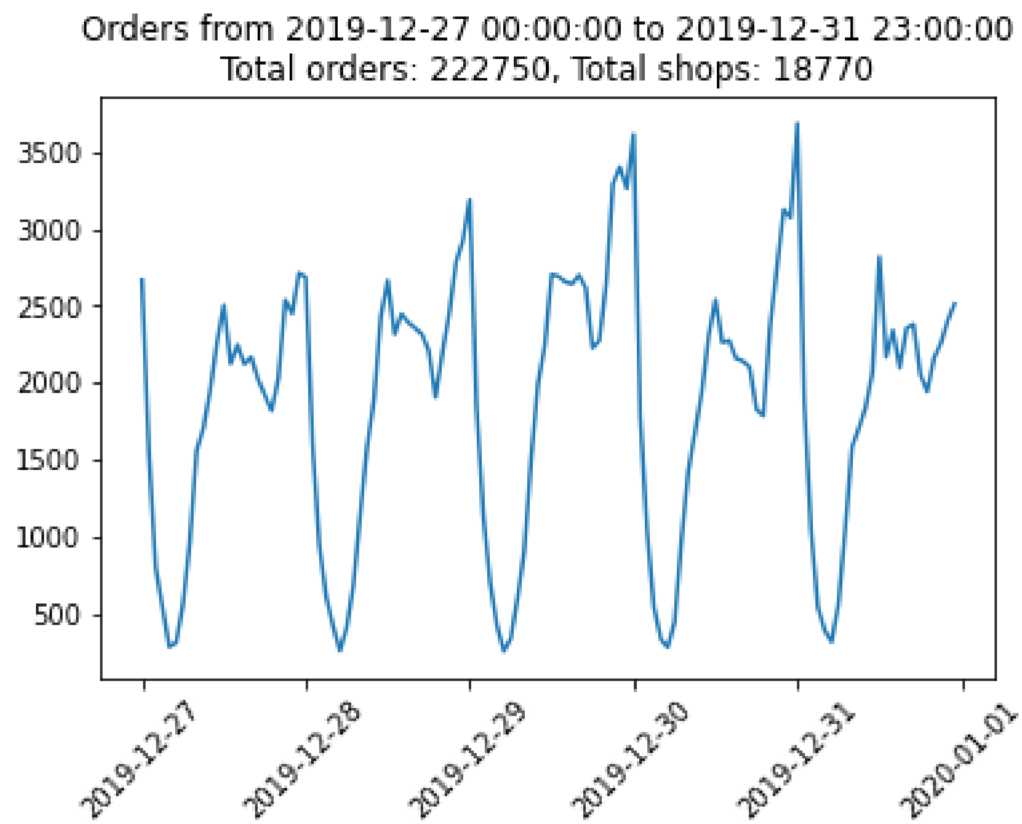
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 222750 entries, 0 to 222749
Data columns (total 4 columns):
 #   Column      Non-Null Count   Dtype
---  ------      --------------   -----
 0   orderid     222750 non-null  int64
 1   shopid      222750 non-null  int64
 2   userid      222750 non-null  int64
 3   event_time  222750 non-null  datetime64[ns]
dtypes: datetime64[ns](1), int64(3)
memory usage: 6.8 MB
```

[4]:

	orderid	shopid	userid	event_time
0	31076582227611	93950878	30530270	2019-12-27 00:23:03
1	31118059853484	156423439	46057927	2019-12-27 11:54:20
2	31123355095755	173699291	67341739	2019-12-27 13:22:35
3	31122059872723	63674025	149380322	2019-12-27 13:01:00
4	31117075665123	127249066	149493217	2019-12-27 11:37:55

The purchase records are taken during 2019 end of year and we should expect to a surge of orders to most shops.
Now we need to investigate which shops conduct order brushing during the festival period.

[5]:
```python
fig, ax = plt.subplots()
df = orders.groupby(['event_time'])['orderid'].count().resample('h').sum().
 ↪reset_index()
ax.plot('event_time','orderid',data=df), plt.xticks(rotation=45)
TOTAL_ORDERS = df['orderid'].sum()
TOTAL_SHOPS = orders['shopid'].unique().size
MIN_DATE, MAX_DATE = df['event_time'].min(), df['event_time'].max()
ax.set_title('Orders from {} to {}\nTotal orders: {}, Total shops: {}'.\
             format(MIN_DATE,MAX_DATE,TOTAL_ORDERS,TOTAL_SHOPS));
```



1.2.2 Divide and conquer

When dealing with any problems, it is always good to divide and conquer, especially when we deal with a massive dataset like this.

Let us consider only **shop ID: 8996761** and 1hr period after **order ID: 31463329902935**

We choose this shop and order ID as it is a confirmed case provided by Shopee.

These are the steps we will conduct to determine if this shop has fake orders and the specific user involved in the fraud activities:

1. Determine the 1 hour timing after order has been conducted, orders within this range will be investigated
2. Compute the concentrate rate
3. If concentrate rate >= 3, determine order proportion for each users
4. Return the shopid and userid if there is order brushing, else return shopid and 0

[6]:
```
SHOP_ID, ORDER_ID = 8996761, 31463329902935
```

[7]:
```
SHOP_DF = orders[orders['shopid'] == SHOP_ID] #filter to only the shop's orders
SHOP_DF = SHOP_DF.assign(fraud_time=SHOP_DF['event_time'] + np.
 ↪timedelta64(1,'h'))#fraud time is 1hr after order
```

1. Determine the 1 hour timing after order has been conducted, orders within this range will be investigated

[8]:
```
#let's get investigation start and end time
START_TIME, END_TIME = SHOP_DF.loc[SHOP_DF['orderid'] ==␣
↪ORDER_ID,['event_time','fraud_time']].values[0]
print('Start time: {}\nEnd time: {}'.format(START_TIME,END_TIME))
```

