Product Data Science

March 12, 2019

1 Q1. SQL

All the questions are answered under Python by using pymysql library which can execute mysql queries in Python.

```
In [1]: #import pymysql library and connect to my local database.
        import pymysql
        conn=pymysql.connect(host='localhost',user='root',password='********',db='uber')
        a=conn.cursor()
```

1.1 Part A

I created the 'uber_trip' table and stored it into the database 'uber' through MySQL Workbench. The contents of the table are listed in the Appendix at the end of this document. The first 10 rows of the table are list below:

```
In [3]: sql1="""
        select * from uber_trip
        limit 10
        """
        a.execute(sql1)
        result=a.fetchall()
        print('(rider_id, trip_id, begintrip_timestamp_utc, trip_status)')
        print(result)
```

```
(rider_id, trip_id, begintrip_timestamp_utc, trip_status)
(('rider06', 'd300000', '2017-07-04T18:18:17Z', 'completed'),
('rider05', 'd300001', '2017-07-06T02:57:41Z', 'completed'),
('rider01', 'd300002', '2017-07-06T02:57:41Z', 'not completed'),
('rider02', 'd300003', '2017-07-06T19:23:12Z', 'completed'),
('rider01', 'd300004', '2017-07-08T23:47:31Z', 'completed'),
('rider07', 'd300005', '2017-07-10T17:30:49Z', 'completed'),
('rider03', 'd300006', '2017-07-10T18:35:43Z', 'completed'),
('rider05', 'd300007', '2017-07-11T20:59:54Z', 'completed'),
('rider03', 'd300008', '2017-07-12T20:12:45Z', 'completed'),
('rider01', 'd300009', '2017-07-12T21:36:28Z', 'completed'))
```

The queries to return the trip_id for the 5th completed trip for each rider are shown below:

```
In [139]: sql2="""
          select trip_id from
          (select rider_id, trip_id, RANK() over (partition by rider_id
          order by begintrip_timestamp_utc asc) as completed_trip_no
               from (select rider_id, trip_id, begintrip_timestamp_utc
               from uber_trip where trip_status='completed') st1) st2
               where completed_trip_no = 5
          """
          a.execute(sql2)
          result=a.fetchall()
          print('(trip_id)')
          print(result)

(trip_id)
(('d300029',), ('d300016',))
```

The above output is correct.

1.2 Part B

I created the 'trip_initiated', 'trip_cancel' and 'trip_complete' tables and stored them into the database 'uber' through MySQL Workbench. The contents of the tables are listed in the Appendix at the end of this document. The queries to create the 'dispatch_events' table are shown below:

```
In [ ]: sql3="""
        create table dispatch_events
        select trip_initiated.trip_id, trip_initiated.rider_id,
        trip_initiated.driver_id, trip_initiated.timestamp as initiated_ts,
        trip_cancel.timestamp as cancel_ts, trip_complete.timestamp as complete_ts
        from trip_initiated left join trip_cancel on
        trip_initiated.trip_id = trip_cancel.trip_id
        left join trip_complete on trip_initiated.trip_id =
        trip_complete.trip_id order by trip_initiated.trip_id"""
        a.execute(sql3)
```

The first 5 rows of the table 'dispatch_events' are shown below:

```
In [5]: sql4="""
        select * from dispatch_events
        limit 5"""
        a.execute(sql4)
        result=a.fetchall()
        print('(trip_id, rider_id, driver_id, initiated_ts, cancel_ts, complete_ts)')
        print(result)
```

```
(trip_id, rider_id, driver_id, initiated_ts, cancel_ts, complete_ts)
(('d300000', 'rider00', 'driver00', '2017-07-08T23:47:31Z', None, '2017-07-08T23:57:31Z'),
('d300001', 'rider01', 'driver01', '2017-07-25T20:41:45Z', None, '2017-07-25T20:45:45Z'),
('d300002', 'rider02', 'driver02', '2017-07-25T20:41:45Z', '2017-07-25T20:42:45Z', None),
('d300003', 'rider03', 'driver03', '2017-07-25T20:41:45Z', None, '2017-07-25T21:41:45Z'),
('d300004', 'rider04', 'driver04', '2017-07-25T20:41:45Z', None, '2017-07-25T20:41:45Z'))
```

1.3 Part C

In this part, I want to write two queries to validate the table 'dispatch_events'.

The first query is used to check whether the 'trip_id' is unique, which is shown below:

```
In [6]: sql5="""
        select distinct(trip_id), count(trip_id) as count
        from dispatch_events
        group by trip_id
        having count > 1"""
        a.execute(sql5)
        result=a.fetchall()
        print(result)
```

```
()
```

The empty output as shown above should be the expect output.

The second query is used to make sure no trips have both nonempty 'cancel_ts' and 'complete_ts', which is shown below:

```
In [8]: sql6="""
        select *
        from dispatch_events
        where cancel_ts is not null and complete_ts is not null"""
        a.execute(sql6)
        result=a.fetchall()
        print(result)
```

```
()
```

The empty output as shown above should be the expect output.

2 Q2. Experimental Design

2.1 Part A

2.1.1

As I can see, the most direct success metric of this test is the **number of trips that completed by each driver** between 4pm and 8pm. If this incentive structure works, this metric should increase

significantly. However, since this incentive structure requires the drivers to complete at least 5 trips in the times of peak demand, it may not work well when most of the drivers usually complete 5 trips in this time period without the incentive structure. This may need to be concerning when choosing the test sample of drivers.

Also, another very valuable success metric is the **waiting time of each trip** (the time between the guest sending a demand and a drive accepting the demand). In my opinion, the main purpose of this incentive structure is to release the lack of drivers that complete the trips in the peak time period, which can decrease the waiting time of the guests and improve the experience of the guests. So this should be another good success metric of this test.

Besides the two success metrics mentioned above, some other invariant tracking metrics should also be monitored. One tracking metric should be the **numbers of demands** in the peak time. This metric should keep almost constant so that we can make sure that the change of the success metric is caused by the incentive structure but not the change of numbers of demands. Also, the **average trip lasting time** should also be an invariant tracking metric since if the trip lasting time change, it will effect the number of trips a driver can complete within a limited time period.

In summary, I suggest two metrics that can be success metric: 1. number of trips that completed by each driver; 2. waiting time of each trip. Also two invariant metrics are 1. numbers of demands; 2. trip lasting time. All the metrics are for the times of peak demand (4pm-8pm).

