Elo Merchant Category Recommendation

Help understand customer loyalty

Procedure:

```
1)Business Problem
2)Data description
3)Exploratory Data Analysis
4)Data preparation/Feature engineering
5)Model Building
6)Submit model on kaggle(5% to 10%)
```

1. Business Problem

1.1 Problem Description:

“Elo Merchant Category Recommendation” challenge that is about helping understand customer loyalty using machine learning. Elo, a large Brazilian payment brand (focused on debit and credit cards), has built machine learning models to understand the most important aspects in their customers’ lifecycle. However, there is a major limitation to their existing models. So far none of their models is specifically tailored for a particular individual or a profile. That means that Elo cannot deliver fully personalized brand recommendations to its customers, nor can it filter unwanted ones.

What is loyalty? According to the Data_Dictionary.xlsx, loyalty is a numerical score calculated 2 months after historical and evaluation period. Additionally, by looking at historical_transactions.csv and new_merchant_transactions.csv, we can find that the historical transactions are the transactions occurred before the "reference date" and new merchant transactions - the ones that occurred after the reference date (according to the 'month_lag' field, which is generously described as "month lag to reference date").

we need to "develop algorithms to identify and serve the most relevant opportunities to individuals, by uncovering signals in customer loyalty". Competition description:

Imagine being hungry in an unfamiliar part of town and getting restaurant recommendations served up, based on your personal preferences, at just the right moment. The recommendation comes with an attached discount from your credit card provider for a local place around the corner!

Right now, Elo, one of the largest payment brands in Brazil, has built partnerships with merchants in order to offer promotions or discounts to cardholders. But do these promotions work for either the consumer or the merchant? Do customers enjoy their experience? Do merchants see repeat business? Personalization is key.

1.2 Problem Statement

Elo has built machine learning models to understand the most important aspects and preferences in their customers’ lifecycle, from food to shopping. But so far none of them is specifically tailored for an individual or profile. This is where you come in.

In this competition, Kagglers will develop algorithms to identify and serve the most relevant opportunities to individuals, by uncovering signal in customer loyalty. Your input will improve customers’ lives and help Elo reduce unwanted campaigns, to create the right experience for customers.

1.3 Real world/Business Objectives and constraints

Objectives:

```
Predict loyalty score to improve customers’ lives and help Elo reduce unwanted camp
```

```
aigns.
Minimize the difference between predicted and actual rating (RMSE)
```

1.4 Credits

References used for both exploratory and modelling.

1) https://www.kaggle.com/fabiendaniel/elo-world

2) https://www.kaggle.com/samaxtech/eda-clean-feng-lgbm-xgboost-stacked-model

3) https://www.kaggle.com/youhanlee/hello-elo-ensemble-will-help-you

4) https://brunogomescoelho.github.io/kaggle/elo_merchant/

5) https://github.com/chandureddivari/kaggle

6) https://github.com/bangd/kaggle/

7) https://www.kaggle.com/juyeong1537/juyeong-merchant-eda

8) https://github.com/bestpredicts/ELO/blob/master/zxs/pre_0214.ipynb

9) https://www.kaggle.com/sudalairajkumar/simple-exploration-notebook-elo

2) Dataset Overview and Imports

2.1) Dataset Description:

The datasets are largely anonymized, and the meaning of the features are not elaborated. External data are allowed File descriptions

train.csv - the training set

test.csv - the test set

historical_transactions.csv - up to 3 months' worth of historical transactions for each card_id

merchants.csv - additional information about all merchants / merchant_ids in the dataset.

new_merchant_transactions.csv - two months' worth of data for each card_id containing ALL purchases that card_id made at merchant_ids that were not visited in the historical data.

sample_submission.csv - a sample submission file in the correct format - contains all card_ids you are expected to predict for.

Data fields Data field descriptions are provided in Data Dictionary.xlsx.

2.2) Imports and Necessary Functions

In [1]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=94731898
9803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%
3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.co
m%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fww
w.googleapis.com%2fauth%2fdrive.photos.readonly%20https%3a%2f%2fwww.googleapis.com%2fauth
%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/drive
```

In [2]:

```
import numpy as np
import pandas as pd
import os
import gc
import matplotlib.pylab as plt
import seaborn as sns
import warnings
import datetime
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.preprocessing import LabelEncoder


warnings.filterwarnings('ignore')
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: FutureWarning: p
andas.util.testing is deprecated. Use the functions in the public API at pandas.testing i
nstead.
  import pandas.util.testing as tm
```

In [3]:

```
def reduce_mem_usage(df, verbose=True):
  #paste the kaggle kernel link
  '''
  The data size is too big to get rid of memory error this method will reduce memory
  usage by changing types. It does the following
  Load objects as categories
  Binary values are switched to int8
  Binary values with missing values are switched to float16
  64 bits encoding are all switched to 32 or 16bits if possible.
  Parameters
  ---------------
  df - DataFrame whose size to be reduced
  ---------------
  '''

  numerics = ['int16', 'int32', 'int64', 'float16', 'float32', 'float64']
  start_mem = df.memory_usage().sum() / 1024**2
  for col in df.columns:
      col_type = df[col].dtypes
      if col_type in numerics:
          c_min = df[col].min()
          c_max = df[col].max()
          if str(col_type)[:3] == 'int':
              if c_min > np.iinfo(np.int8).min and c_max < np.iinfo(np.int8).max:
                  df[col] = df[col].astype(np.int8)
              elif c_min > np.iinfo(np.int16).min and c_max < np.iinfo(np.int16).max:
                  df[col] = df[col].astype(np.int16)
              elif c_min > np.iinfo(np.int32).min and c_max < np.iinfo(np.int32).max:
                  df[col] = df[col].astype(np.int32)
              elif c_min > np.iinfo(np.int64).min and c_max < np.iinfo(np.int64).max:
                  df[col] = df[col].astype(np.int64)
          else:
              if c_min > np.finfo(np.float16).min and c_max < np.finfo(np.float16).max:
                  df[col] = df[col].astype(np.float16)
              elif c_min > np.finfo(np.float32).min and c_max < np.finfo(np.float32).max
:
                  df[col] = df[col].astype(np.float32)
              else:
                  df[col] = df[col].astype(np.float64)
  end_mem = df.memory_usage().sum() / 1024**2
  if verbose: print('Mem. usage decreased to {:5.2f} Mb ({:.1f}% reduction)'.format(end_
mem, 100 * (start_mem - end_mem) / start_mem))
  return df
```

In [4]:

```
def lab_enc(df, cols, prefix=''):
  '''
```

```
    label encode the values in the specified columns and
    return the data frame
    Parameters
    --------------------
    df   - Original DataFrame
    cols - label encode the specified columns
    --------------------
    '''

    lbl_enc = LabelEncoder()
    for col in cols:
      df[col] = lbl_enc.fit_transform(df[col].astype(str))
      np.save('{}_{}_enc.npy'.format(prefix, col), lbl_enc.classes_)
    return df
```

In [5]:

```
def get_basic_time_feat(df, grpby, col, s):
    '''
    create basic time feats like differece in minute, days etcetera
    and return the dataframe.