```
Start time: 2019-12-31T11:48:49.000000000
End time: 2019-12-31T12:48:49.000000000
```

[9]:
```
#now, let's get all the orders that falls within the investigation time,␣
↪including the investigation time
INVESTIGATION_DF = SHOP_DF[(SHOP_DF['event_time'] >= START_TIME) &␣
↪(SHOP_DF['event_time'] <= END_TIME)]
INVESTIGATION_DF
```

[9]:
```
                orderid   shopid     userid          event_time  \
17116   31463329902935  8996761  215382704 2019-12-31 11:48:49
26346   31463906062704  8996761    2136861 2019-12-31 11:58:26
64862   31463701425020  8996761  215382704 2019-12-31 11:55:01
83426   31463960795761  8996761    2136861 2019-12-31 11:59:20
166235  31463618079296  8996761  215382704 2019-12-31 11:53:38
197220  31463516755431  8996761  215382704 2019-12-31 11:51:56

                fraud_time
17116  2019-12-31 12:48:49
26346  2019-12-31 12:58:26
64862  2019-12-31 12:55:01
83426  2019-12-31 12:59:20
166235 2019-12-31 12:53:38
197220 2019-12-31 12:51:56
```

2.Compute the concentrate rate

[10]:
```
def concentrate_rate(df):
    TOTAL_ORDERS = df['orderid'].size
    TOTAL_UNIQUE_USERS = df['userid'].unique().size
    return TOTAL_ORDERS/TOTAL_UNIQUE_USERS
```

Since concentrate rate > 3, shop is considered order brushing

[11]:
```
concentrate_rate(INVESTIGATION_DF)
```

[11]:
```
3.0
```

3. If concentrate rate >= 3, determine order proportion for each users

user 215382704 is our suspicious buyer

[12]:
```
INVESTIGATION_DF['userid'].value_counts()/INVESTIGATION_DF['userid'].size
```

[12]:
```
215382704    0.666667
2136861      0.333333
Name: userid, dtype: float64
```

4. Return the shopid and userid if there is order brushing, else return shopid and 0

Since this is just an illustration, let's just display the expected result here:

(8996761, 215382704)

1.2.3 Deployment

Now let's build a workflow to automate our process here

[13]:
```
class OrderBrushing:
    def __init__(self, df):
        self.df = df
        self.result = {}
        self.output = None

    #########
    #metrics#
    #########
    #concentrate rate
    def concentrate_rate(self, df):
        total_orders = df['orderid'].size
        total_unique_users = df['userid'].unique().size
        return total_orders/total_unique_users

    #order proportion
    def order_proportion(self, df):
        user_orders = df['userid'].value_counts()
        total_orders = df['orderid'].size
        return user_orders/total_orders

    ####################
    #data preprocessing#
    ####################
    #add fraud time to df
    def add_1hr_time(self, df):
        self.df = self.df.assign(fraud_time=df['event_time'] + np.
↪timedelta64(1,'h'))
        return self.df