2.1.2

The ideal time period of this test should be about 1 to 2 months. The success and tracking metrics mentioned above should keep almost constant in the same months in the past years. Also we need to select suitable experimental samples. Since the drivers in same city may affect each other, we need to select the driver samples (S1 and S2) in at least two cities or two sets of cities, which have nearly similar numbers of drivers, demands (or trips) as well as the average trip lasting time and waiting time in the past observations. Considering the statistical rigor, in order to reduce the effects of different time and driver samples, we can build a Latin Square design shown by the figure below. In this design, the time period is divided into two equal length parts T1 and T2, which should be both 2 or 4 complete weeks (to remove the weekly cycle effect). In T1 we assign the incentive structure only to S1 and then in T2 we end the incentive structure in S1 and assign it to S2. Finally, combining the observed data in S1 and S2, we can obtain two sets of data with (S1 in T1 + S2 in T2) and without (S2 in T1 + S1 in T2) the incentive structure.

Sample\Time	T1	T2
S1	incentive	no incentive
S2	no incentive	incentive

2.1.3

First, I will do the sanity check, which means that test whether the invariant metrics are similar for the two data groups: control group without incentive structure and experimental group with

incentive structure. For the two data sample of the testing metric x_1,x_2, we assume the following null hypothesis:

$H_0 : \mu_1 - \mu_2 = 0$ which means the mean values of the metric in the two sample group are the same, with the alternative hypothesis:

$H_A : \mu_1 - \mu_2 \neq 0$

The test statistic T should obey normal distribution since the sample size is large (>30):

$T = \frac{\bar{x_1} - \bar{x_2}}{\sqrt{Var(\bar{x_1} - \bar{x_2})}} \sim N(0, 1)$

where $\bar{x_1}$ and $\bar{x_2}$ are the mean values of variables x_1,x_2, Var means the variance which can be estimated by $\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}$, where n_1, n_2 are the sample sizes of the two data samples and s_1, s_2 are the corresponding standard deviations. Calculating the required parameters from the observed data we can make a confidence interval of $\mu_1 - \mu_2$, which should be:

$(\bar{x_1} - \bar{x_2}) \pm z_{\alpha/2} \times (\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2})$,

where $z_{\alpha/2} = 1.96$ for a 95% $(= 1 - \alpha)$ confidence level. If the confidence interval include 0, we will accept the null hypothesis and conclude the means of these two sample metrics are the same. Otherwise, we should conclude the means are different. Also, we can calculate the P value of the test statistic and if the P value is larger than $\alpha/2$, accept H_0, otherwise, accept H_A.

In the sanity check, if the invariant metrics keep constant (H_0 is accepted), we can move on to the test of the success metrics. If not, we need to check and modify the design of the experiment and repeat another trial.

After the sanity check, we need to check whether the success metrics for the two samples are different. We can just do the same hypothesis test in the sanity check mentioned above. If the mean value of the number of trips that completed by each driver is larger or the mean value of waiting time of each trip is smaller in the sample with incentive structure than those in the sample without incentive structure, we can conclude that the incentive structure performs well as we expect. Otherwise, we should conclude that this incentive structure is not a good method to increase the available supply in the peak time. By the way, when determining whether the means are the same or different, we need to set up a significant boundary, which is a critical value that to determine whether the difference between the two means are significant enough. Two kinds of significant boundaries need to be considered: statistical boundary and practical boundary.

2.2 Part B

One metric that most interests me is the **daily enhancement of the number of Uber drivers (the number of new Uber drivers)** before and after the campaign starting. This metric is the most direct and easy to obtain. If the campaign works, the number of new drivers should significantly increase after the campaign starting.

However, only checking the number of new Uber drivers is kind of one-sided. Maybe this number will always increase fast as time goes by even without the campaign. This can be solved by check the metric which is the **increase rate of number of new Uber drivers**. If the rate increase after the campaign starting, the conclusion that the campaign works will become stronger.

Another factor need to be consider is that the increase of number of new Uber drivers may caused by the increase of the number of all the drivers including Uber, Lyft and traditional taxi companies. Which means the increase of new Uber driver may be caused by more and more people become taxi drivers, but they may not prefer to work for Uber comparing with other companies. To solve this concerning, we need to check the metric which is the **ratio between new Uber drivers and all kinds of new taxi drivers**. If this ratio increases after the campaign, it should indicate that more new taxi drivers are likely to work for Uber due to the effect of the campaign.

However, this metric may be difficult to obtain since we need the data of new drivers in other companies.

3 Q3. Modeling

This questions are solved through Python and its corresponding libraries.

3.1 Part A

Import libraries and read the data.

```
In [103]: # data analysis and wrangling
          import pandas as pd
          import numpy as np
          import random as rnd

          # visualization
          import seaborn as sns
          import matplotlib.pyplot as plt
          %matplotlib inline

          # machine learning
          from sklearn.linear_model import LogisticRegression
          from sklearn.svm import SVC, LinearSVC
          from sklearn.ensemble import RandomForestClassifier
          from sklearn.neighbors import KNeighborsClassifier
          from sklearn.naive_bayes import GaussianNB
          from sklearn.linear_model import Perceptron
          from sklearn.linear_model import SGDClassifier
          from sklearn.tree import DecisionTreeClassifier
          from sklearn.model_selection import train_test_split
          df = pd.read_csv('ProductDataSet.csv')
```