    Parameters
    ---------------------
    df       - Features will be created
    grpby    - group the DF based on this value
    col      - column where the operations will be performed
    s        - shift value
    ---------------------
    '''

    df = df.sort_values(col)
    for i in range(s):
      df['prev_{}_'.format(i+1)+col] = df.groupby([grpby])[col].shift(i+1)
      df['purchase_date_diff_{}_days'.format(i+1)] = (df[col] - df['prev_{}_'.format(i+1)+
col]).dt.days.values
      df['purchase_date_diff_{}_seconds'.format(i+1)] = df['purchase_date_diff_{}_days'.fo
rmat(i+1)].values * 24 * 3600
      df['purchase_date_diff_{}_seconds'.format(i+1)] += (df[col] - df['prev_{}_'.format(i
+1)+col]).dt.seconds.values
      df['purchase_date_diff_{}_hours'.format(i+1)] = df.iloc[:, -1].values // 3600

    return df
```

In [6]:

```
def s_agg(new_df, df, op, prefix, grpby, col):
    '''
    takes the data frame as input and return the dataframe with the aggregate operations pe
rformed.

    Parameters
    ----------------------------
    new_df   - DF with new features added
    df       - original DF
    op       - statistical operations like min, max, mean etc.
    prefix   - prefix for the feature name
    grpby    - based on which column to group by
    col      - operations will be performed on this column
    ----------------------------
    '''

    for o in op:
      new_df[prefix+col+'_{}'.format(o)] = df.groupby([grpby])[col].agg([o]).values
    return new_df
```

In [7]:

```
def find_single_val(new_df, df, col, grpby, op, name='',  prefix='', use_col=False):
    '''
    find a value like min, max, mean in the specified column and return the DF
```

```
    Parameters
    ------------------
    new_df    - features will be added to this DF
    df        - original DF from which the features will be created
    col       - operations will be performed on this column
    grpby     - based on this column we'll to group by
    name      - name for the new features created
    op        - statistical operations to be performed
    prefix    - added to the name of the feature -- default value empty
    use_col   - if set True then the original column name will be uesd to name the new featu
re -- default value False
    ------------------
    '''

    if use_col:
      for c in col:
        for o in op:
          if o is 'min':
            new_df[prefix+'_'+c+'_'+'{}'.format(o)] = df.groupby([grpby])[c].min().values
          elif o is 'max':
            new_df[prefix+'_'+c+'_'+'{}'.format(o)] = df.groupby([grpby])[c].max().values
          elif o is 'mean':
            new_df[prefix+'_'+c+'_'+'{}'.format(o)] = df.groupby([grpby])[c].mean().values
          elif o is 'sum':
            new_df[prefix+'_'+c+'_'+'{}'.format(o)] = df.groupby([grpby])[c].sum().values
          elif o is 'nunique':
            new_df[prefix+'_'+c+'_'+'{}'.format(o)] = df.groupby([grpby])[c].nunique().val
ues
          elif o is 'std':
            new_df[prefix+'_'+c+'_'+'{}'.format(o)] = df.groupby([grpby])[c].std().values
          elif o is 'count':
            new_df[prefix+'_'+c+'_'+'{}'.format(o)] = df.groupby([grpby])[c].count().value
s

    else:
      for c in col:
        for o in op:
          if o is 'min':
            new_df[name] = df.groupby([grpby])[c].min().values
          elif o is 'max':
            new_df[name] = df.groupby([grpby])[c].max().values
          elif o is 'mean':
            new_df[name] = df.groupby([grpby])[c].mean().values
          elif o is 'sum':
            new_df[name] = df.groupby([grpby])[c].sum().values
          elif o is 'nunique':
            new_df[name] = df.groupby([grpby])[c].nunique().values
          elif o is 'std':
            new_df[name] = df.groupby([grpby])[c].std().values
          elif o is 'count':
            new_df[name] = df.groupby([grpby])[c].count().values

    return new_df
```

In [8]:

```
def get_monthlag_stat(new_df, df, grpby, op, col, name, prefix=''):

    '''
    group by the the specified column and find the count or sum depending on the input.
    Then perform basic operations like std, min, max etcetera
    parameters
    ----------------------------
    new_df - new features will be added to this DF
    df     - original DF
    grpby  - column using which we will group the data by
    col    - operations will be performed on this column
    name   - name for this columnn
    prefix - prefix to the column name
    ----------------------------
```

```
    '''
    if op == 'sum':
      tmp = df.groupby(grpby)[col].sum().unstack()
      new_df[prefix+grpby[1]+'_'+name[0]] = tmp.reset_index().iloc[:, -1].values
      new_df[prefix+grpby[1]+'_'+name[1]] = tmp.reset_index().iloc[:, -2].values

    if op == 'count':
      tmp = df.groupby(grpby)[col].count().unstack()
      # check if there is any null value and fill it with 0
      # for the sum we are not performing any null value imputation
      # as we are directly using the value. However, here we are performing operations like
      # min, max, std etcetera so we are imputing the null values.
      if tmp.isna().sum().any() > 0:
        tmp = tmp.fillna(0.0)
      new_df[prefix+grpby[1]+'_'+name[0]] = tmp.reset_index().iloc[:, 1:].std(axis=1).valu
es
      new_df[prefix+grpby[1]+'_'+name[1]] = tmp.reset_index().iloc[:, 1:].max(axis=1).valu
es
    return new_df
```

In [9]:

```
#https://www.kaggle.com/fabiendaniel/elo-world?scriptVersionId=8335387
def successive_aggregates(df, field1, field2):
    '''
    what this function does is that it group the data twice and find
    basic aggregate values.
    First it will goup by card_id and all the specified column one by one.
    Then it will find the agg values like mean, min, max and std
    for the purchase amount for each group.
    Parameters
    -------------------
    df      - original DataFrame
    field1  - first groupby along with card_id
    field2  - second grouby along with card_id
    -------------------
    '''

    t = df.groupby(['card_id', field1])[field2].mean()
    u = pd.DataFrame(t).reset_index().groupby('card_id')[field2].agg(['mean', 'min', 'ma
x', 'std'])
    u.columns = ['new_transac_' + field1 + '_' + field2 + '_' + col for col in u.columns
.values]
    u.reset_index(inplace=True)
    return u
```

In [10]:

```
def get_influential(df, col_name, date):
  '''
  This function return whether a purchase is influential or not.
  A purchase is considered influential if it is made 100 days before a festival.
  If it is not influential it will give a value 0 else the actual value.
  Parameters
  ---------------------------------
  df       - Dataframe where the operations will be performed
  col_name - name of the new feature
  date     - on which date the holiday is occuring
  ---------------------------------
  '''

  df[col_name] = (pd.to_datetime(date) - pd.to_datetime(df['purchase_date'])).dt.days
  df[col_name] = df[col_name].apply(lambda x: x if x > 0 and x < 100 else 0)
  return df
```

3) Exploratory Data Analysis

Importing the data

In [ ]:

```
train        = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/train.csv', pars
e_dates=["first_active_month"])
sample       = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/sample_submissio
n.csv')
test         = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/test.csv', parse
_dates=["first_active_month"])
ht           = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/historical_trans
actions.csv',parse_dates=['purchase_date'])
merchant     = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/merchants.csv')
new_merchant = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/new_merchant_tran
sactions.csv',parse_dates=["purchase_date"])

train        = reduce_mem_usage(train)
test         = reduce_mem_usage(test)
ht           = reduce_mem_usage(ht)
merchant     = reduce_mem_usage(merchant)
new_merchant = reduce_mem_usage(new_merchant)
```

In [ ]:

```
train        = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/train.csv', pars
e_dates=["first_active_month"])
test         = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/test.csv', parse
_dates=["first_active_month"])
```

In [ ]:

```
new_merchant = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/new_merchant_tran
sactions.csv',parse_dates=["purchase_date"])
```

In [ ]:

```
new_merchant['purchase_date'].max()
```

Out[ ]:

```
Timestamp('2018-04-30 23:59:59')
```

3.1) Exploring train and test data

3.1.1) Overview

In [ ]:

```
print('The shape of train is:', train.shape)
print('The shape of test is:', test.shape)
```

```
The shape of train is: (201917, 6)
The shape of test is: (123623, 5)
```

In [ ]:

```
train.head()
```

Out[ ]:

	first_active_month	card_id	feature_1	feature_2	feature_3	target
0	2017-06-01	C_ID_92a2005557	5	2	1	-0.820312
1	2017-01-01	C_ID_3d0044924f	4	1	0	0.392822
2	2016-08-01	C_ID_d639edf6cd	2	2	0	0.687988
3	2017-09-01	C_ID_186d6a6901	4	3	0	0.142456
4	2017-11-01	C_ID_cdbd2c0db2	1	3	0	-0.159790

In [ ]:

```
test.head()
```

Out[ ]:

	first_active_month	card_id	feature_1	feature_2	feature_3
0	2017-04-01	C_ID_0ab67a22ab	3	3	1
1	2017-01-01	C_ID_130fd0cbdd	2	3	0
2	2017-08-01	C_ID_b709037bc5	5	1	1
3	2017-12-01	C_ID_d27d835a9f	2	1	0
4	2015-12-01	C_ID_2b5e3df5c2	5	1	1

The target in the train set is the value we have to predict

3.1.2) Let's check for

1. **Overlapping cards in train and test set**
2. **Duplicates cards in train and test set**
3. **Any null values**

In [ ]:

```
train['card_id'].isin(test['card_id']).sum()
```

Out[ ]:

```
0
```

In [ ]:

```
print('Duplicates in train:',train.duplicated(['card_id']).sum())
print('Duplicates in test:',test.duplicated(['card_id']).sum())
```

```
Duplicates in train: 0
Duplicates in test: 0
```

It looks like there aren't any overlapping or duplicates in both train and test

In [ ]:

```
train.isnull().sum()
```

Out[ ]:

```
first_active_month    0
card_id               0
feature_1             0
feature_2             0
feature_3             0
target                0
dtype: int64
```

In [ ]:

```
test.isnull().sum()
```

Out[ ]:

```
first_active_month    1
card_id               0
feature_1             0
feature_2             0
feature_3             0
dtype: int64
```

Train set doesn't have any null values. However the test set has one null value in first_active_month. We need to impute the null value in the test set. We can impute this value with the mode.

3.1.3) Let's investigate on the anonymized featuers in the train set

In [ ]:

```
print(train['feature_1'].unique())
print(train['feature_2'].unique())
print(train['feature_3'].unique())
```