    #return all shop id
    def get_shop_id(self, df):
        shop_id = df['shopid'].unique()
```

```
        return shop_id

    #get orders belonging to a specific shop
    def get_shop_df(self, df, shop_id):
        shop_df = df[df['shopid']==shop_id]
        return shop_df

    ##############
    #calculations#
    ##############
    #determine if user is suspicious, given a specific shop's order brushing␣
↪orders
    #assumption: we assume the df contains order brushing orders
    def suspicious_users(self,df):
        order_proportion = self.order_proportion(df)
        index = order_proportion == np.amax(order_proportion)
        suspicious_users = order_proportion[index].index.values
        return suspicious_users

    #determine if a shop is order brushing
    def order_brushing_shop(self, df):
        concentrate_rate = self.concentrate_rate(df)
        return True if concentrate_rate>=3 else False

    ###########
    #algorithm#
    ###########
    #return a df that contains all the orders within 1hr period, given a␣
↪specific row
    def get_investigation_df(self, df, row):
        df = df.reset_index(drop=True)
        start_time, end_time = df.iloc[row, 3], df.iloc[row, 4]
        orders_index = (df['event_time'] >= start_time) & (df['event_time'] <=␣
↪end_time)
        investigation_df = df[orders_index]
        return investigation_df

    #investigate if there is order brushing behaviour for a given shop
    def investigate(self, df, shop_id):
        shop_df = self.get_shop_df(df, shop_id)
        shop_fraud_df = []
        orders_index = np.arange(shop_df['orderid'].size)
        investigation_dfs = pd.Series(orders_index).apply(lambda x: self.
↪get_investigation_df(shop_df,x))
        frauds_index = investigation_dfs.apply(self.order_brushing_shop)
        if frauds_index.any():
```

```
            shop_fraud_df = pd.concat(investigation_dfs[frauds_index].values,␣
↪axis=0)
            fraud_users = self.suspicious_users(shop_fraud_df)
        else:
            fraud_users = [0]
        return [shop_id, fraud_users]

    ############
    #production#
    ############
    #return our final result
    def get_output(self):
        return self.output

    #now let's run the magic
    def run(self):
        self.df = self.add_1hr_time(self.df)
        shop_ids = pd.Series(self.get_shop_id(self.df))
        self.result = shop_ids.apply(lambda x: self.investigate(self.df, x))
        self.output = pd.DataFrame([map(lambda df: df[0], self.result),␣
↪map(lambda df: df[1], self.result)]).T
        self.output.columns = ['shopid','userid']
        self.output['userid'] = self.output['userid'].astype('str').str.
↪replace('  ','&').\
                                    astype('str').str.replace('[','').str.
↪replace(']','')
        return self
```

1.2.4 Vectorisation

Whenever I'm dealing with medium to large datasets like this, I'll remember this one time I used a for loop over <5k data and it ran for 3 days without completion.
Time complexitiy is a real issue.

In this competition, I spent ~30mins just trying to vectorise the process. The above pipeline is what I used during the competition, even though I avoided using any explicit for loops, the performance for pandas apply is not always ideal.
Since I have time now, I tried to clean up the codes to speed up the performance by vectorising the process using numpy. Let's compare the difference in performance.

For more information about the difference in speed between for loop, apply and numpy, I found this article pretty neat:
A Beginner's Guide to Optimizing Pandas Code for Speed:
https://engineering.upside.com/a-beginners-guide-to-optimizing-pandas-code-for-speed-c09ef2c6a4d6

[14]:
```
class OrderBrushing_fast:
    def __init__(self,array):
```

```
        self.array = array
        self.result = None
        self.output = None

    #########
    #metrics#
    #########
    #concentrate rate
    def concentrate_rate(self, array):
        total_orders = array[:,1].size
        total_unique_users = np.unique(array[:,2]).size
        return total_orders/total_unique_users

    #order proportion
    def order_proportion(self, array):
        user_orders = np.unique(array[:,2],return_counts=True)
        total_orders = array.shape[0]
        order_proportion = np.vstack([user_orders[0], user_orders[1]/
↪total_orders])
        return order_proportion

    ####################
    #data preprocessing#
    ####################
    #add fraud time to array
    def add_1hr_time(self, array):
        new_col = (array[:,3] + timedelta(hours=1)).reshape(-1,1)
        self.array = np.hstack([array,new_col])
        return self.array