Calculate three parameters of time difference (in unit of day): Dt1, the difference between first_completed_trip_timestamp and signup_timestamp; Dt2, the difference between bgc_date and signup_timestamp; Dt3, the difference between vehicle_added_date and signup_timestamp. Using the Dt1 to determine whether the drivers took a first trip within 30 days of signing up and save it as Within30.

```
In [140]: df['Dt1']=(pd.to_datetime(df['first_completed_trip_timestamp'])
                     -pd.to_datetime(df['signup_timestamp']))/np.timedelta64(1,'h')/24.
          df['Dt2']=(pd.to_datetime(df['bgc_date'])
                     -pd.to_datetime(df['signup_timestamp']))/np.timedelta64(1,'h')/24
          df['Dt3']=(pd.to_datetime(df['vehicle_added_date'])
                     -pd.to_datetime(df['signup_timestamp']))/np.timedelta64(1,'h')/24.
          df['Within30']=(df['Dt1']<=30)*1
          #df=df.set_index('id')
          df.head()
```

```
Out[140]:                                     id city_name signup_os signup_channel  \
          0  082befb0-c1de-4c14-8700-94a7943a7545    Strark       NaN            R2D
          1  1ae6156f-63fc-40cf-9734-0995978c4b6e    Berton   ios web           Dost
          2  2a4a4eef-14ef-4ceb-82eb-66f1f7d0d219    Berton       NaN            R2D
          3  56fe7597-3ad8-4798-8be8-5fbc4e2d3151    Berton   ios web       Referral
          4  67370341-68a5-415f-acf2-be58832a8f9c   Wrouver     other       Referral

                signup_timestamp              bgc_date    vehicle_added_date  \
          0  2017-07-06T20:42:17Z                   NaN  2017-07-06T20:42:56Z
          1  2017-07-03T17:41:07Z  2017-07-03T17:42:06Z  2017-07-03T17:41:14Z
          2  2017-07-10T22:55:29Z                   NaN  2017-07-11T17:44:44Z
          3  2017-07-27T18:27:21Z  2017-07-27T18:31:43Z  2017-07-27T18:31:09Z
          4  2017-07-17T22:20:35Z  2017-07-17T22:21:09Z  2017-07-17T22:20:53Z

            vehicle_make vehicle_model  vehicle_year ...  doy  city  os  channel  make  \
          0   Volkswagen            CC          2012 ...    6     3   0        4     1
          1       Toyota       4Runner          2003 ...    3     2   1        5     2
          2      Hyundai       Elantra          2015 ...   10     2   0        4     3
          3        Honda         Civic          2017 ...   27     2   1        1     4
          4        Honda        Accord          1999 ...   17     1   2        1     4

             year  hours   doys  check  add
          0    20   20.0  187.0      1    0
          1    11   17.0  184.0      0    0
          2    23   22.0  191.0      1    0
          3    25   18.0  208.0      0    0
          4     7   22.0  198.0      0    0

          [5 rows x 28 columns]
```

Here is a problem: some rows have null signup_timestamp or first_completed_trip_timestamp values, which leads to null Dt1 values. If we do not consider these rows, the fraction of drivers took a first trip within 30 days of signing up is:

```
In [105]: df1=df.dropna(subset=['first_completed_trip_timestamp', 'signup_timestamp'])
          df['Within30'].sum()/len(df1)

Out[105]: 0.9954464140510653
```

The fraction is nearly 1, which means almost all the drivers took a first trip within 30 days. Thus I think the data with null values should be considered and treated as drivers did not take a first trip within 30 day. Then the fraction becomes:

```
In [106]: df['Within30'].sum()/len(df)

Out[106]: 0.4953467670146476
```

The fraction is about 0.5, which means about half of the drivers took a first trip within 30 days.

Next we want to check whether the other features have significant effect on the Within30 value. First, let's check the city_name, the mean values of Within30 (equals to the fraction of drivers took a first trip within 30 days) for different city are shown below:

```
In [142]: df[['city_name', 'Within30']].groupby(['city_name'], as_index=False) \
          .mean().sort_values(by='Within30', ascending=False)

Out[142]:   city_name  Within30
          2   Wrouver  0.505045
          0    Berton  0.498414
          1    Strark  0.491289
```

We can see slightly differences between different cities. The next feature is the signup_os:

```
In [108]: df[['signup_os', 'Within30']].groupby(['signup_os'], as_index=False) \
          .mean().sort_values(by='Within30', ascending=False)

Out[108]:      signup_os  Within30
          1      ios web  0.542609
          3        other  0.517879
          4      windows  0.510086
          0  android web  0.498919
          2          mac  0.474930
```

There are some considerable differences. Then we check the signup_channel:

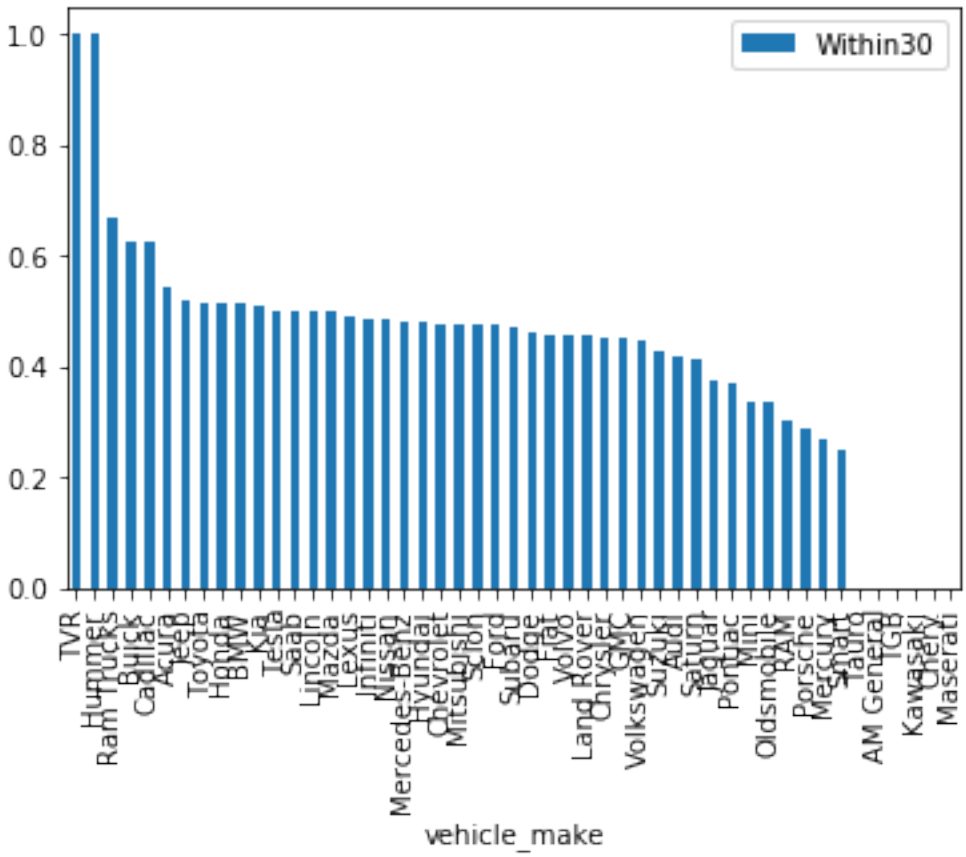
```
In [109]: df[['signup_channel', 'Within30']].groupby(['signup_channel'], as_index=False) \
          .mean().sort_values(by='Within30', ascending=False)

Out[109]:   signup_channel  Within30
          4       Referral  0.559298
          1        Organic  0.452481
          2           Paid  0.430417
          3            R2D  0.405158
          0           Dost  0.386364
```

The differences are significant.
Next, we are going to check the effect of vehicle_make:

```
In [143]: df[['vehicle_make', 'Within30']].groupby(['vehicle_make'], as_index=False) \
          .mean().sort_values (by='Within30', ascending=False).plot.bar(x='vehicle_make')

Out[143]: <matplotlib.axes._subplots.AxesSubplot at 0x15cb4cf5e48>
```

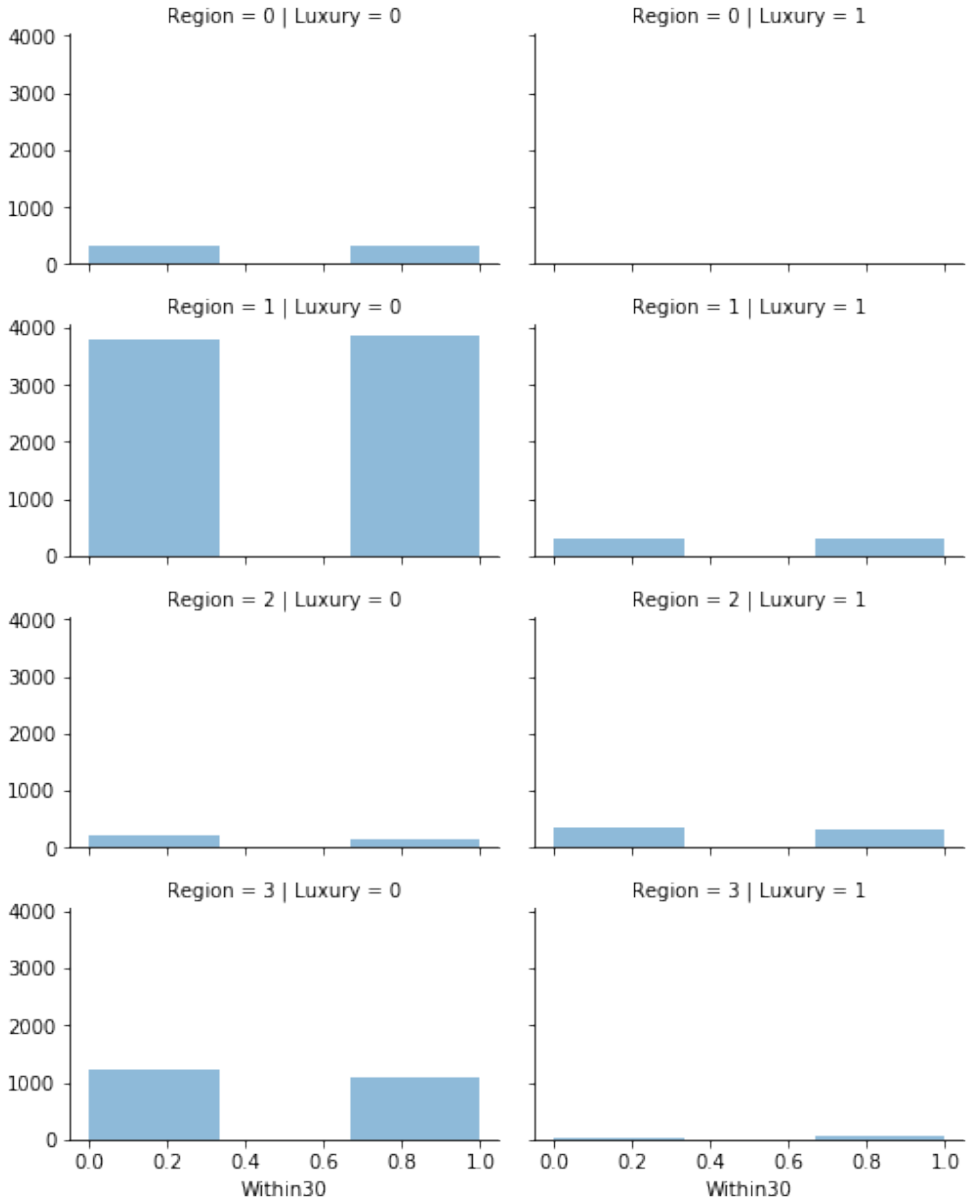


There are differences between different makes. However, the amount of values of vehicle_make is very large. Here I tried to classify the makes into 3 regions: Europe, Asia and America, as well as two levels: luxury and not. Then we checked if there are significant effects of these two factors:

```
In [144]: df['Region']=0
          luxury=['Acura','Audi','Lexus','Infiniti','BMW','Mercedes-Benz','Volvo','Land Rover',
                  'Porsche','Maserati','Cadillac','Tesla','Jaguar']
          asia=['Toyota', 'Honda', 'Nissan','Acura','Kia','Mazda','Lexus','Infiniti','Hyundai',
                'Mitsubishi','Scion','Subaru','Suzuki','TGB','Kawasaki','Chery']
          europe=['TVR', 'BMW', 'Saab','Mercedes-Benz','Fiat','Volvo','Land Rover','Volkswagen',
                  'Audi','Mini','Porsche','Smart','Tauro','Maserati']
          america=['Hummer','Ram Trucks','Buick','Cadillac','Jeep','Tesla','Lincoln','Chevrolet'
                   'Ford', 'Dodge','Chrysler','GMC','Saturn','Jaguar','Pontiac','Oldsmobile',
                   'RAM','Mercury','AM General']
          df['Region'].loc[df['vehicle_make'].isin(asia)]=1
          df['Region'].loc[df['vehicle_make'].isin(europe)]=2
          df['Region'].loc[df['vehicle_make'].isin(america)]=3
          df['Luxury']=(df['vehicle_make'].isin(luxury))*1
          grid = sns.FacetGrid(df, row='Region', col='Luxury', size=2.2, aspect=1.6)
```

```
grid.map(plt.hist, 'Within30', alpha=.5, bins=3)
grid.add_legend()
```

```
Out[144]: <seaborn.axisgrid.FacetGrid at 0x15cb70efcc0>
```