    #return all shop id
    def get_shop_id(self, array):
        shop_id = np.unique(array[:,1]).reshape(-1,1)
        return shop_id

    #get orders belonging to a specific shop
    def get_shop_array(self, array, shop_id):
        shop_array = array[array[:,1] == shop_id]
        return shop_array

    ##############
    #calculations#
    ##############
    #determine if user is suspicious, given a specific shop's order brushing␣
↪orders
    #assumption: we assume the array contains order brushing orders
    def suspicious_users(self, array):
```

```
        users, user_proportion = self.order_proportion(array)
        highest_proportion = np.max(user_proportion)
        highest_proportion_user = users[user_proportion == highest_proportion]
        return highest_proportion_user

    #determine if a shop is order brushing
    def order_brushing_shop(self, array):
        concentrate_rate = self.concentrate_rate(array)
        return True if concentrate_rate>=3 else False

    ###########
    #algorithm#
    ###########
    #return an array that contains all the orders within 1hr period, given a␣
↪specific row
    def get_investigation_array(self, array, row):
        start_time, end_time = array[row,3], array[row,4]
        orders_index = (array[:,3] >= start_time) & (array[:,3] <= end_time)
        investigation_array = array[orders_index]
        return investigation_array

    #investigate if there is order brushing behaviour for a given shop
    def investigate(self, array, shop_id):
        shop_array = self.get_shop_array(array, shop_id)
        shop_fraud_df = []
        orders_index = np.arange(shop_array[:,1].size)
        investigation_arrays = np.array(list(map(lambda x: self.
↪get_investigation_array(shop_array,x),
                                                   orders_index)))
        frauds_index = np.array(list(map(self.order_brushing_shop,␣
↪investigation_arrays)))
        if frauds_index.any():
            shop_fraud_array = np.vstack(investigation_arrays[frauds_index])
            fraud_users = self.suspicious_users(shop_fraud_array)
        else:
            fraud_users = 0
        return [shop_id, fraud_users]

    ############
    #production#
    ############
    #return our final result
    def get_output(self):
        return self.output

    #now let's run the magic
    def run(self):
```

```
        self.array = self.add_1hr_time(self.array)
        shop_ids = np.hstack(self.get_shop_id(self.array))
        self.result = np.array(list(map(lambda x: self.investigate(self.array,␣
↪x), shop_ids)))
        self.output = pd.DataFrame([map(lambda df: df[0], self.result),␣
↪map(lambda df: df[1], self.result)]).T
        self.output.columns = ['shopid','userid']
        self.output['userid'] = self.output['userid'].astype('str').str.
↪replace(' ','&').\
                                        astype('str').str.replace('[','').str.
↪replace(']','')
        return self
```

1.2.5 Compare performance between Vectorization with Pandas series and NumPy arrays

We only use the first 10 rows to illustrate the speed difference

[15]:
```
%%timeit
#Pandas
slow_pipe = OrderBrushing(orders.iloc[:10,:]).run()
```

```
26.1 ms ± 957 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[16]:
```
#convert data to numpy format
orders_array = pd.read_csv('order_brush_order.csv').values
orders_array[:,3] = np.array(orders_array[:,3],dtype='datetime64')
```

[17]:
```
%%timeit
#Numpy
fast_pipe = OrderBrushing_fast(orders_array[:10,:]).run()
```

```
2.69 ms ± 196 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Numpy is the clear winner. Note that not only now our run time is 10x faster, we have a more consistent runtime.
However, Pandas is really useful for the user-friendly methods. One needs to weigh between speed and convenience

1.2.6 Pipeline

With a proper pipeline, our problem becomes so easy now!

We can deploy this to any new dataframe we face in the future
We will use the Numpy method since it's faster

[18]:
```
pipe = OrderBrushing_fast(orders_array).run()
result = pipe.get_output()
result.head()
```

[18]:

	shopid	userid
0	10009	0
1	10051	0
2	10061	0
3	10084	0
4	10100	0

[19]:
```
#let's catch some fake orders
result[~result['userid'].isin(['0'])].head()
```

[19]:

	shopid	userid
40	10402	77819
57	10536	672345
111	42472	740844
114	42818	170385453
129	76934	190449497

That's it, we have successfully find out shops that are order brushing!

1.2.7 Concluding remarks

It was an intense 3 hours from understanding the problem to translating the algorithm.
We manage to get 0.89196 out of best 1 and we still have so much more things to do to catch the missing cases!