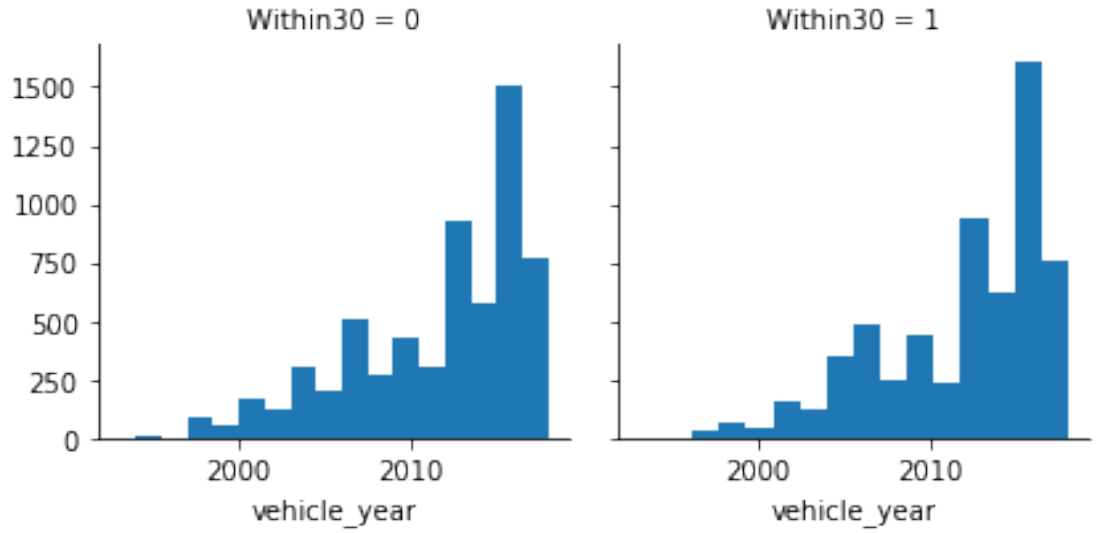


From the above figure, we can see that for different regions and levels, the amount of Within30=0 and Within30=1 are allot the same, which indicates very week effects of these two factors. The kinds of vehicle_model are much more than vehicle_make and I decide to not consider about effect of vehicle_model.

Next factor is the vehicle_year, which may be an important factor:

```
In [112]: g = sns.FacetGrid(df, col='Within30')
          g.map(plt.hist, 'vehicle_year', bins=16)
```

```
Out[112]: <seaborn.axisgrid.FacetGrid at 0x15cb4bb2fd0>
```

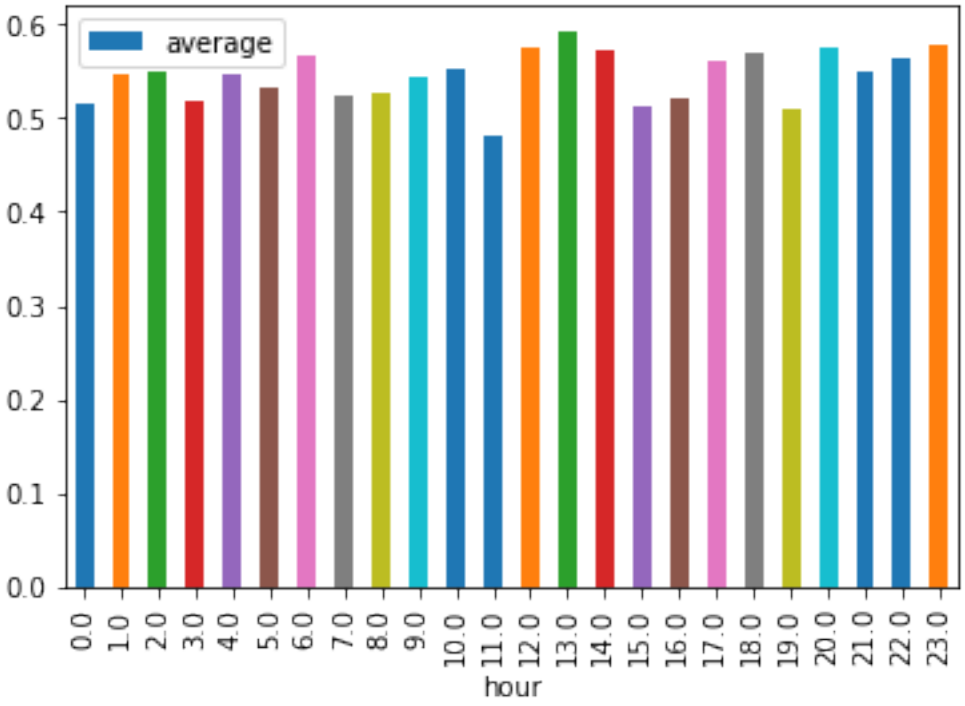


But the result shows that the distribution of the vehicle_year for different Within30 values are almost the same. This concludes that the vehicle_year is almost independent of the Within30 values.

Considering the three time difference values: the Dt1 is equivalent to the Within30. The Dt2 and Dt3 values mostly distributed about several minutes, which should have no important influence. Mining the data deeper, I consider that maybe the day and hour of signing up are important factors. Thus I plot the bar plot of the mean Within30 values for different days and hours:

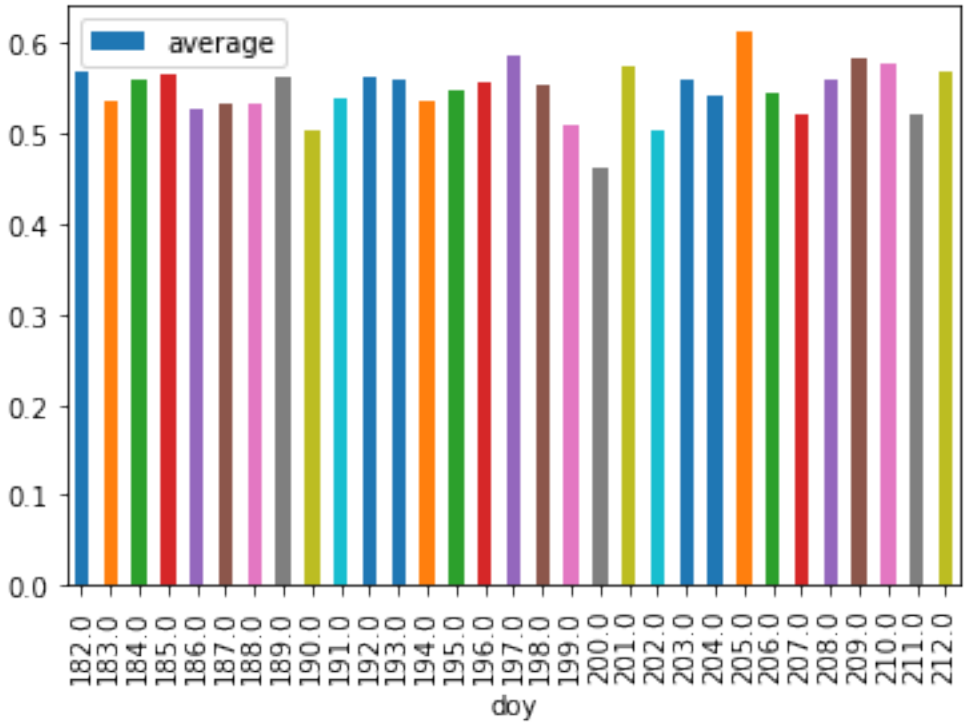
```
In [113]: hc = df
          hc['hour'] = pd.to_datetime(hc['signup_timestamp']).dt.hour
          hc=hc.groupby('hour')['Within30'].agg(['sum','count'])
          hc.reset_index
          hc['average']=hc['sum']/hc['count']
          hc.plot.bar(y='average')

Out[113]: <matplotlib.axes._subplots.AxesSubplot at 0x15cb4cd5668>
```



In [114]:
```
dc = df
dc['doy'] = pd.to_datetime(dc['signup_timestamp']).dt.dayofyear
dc=dc.groupby('doy')['Within30'].agg(['sum','count'])
dc.reset_index
dc['average']=dc['sum']/dc['count']
dc.plot.bar(y='average')
```

Out[114]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x15cb4d1fb00>
```



There are considerable differences between days and hours, but without regularity.

3.2 Part B

Now we need to build the classification model. First, I assume that the value of Within30 is affected by the city, os, channel, make, region and level of vehicle, vehicle year, day and hour of signing up. I transform all the values of the above factors into positive integer and null values are set to be 0.

```
In [145]: df['city'] = df['city_name'].map( {'Wrouver': 1, 'Berton': 2, 'Strark':3 , \
                                    np.nan:0} ).astype(int)
          df['os'] = df['signup_os'].map( {'ios web': 1, 'other': 2, 'windows':3, \
                                   'android web':4, 'mac':5, np.nan:0} ).astype(int)
          df['channel'] = df['signup_channel'].map( {'Referral': 1, 'Organic': 2, \
                                   'Paid':3, 'R2D':4, 'Dost':5, np.nan:0} ).astype(int)
          makes=df.vehicle_make.unique()
          df['make']=0
          for i in range(len(makes)):
              df['make'].loc[df['vehicle_make']==makes[i]]=i+1
          years=np.sort(df.vehicle_year.unique())
          df['year']=0
          for i in range(len(years)):
              df['year'].loc[df['vehicle_year']==years[i]]=i+1
          df['hours'] = pd.to_datetime(df['signup_timestamp']).dt.hour
```

```
hours=np.sort(df.hours.unique())
df['hour']=0
for i in range(len(hours)):
    df['hour'].loc[df['hours']==hours[i]]=i+1

df['doys'] = pd.to_datetime(df['signup_timestamp']).dt.dayofyear
doys=np.sort(df.doys.unique())
df['doy']=0
for i in range(len(doys)):
    df['doy'].loc[df['doys']==doys[i]]=i+1
```

Then create the data set of the model and split them into training and testing samples. The size of the testing sample is 25% of the total sample.

```
In [146]: df_model=df[['Within30','Region','Luxury','make','year','doy','hour',
                      'city','os','channel']]
          train, test = train_test_split(df_model, test_size=0.25)
          Y_train=train['Within30']
          X_train=train.drop('Within30', axis=1)
          Y_test=test['Within30']
          X_test=test.drop('Within30', axis=1)
```

```
In [117]: df_model.head()
```

Out[117]:

	Within30	Region	Luxury	make	year	doy	hour	city	os	channel
0	0	2	0	1	20	6	21	3	0	4
1	1	1	0	2	11	3	18	2	1	5
2	0	1	0	3	23	10	23	2	0	4
3	1	1	0	4	25	27	19	2	1	1
4	0	1	0	4	7	17	23	1	2	1

The size of the training input, training output, testing input and testing output:

```
In [118]: X_train.shape, Y_train.shape, X_test.shape, Y_test.shape
```

```
Out[118]: ((9267, 9), (9267,), (3090, 9), (3090,))
```

Finally, we can training the classification model now. In this program, I choose five models: Logistic Regression, Support Vector Machines(SVM), Decision Tree, Random Forest and Gaussian Naive Bayes. The codes for training models and calculating the performances of the models are shown in the Appendix at the end of this document. The performance scores (accuracy rate) of different models are list below:

```
In [120]: models = pd.DataFrame({
              'Model': ['Support Vector Machines', 'Logistic Regression',
                        'Random Forest', 'Naive Bayes', 'Decision Tree'],
              'Score_train': [acc_svc1, acc_log1, acc_random_forest1,
                              acc_gaussian1, acc_decision_tree1],
              'Score_test': [acc_svc2, acc_log2, acc_random_forest2,
                             acc_gaussian2, acc_decision_tree2]})
          models.sort_values(by='Score_test', ascending=False)
```

```
Out[120]:                    Model  Score_train  Score_test
          2            Random Forest        99.89       61.17
          0  Support Vector Machines        88.32       60.42
          3              Naive Bayes        59.18       59.42
          1      Logistic Regression        58.77       58.87
          4            Decision Tree        99.89       57.48
```

From the scores above, we can see all the models performs not very well and similarly on the testing data but the Random Forest, Support Vector Machines, Decision Tree models performs very well on the training data. This result indicates an overfitting of these models. In order to improve the performance of the models, we can drop some features to reduce the overfitting. Before deciding which features should be dropped, we calculated the coefficient of the features in the decision function of the Logistic Regression model. The results are:

```
In [121]: coeff_df = pd.DataFrame(train.columns.delete(0))
          coeff_df.columns = ['Feature']
          coeff_df["Correlation"] = pd.Series(logreg.coef_[0])

          coeff_df.sort_values(by='Correlation', ascending=False)
```

```
Out[121]:    Feature  Correlation
          1   Luxury     0.074305
          5     hour     0.032695
          4      doy     0.028052
          3     year     0.016607
          2     make     0.000433
          7       os    -0.013844
          6     city    -0.028927
          0   Region    -0.036735
          8  channel    -0.168236
```

From the table above, we can see all the features except the channel have poor correlations to the Within30 values. This result is similar to what we obtained in Part A. Thus I decide to drop the 3 features with lowest correlations which are year, make and os. Then treat the models again and obtain the new scores:

```
In [122]: X_train=train.drop(['Within30','year','make','os'], axis=1)
          X_test=test.drop(['Within30','year','make','os'], axis=1)
```

```
In [124]: # run the model training and  scoring codes (not shown)
```

```
Out[124]:                    Model  Score_train  Score_test
          0  Support Vector Machines        69.50       60.91
          1      Logistic Regression        58.53       58.77
          3              Naive Bayes        58.63       58.54
          2            Random Forest        90.29       58.32
          4            Decision Tree        90.29       57.12
```

From the results, we can see that both the performances on testing and training data decrease. This indicates that the overfitting problem can not be solved by simply reduced the features we selected. To improve the performance of the models, we need to mine the data to exact some important features as well as collect data of more useful features such as the ages and genders of the drivers.

Looking back to the data, I found that whether the data of bgc_date and vehicle_added_date are null maybe important features. Thus we added two features named check and add and checked their effects:

```
In [125]: df['check']=pd.isnull(df['bgc_date'])*1
          df[['check', 'Within30']].groupby(['check'], as_index=False).mean() \
          .sort_values(by='Within30', ascending=False)
```

```
Out[125]:    check  Within30
          0      0  0.516340
          1      1  0.459376
```

```
In [126]: df['add']=pd.isnull(df['vehicle_added_date'])*1
          df[['add', 'Within30']].groupby(['add'], as_index=False).mean() \
          .sort_values(by='Within30', ascending=False)
```

```
Out[126]:    add  Within30
          1    1  0.511327
          0    0  0.493571
```

The effect of check is considerable. Thus we added it into the original training data and training the models again:

```
In [132]: X_train=train.drop('Within30', axis=1)
          X_test=test.drop('Within30', axis=1)
          X_train['check']=df.loc[ X_train.index, 'check']
          X_test['check']=df.loc[ X_test.index, 'check']
```

```
In [136]: # run the model training and  scoring codes (not shown)
```

```
Out[136]:                    Model  Score_train  Score_test
          2            Random Forest        99.90       62.20
          0  Support Vector Machines        87.32       60.10
          3              Naive Bayes        59.20       59.77
          1      Logistic Regression        58.70       59.00
          4            Decision Tree        99.90       58.09
```

The models perform a little bit now, but still not well enough.

3.3 Part C

The model results show that whether the driver would take a first trip within 30 days is almost independent of the drivers' vehicle information, location and sign up OS. But the sign up channel is a important feature: the drivers sign up through Referral channel have significant larger probability to generate first trips. Thus in order to generate more first trips, Uber should pay more attention to the sign up channel, especially the Referral channel. Some structures such as a reward for reference may be applied. However, more data of other features are needed to obtain better models and find out other important methods to generate more first trips.

4 Appendix

The content of table 'uber_trip':

rider_id	trip_id	begintrip_timestamp_utc	trip_status
rider06	d300000	2017-07-04T18:18:17Z	completed
rider05	d300001	2017-07-06T02:57:41Z	completed
rider01	d300002	2017-07-06T02:57:41Z	not completed
rider02	d300003	2017-07-06T19:23:12Z	completed
rider01	d300004	2017-07-08T23:47:31Z	completed
rider07	d300005	2017-07-10T17:30:49Z	completed
rider03	d300006	2017-07-10T18:35:43Z	completed
rider05	d300007	2017-07-11T20:59:54Z	completed
rider03	d300008	2017-07-12T20:12:45Z	completed
rider01	d300009	2017-07-12T21:36:28Z	completed
rider05	d300010	2017-07-13T04:35:12Z	completed
rider08	d300011	2017-07-14T02:27:09Z	completed
rider05	d300012	2017-07-19T15:22:48Z	not completed
rider02	d300013	2017-07-20T04:17:42Z	completed
rider02	d300014	2017-07-20T08:26:24Z	completed
rider05	d300015	2017-07-20T19:33:52Z	completed
rider05	d300016	2017-07-22T18:06:42Z	completed
rider04	d300017	2017-07-23T23:58:39Z	completed
rider03	d300018	2017-07-24T00:21:44Z	completed
rider03	d300019	2017-07-25T00:23:46Z	not completed
rider02	d300020	2017-07-25T20:41:45Z	completed
rider01	d300021	2017-07-25T20:41:45Z	not completed
rider03	d300022	2017-07-25T20:41:45Z	completed
rider04	d300023	2017-07-25T20:41:45Z	completed
rider06	d300024	2017-07-27T15:05:35Z	completed
rider05	d300025	2017-07-29T18:56:36Z	completed
rider06	d300026	2017-07-30T22:42:58Z	completed
rider01	d300027	2017-07-30T23:47:21Z	completed
rider01	d300028	2017-07-31T00:33:44Z	completed
rider03	d300029	2017-07-31T02:05:45Z	completed

The content of table 'trip_initiated':

trip_id	rider_id	driver_id	timestamp
d300000	rider00	driver00	2017-07-08T23:47:31Z
d300001	rider01	driver01	2017-07-25T20:41:45Z
d300002	rider02	driver02	2017-07-25T20:41:45Z
d300003	rider03	driver03	2017-07-25T20:41:45Z
d300004	rider04	driver04	2017-07-25T20:41:45Z
d300005	rider05	driver05	2017-07-31T00:33:44Z
d300006	rider06	driver06	2017-07-31T02:05:45Z
d300007	rider07	driver07	2017-07-13T04:35:12Z
d300008	rider08	driver08	2017-07-25T00:23:46Z
d300009	rider09	driver09	2017-07-20T19:33:52Z
d300010	rider10	driver10	2017-07-23T23:58:39Z
d300011	rider11	driver11	2017-07-06T19:23:12Z
d300012	rider12	driver12	2017-07-04T18:18:17Z
d300013	rider13	driver13	2017-07-10T18:35:43Z
d300014	rider14	driver14	2017-07-06T02:57:41Z
d300015	rider15	driver15	2017-07-06T02:57:41Z
d300016	rider16	driver16	2017-07-19T15:22:48Z
d300017	rider17	driver17	2017-07-30T22:42:58Z
d300018	rider18	driver18	2017-07-10T17:30:49Z
d300019	rider19	driver19	2017-07-11T20:59:54Z

The content of table 'trip_cancel':

	trip_id	rider_id	driver_id	timestamp
▶	d300002	rider02	driver02	2017-07-25T20:42:45Z
	d300008	rider08	driver08	2017-07-25T00:25:46Z
	d300012	rider12	driver12	2017-07-04T18:19:17Z
	d300013	rider13	driver13	2017-07-10T18:36:43Z
	d300017	rider17	driver17	2017-07-30T22:43:58Z

The content of table 'trip_complete':

	trip_id	rider_id	driver_id	timestamp
▶	d300000	rider00	driver00	2017-07-08T23:57:31Z
	d300001	rider01	driver01	2017-07-25T20:45:45Z
	d300003	rider03	driver03	2017-07-25T21:41:45Z
	d300004	rider04	driver04	2017-07-25T20:41:45Z
	d300005	rider05	driver05	2017-07-31T01:33:44Z
	d300006	rider06	driver06	2017-07-31T02:23:45Z
	d300007	rider07	driver07	2017-07-13T04:44:12Z
	d300009	rider09	driver09	2017-07-20T19:52:52Z
	d300010	rider10	driver10	2017-07-23T23:59:39Z
	d300011	rider11	driver11	2017-07-06T19:42:12Z
	d300014	rider14	driver14	2017-07-06T03:57:41Z
	d300015	rider15	driver15	2017-07-06T03:57:41Z
	d300016	rider16	driver16	2017-07-19T15:32:48Z
	d300018	rider18	driver18	2017-07-10T17:45:49Z
	d300019	rider19	driver19	2017-07-11T21:59:54Z

The content of table 'dispatch_events':

	trip_id	rider_id	driver_id	initiated_ts	cancel_ts	complete_ts
▶	d300000	rider00	driver00	2017-07-08T23:47:31Z	NULL	2017-07-08T23:57:31Z
	d300001	rider01	driver01	2017-07-25T20:41:45Z	NULL	2017-07-25T20:45:45Z
	d300002	rider02	driver02	2017-07-25T20:41:45Z	2017-07-25T20:42:45Z	NULL
	d300003	rider03	driver03	2017-07-25T20:41:45Z	NULL	2017-07-25T21:41:45Z
	d300004	rider04	driver04	2017-07-25T20:41:45Z	NULL	2017-07-25T20:41:45Z
	d300005	rider05	driver05	2017-07-31T00:33:44Z	NULL	2017-07-31T01:33:44Z
	d300006	rider06	driver06	2017-07-31T02:05:45Z	NULL	2017-07-31T02:23:45Z
	d300007	rider07	driver07	2017-07-13T04:35:12Z	NULL	2017-07-13T04:44:12Z
	d300008	rider08	driver08	2017-07-25T00:23:46Z	2017-07-25T00:25:46Z	NULL
	d300009	rider09	driver09	2017-07-20T19:33:52Z	NULL	2017-07-20T19:52:52Z
	d300010	rider10	driver10	2017-07-23T23:58:39Z	NULL	2017-07-23T23:59:39Z
	d300011	rider11	driver11	2017-07-06T19:23:12Z	NULL	2017-07-06T19:42:12Z
	d300012	rider12	driver12	2017-07-04T18:18:17Z	2017-07-04T18:19:17Z	NULL
	d300013	rider13	driver13	2017-07-10T18:35:43Z	2017-07-10T18:36:43Z	NULL
	d300014	rider14	driver14	2017-07-06T02:57:41Z	NULL	2017-07-06T03:57:41Z
	d300015	rider15	driver15	2017-07-06T02:57:41Z	NULL	2017-07-06T03:57:41Z
	d300016	rider16	driver16	2017-07-19T15:22:48Z	NULL	2017-07-19T15:32:48Z
	d300017	rider17	driver17	2017-07-30T22:42:58Z	2017-07-30T22:43:58Z	NULL
	d300018	rider18	driver18	2017-07-10T17:30:49Z	NULL	2017-07-10T17:45:49Z
	d300019	rider19	driver19	2017-07-11T20:59:54Z	NULL	2017-07-11T21:59:54Z

The codes for traning and scoring models:

```
In [ ]: # Logistic Regression

        logreg = LogisticRegression(solver='lbfgs')
        logreg.fit(X_train, Y_train)
        Y_pred = logreg.predict(X_test)
        acc_log1 = round(logreg.score(X_train, Y_train) * 100, 2)
        acc_log2 = round(logreg.score(X_test, Y_test) * 100, 2)
        [acc_log1,acc_log2]

        # Support Vector Machines
```

```
svc = SVC(gamma='auto')
svc.fit(X_train, Y_train)
Y_pred = svc.predict(X_test)
acc_svc1 = round(svc.score(X_train, Y_train) * 100, 2)
acc_svc2 = round(svc.score(X_test, Y_test) * 100, 2)
[acc_svc1,acc_svc2]

# Decision Tree

decision_tree = DecisionTreeClassifier()
decision_tree.fit(X_train, Y_train)
Y_pred = decision_tree.predict(X_test)
acc_decision_tree1 = round(decision_tree.score(X_train, Y_train) * 100, 2)
acc_decision_tree2 = round(decision_tree.score(X_test, Y_test) * 100, 2)
[acc_decision_tree1,acc_decision_tree2]

# Random Forest

random_forest = RandomForestClassifier(n_estimators=100)
random_forest.fit(X_train, Y_train)
Y_pred = random_forest.predict(X_test)
random_forest.score(X_train, Y_train)
acc_random_forest1 = round(random_forest.score(X_train, Y_train) * 100, 2)
acc_random_forest2 = round(random_forest.score(X_test, Y_test) * 100, 2)
[acc_random_forest1,acc_random_forest2]

# Gaussian Naive Bayes

gaussian = GaussianNB()
gaussian.fit(X_train, Y_train)
Y_pred = gaussian.predict(X_test)
acc_gaussian1 = round(gaussian.score(X_train, Y_train) * 100, 2)
acc_gaussian2 = round(gaussian.score(X_test, Y_test) * 100, 2)
[acc_gaussian1,acc_gaussian2]
```