```
[5 4 2 1 3]
[2 1 3]
[1 0]
```

- **All the anonymized features are categorical**
- **feature_1 can take five values[1,2,3,4,5]**
- **feature_2 can take three values[1,2,3]**
- **feature_1 can take two values[0,1]**

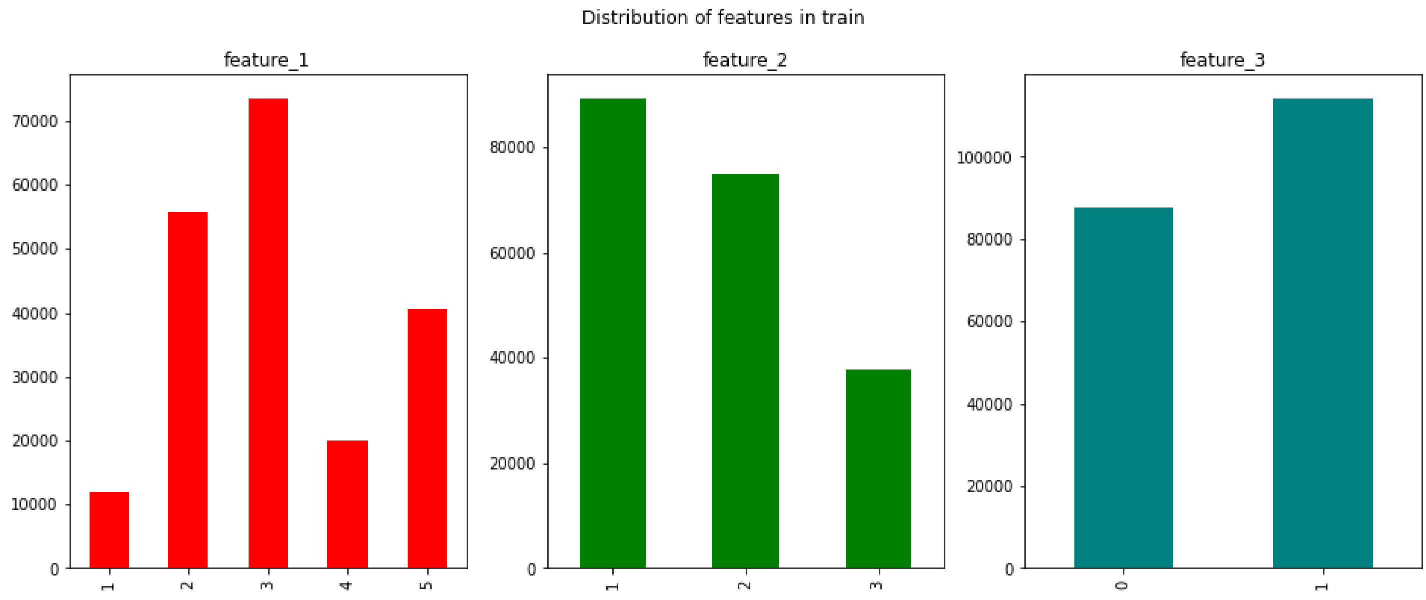
checking the distribution of train and test set

In [ ]:

```
features = ['feature_1','feature_2','feature_3']
```

In [ ]:

```
fig, ax = plt.subplots(1, 3, figsize = (16, 6))
plt.suptitle('Distribution of features in train')
features = ['feature_1','feature_2','feature_3']
colors = ['red', 'green', 'teal']

for idx, feature in enumerate(features):
  train[feature].value_counts().sort_index().plot(kind='bar', ax=ax[idx], color=colors[i
dx], title=feature)
```



All the anonymized features are categorical

feature_1 can take five values[1,2,3,4,5]

feature_2 can take three values[1,2,3]

feature_1 can take two values[0,1]

Since all these features are categorical they should be one hot encoded.

In [ ]:

```
#https://www.kaggle.com/batalov/making-sense-of-elo-data-eda

plt.figure(figsize=[14,6])
plt.suptitle('Feature distributions in train and test', fontsize=12, y=1.1)
for num, col in enumerate(['feature_1', 'feature_2', 'feature_3']):
    plt.subplot(2, 3, num+1)
    if col is not 'target':
        v_c = train[col].value_counts() / train.shape[0]
        plt.bar(v_c.index, v_c, label=('train'), align='edge', width=-0.3, edgecolor=[0.
1]*3, color='red')
        v_c = test[col].value_counts() / test.shape[0]
        plt.bar(v_c.index, v_c, label=('test'), align='edge', width=0.3, edgecolor=[0.1]
*3,color='teal')
        plt.title(col)
        plt.legend()
    plt.tight_layout()
plt.tight_layout()
plt.show()
```

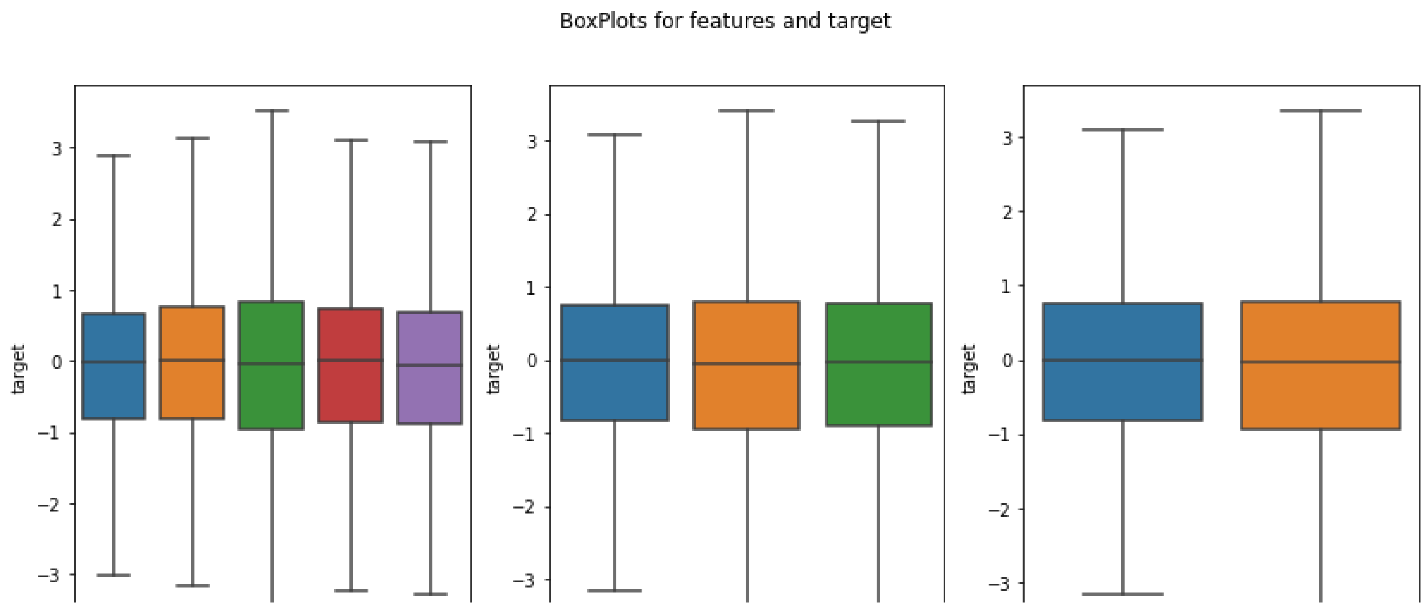
There are no differences in the train and test distribution. Both the distribution looks identical

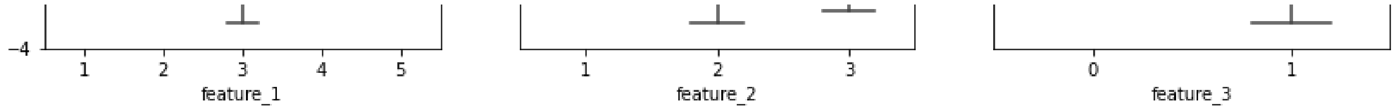
So, I guess we don't have to do time based splitting since the distribution of the train and test set are same

3.1.4) Let's plot the features against the target variable

In [ ]:

```
fig, ax = plt.subplots(1, 3, figsize = (14, 6))
plt.suptitle('BoxPlots for features and target')
for idx, feature in enumerate(features):
  sns.boxplot(x=feature, y="target", data=train, ax=ax[idx], showfliers=False)
```





The distribution of the features w.r.t the target all looks same. So these features may not be helpful in predicting the target.

We might need to engineer new features based off of these features.

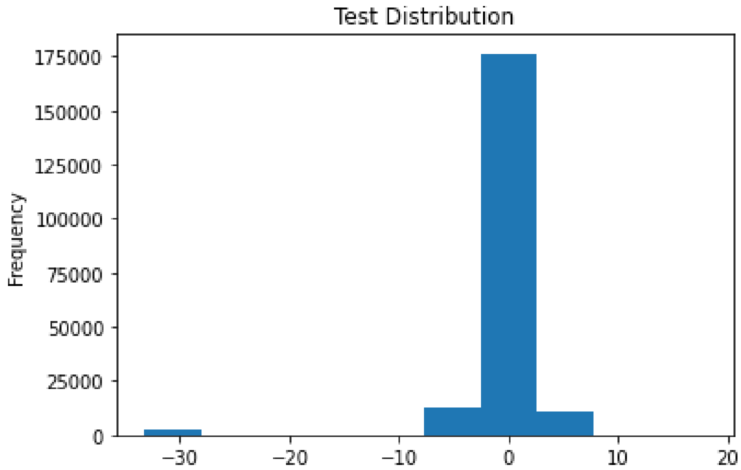
3.1.5) Investigating Target Variable

In [ ]:

```
train['target'].plot(kind='hist',title='Test Distribution')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe643415eb8>
```



Looks interesting! The values are normally distributed around mean 0. However if you take a closer look there are some outliers at -30.

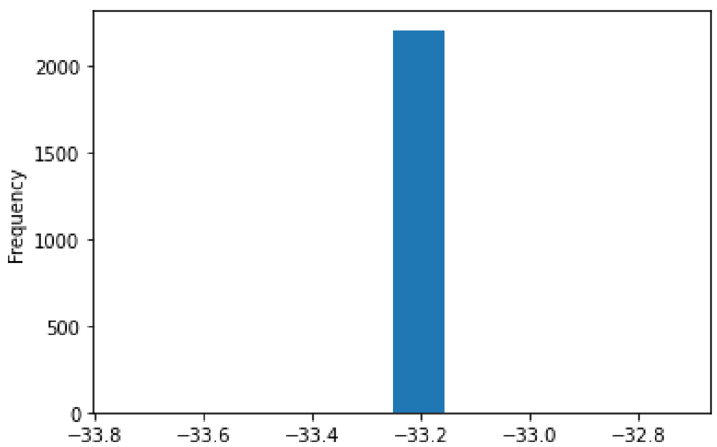
Let's zoom in to the values less than -20.

In [ ]:

```
train['target'][train['target']< -20].plot.hist()
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe6563df8d0>
```



There are quite a lot of them. Let's find the exact number.

In [ ]:

```
train['target'][train['target']<-30].count()
```

Out[ ]:

2207

There are a total of 2207 outliers.

Since there are these many outliers we need to find a way to handle them.

Apparantly dropping these outliers is not an option as these proportion of outliers could be in the test set also.

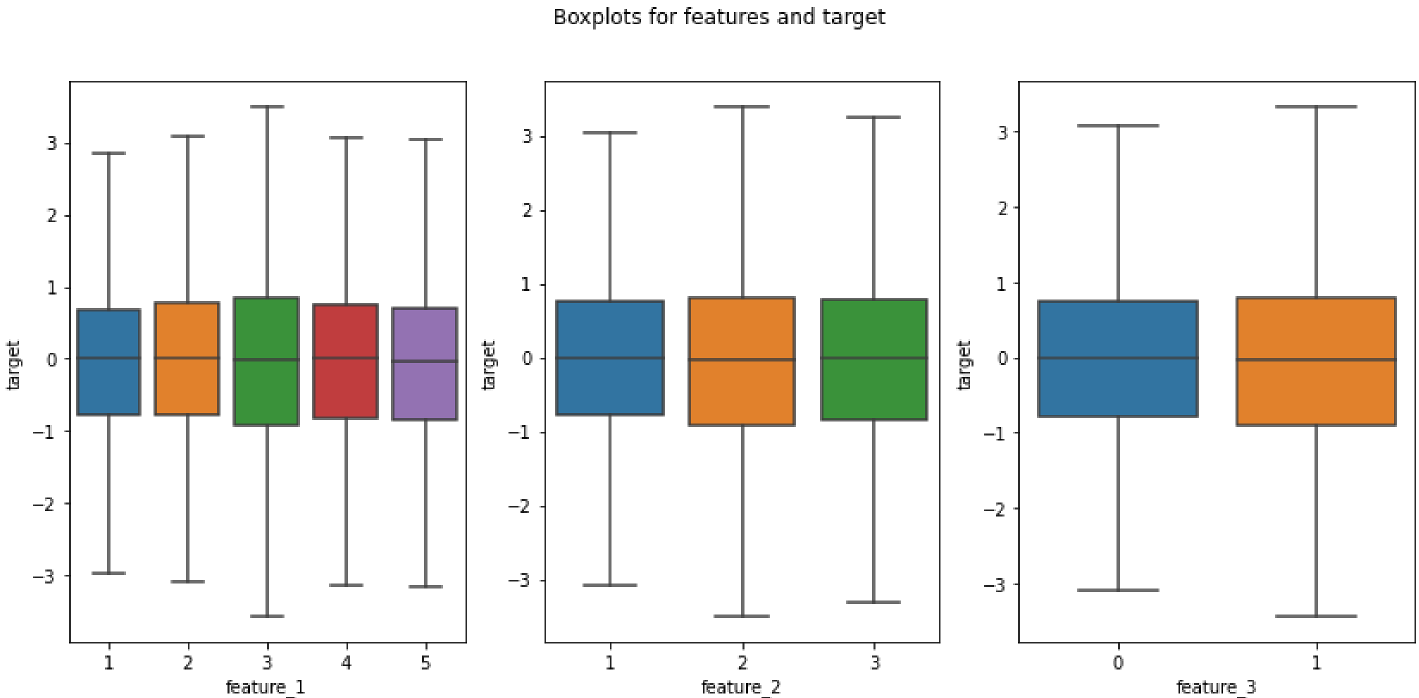
One way is that we could crete a new column and mark them as outliers or not and see whether the model could make any sense out of it.

Since there are outliers let's once again plot the features against the target variable however

this time without the outliers

In [ ]:

```
'''
we are terming anything less than -30 in the target variable as outliers. So we are creat
ing a new dataframe train_no_out
which contains rows without outliers.
Then we can plot a boxplot in this new dataframe to see whether the features can be usefu
l.
'''

train_no_out = train.loc[train['target'] > -30]

fig, ax = plt.subplots(1, 3, figsize = (14, 6))
plt.suptitle('Boxplots for features and target')
for idx, feature in enumerate(features):
  sns.boxplot(x=feature, y="target", data=train_no_out, ax=ax[idx], showfliers=False)
```



Even after removing the outliers the features doesn't seem to separte the target classes. They still they look the same.

So we definitely need to create new features.

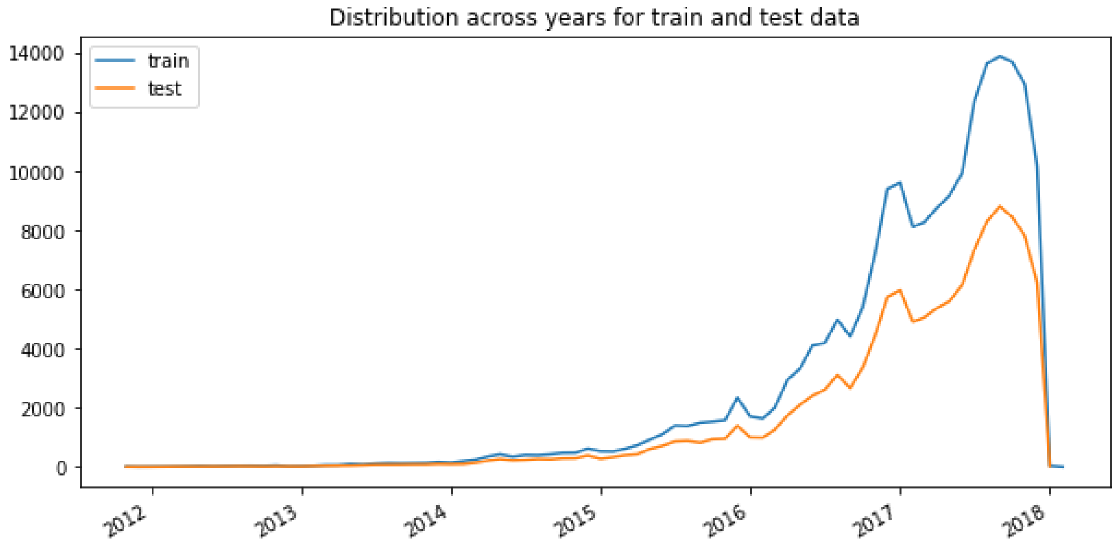
3.1.6) first_active_month

In [ ]:

```
#https://www.kaggle.com/tminima/elo-eda
'''
getting the number of transactions for every year both train and test separately
so that we can plot the distribution of values for train and test data.
'''
labels=['train','test']

temp = train.first_active_month.value_counts().sort_index()
temp1 = test.first_active_month.value_counts().sort_index()

ax = temp.plot(figsize=(10, 5))
ax = temp1.plot(figsize=(10,5))

ax.set_title("Distribution across years for train and test data")
ax.legend(labels)
```

Out[ ]:

```
<matplotlib.legend.Legend at 0x7f89e27812e8>
```



This feature the first active month tells us when the first purchase was made through out years.

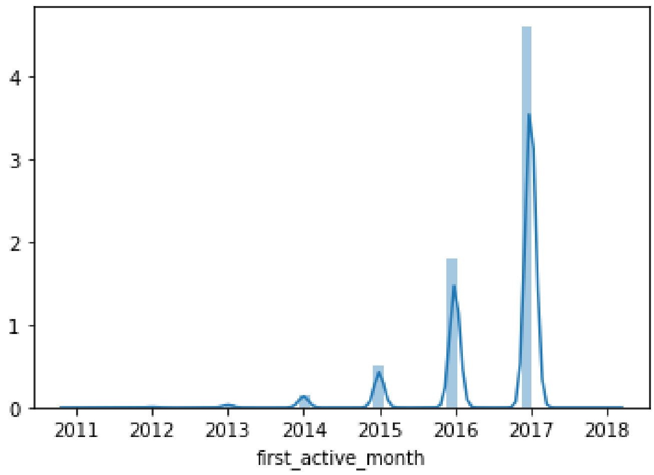
Here the train and test distribution looks same. Since they look similar except for the frequency, we don't have to do time based splitting

In [ ]:

```
year = pd.DatetimeIndex(train['first_active_month']).year
month = pd.DatetimeIndex(train['first_active_month']).month
sns.distplot(year)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe61f1d1588>
```



The above plot gives us the number of cards which made their first purchase each year.

In the given data the maximum number of first purchase is made in the year 2017. There is a sharp decline in the number of cards first active in the year 2018. This could be because we have fewer data for that year.

Let's check that quickly

In [ ]:

```
year.value_counts()
```

Out[ ]:

```
2017    130519
2016     51277
2015     14142
2014      4523
2013      1129
2012       282
2018        35
2011        10
Name: first_active_month, dtype: int64
```

As you can see there are only 35 entries for the year 2018. So that is the reason for the sharp decline in the graph.

Let's proceed further and visualize this feature for each month across years and see whether we can get any useful information from it.

Let's visualize this feature across months for all the years

In [ ]:

```
'''
create a new dataframe t and get only the column
frist_active_month from the train set so we can separter the
year and month and visualize this feature monthwise for every year.
'''
t = train[['first_active_month']]
t['year'] = pd.DatetimeIndex(t['first_active_month']).year
t['month'] = pd.DatetimeIndex(t['first_active_month']).month_name()
t
```

Out[ ]:

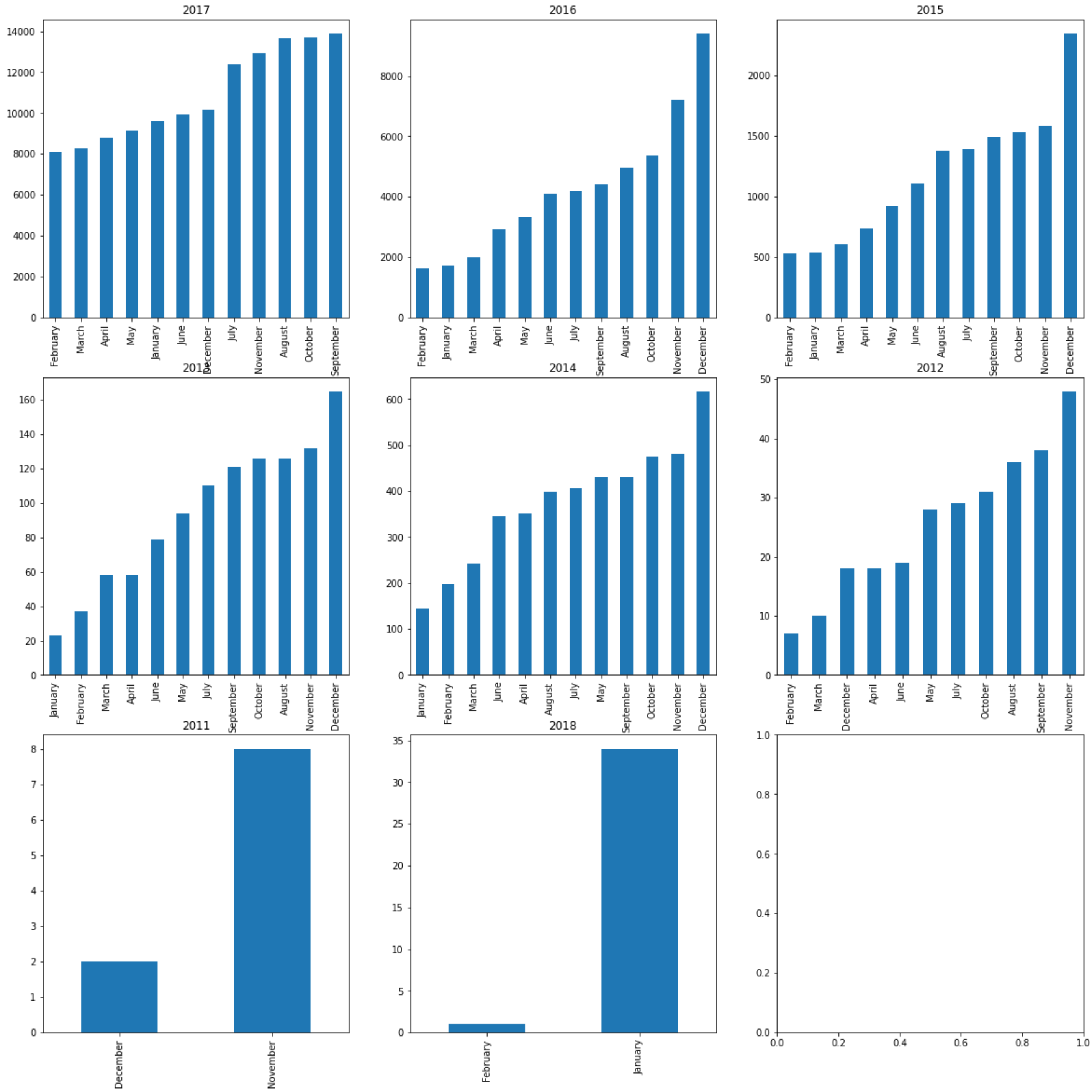
	first_active_month	year	month
0	2017-06-01	2017	June
1	2017-01-01	2017	January
2	2016-08-01	2016	August
3	2017-09-01	2017	September
4	2017-11-01	2017	November
...	...	...	...
201912	2017-09-01	2017	September
201913	2015-10-01	2015	October
201914	2017-08-01	2017	August
201915	2016-07-01	2016	July
201916	2017-07-01	2017	July

201917 rows × 3 columns

In [ ]:

```
'''
```

```
find the number of unique values in the year.
Now group the dataframe using year and find the value count of each month for
every year. So that we can plot the purchases for every month year wise.
'''
%matplotlib inline
fig, ax = plt.subplots(3, 3, figsize = (20, 20))
ax = ax.ravel()

y = t['year'].unique()

for idx, year in enumerate(y):
 t.groupby('year').get_group(year)['month'].value_counts().sort_values().plot(kind='bar'
,ax=ax[idx], title=year)
```



In almost all the year the first_purchase was made in the month december. May be due christmas and new year the credit card company might rolled out some new offers.

For year 2011 and 2018 only two months data are available

Other than these there is not much we can use for further analysis

Correlation between the features in train

In [ ]:

```
cols = ['feature_1','feature_2','feature_3']

n = train[cols]

vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(n.iloc[:,:].values, i) for i in range(n.s
hape[1])]
vif["features"] = n.columns
vif
```

Out[ ]:

	VIF Factor	features
0	5.751775	feature_1
1	3.388442	feature_2
2	3.390544	feature_3

The VIF values for all the three features are well under 10. So there is no problem of multicollinearity in the train data.

3.2) Explore Historical Transaction

3.2.1) Overview

In [ ]:

```
print('The shape of the data is:', ht.shape)
```

```
The shape of the data is: (29112361, 14)
```

In [ ]:

```
ht.head()
```

Out[ ]:

	authorized_flag	card_id	city_id	category_1	installments	category_3	merchant_category_id	merchant_id
0	Y	C_ID_4e6213e9bc	88	N	0	A	80	M_ID_e020e9b302
1	Y	C_ID_4e6213e9bc	88	N	0	A	367	M_ID_86ec983688
2	Y	C_ID_4e6213e9bc	88	N	0	A	80	M_ID_979ed661fc
3	Y	C_ID_4e6213e9bc	88	N	0	A	560	M_ID_e6d5ae8ea6
4	Y	C_ID_4e6213e9bc	88	N	0	A	80	M_ID_e020e9b302

In [ ]:

```
ht.isnull().sum()
```

Out[ ]:

```
authorized_flag         0
card_id                 0
city_id                 0
category_1              0
installments            0
```

```
category_3               178159
merchant_category_id          0
merchant_id              138481
month_lag                     0
purchase_amount               0
purchase_date                 0
category_2              2652864
state_id                      0
subsector_id                  0
dtype: int64
```

There are null values in the features category_3, merchant_id and category_2.

Since category_2 and 3 are categorical we could lable encode them and create a new category for the missing values.

3.2.2) Let's calculate the percentage of missing values

In [ ]:

```
print('Percentage of missing values in merchant_id:',ht['merchant_id'].isnull().sum() / l
en(ht)*100,'%')
print('Percentage of missing values in category_2:',ht['category_2'].isnull().sum() / len
(ht)*100,'%')
print('Percentage of missing values in category_3:',ht['category_3'].isnull().sum() / len
(ht)*100,'%')
```

```
Percentage of missing values in merchant_id: 0.4756776683278969 %
Percentage of missing values in category_2: 9.1125003568072 %
Percentage of missing values in category_3: 0.6119702898710276 %
```

Since there are more than 9% of missing values in category_2 we cannot simply drop them. We need find a way to impute them. The other two features merchant_id and category_3 has fewer than 1% of missing values so we can safely drop them.

In [ ]:

```
print('Percentage missing values in the whole dataset:', ht.isnull().sum().sum() / len(h
t) * 100, '%')
```

```
Percentage missing values in the whole dataset: 10.200148315006125 %
```

3.2.3) Exploring Authorized Flag

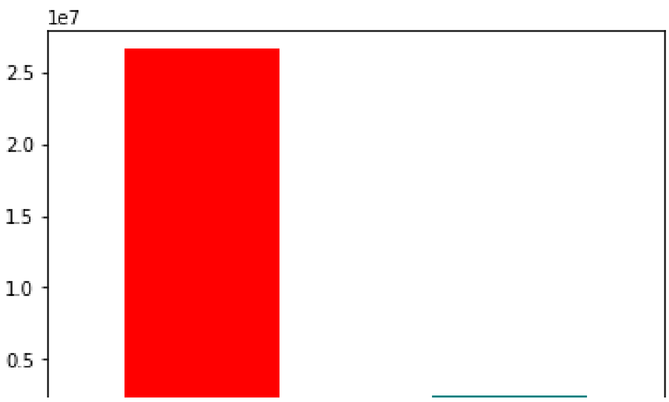
Let's find out how many transactions are authorized.

In [ ]:

```
ht['authorized_flag'].value_counts().plot(kind='bar',color=['red','teal'])
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3664a32278>
```



There are some notable amount of unauthorized transactions. Here Y means authorizedd and N means unauthorized.

Let's find the percentage of unauthorized transactions.

In [ ]:

```
print('Number of authorized transactions:',ht['authorized_flag'].value_counts()['Y'])
print('Number of un-authorized transactions:',ht['authorized_flag'].value_counts()['N'])
print('Percentage of authorized transactions:', round(ht['authorized_flag'].value_counts
()['Y'] / len(ht) * 100,2),'%')
print('Percentage of un-authorized transactions:', round(ht['authorized_flag'].value_coun
ts()['N'] / len(ht) * 100,2),'%')
```

```
Number of authorized transactions: 26595452
Number of un-authorized transactions: 2516909
Percentage of authorized transactions: 91.35 %
Percentage of un-authorized transactions: 8.65 %
```

About 9% of the transactions are unauthorized. So these can be a good feature to predict the loyalty score for the customers.

While doing feature engineering we could create features separately for authorized and unauthorized.

In [ ]:

```
# mapping Y to 1 and N to 0
ht['authorized_flag'] = ht['authorized_flag'].apply(lambda x: 1 if x == 'Y' else 0)
```

In [ ]:

```
a = ht.groupby('card_id')['authorized_flag'].value_counts().sort_values(ascending=False)
a.describe()
```

Out[ ]:

```
count    601241.000000
mean         48.420452
std          82.286143
min           1.000000
25%           6.000000
50%          18.000000
75%          55.000000
max        4122.000000
Name: authorized_flag, dtype: float64
```

In [ ]:

```
a
```

Out[ ]:

```
card_id          authorized_flag
C_ID_3d3dfdc692  1                  4122
C_ID_0cd2ce025c  1                  2537
C_ID_cc3d4cd4e3  1                  2027
C_ID_5ccc07beb9  1                  1963
C_ID_9f81506906  1                  1592
                                    ... 
C_ID_44f218940c  0                     1
C_ID_30a18c60cb  0                     1
C_ID_44f23504f3  0                     1
C_ID_7b965f1865  0                     1
C_ID_7521e9958a  0                     1
Name: authorized_flag, Length: 601241, dtype: int64
```

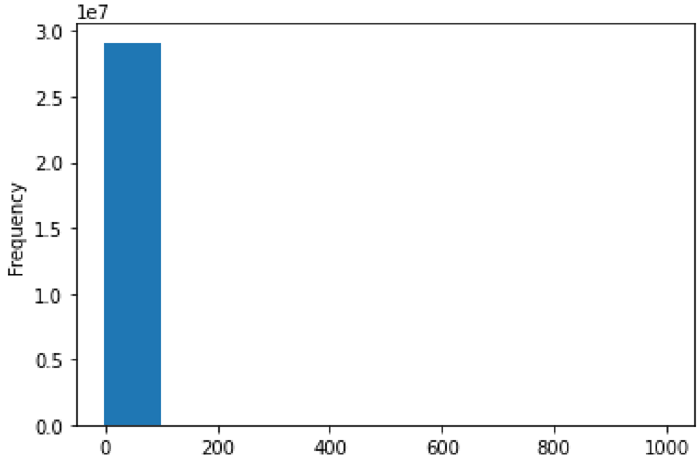
3.2.4) Installment

In [ ]:

```
ht['installments'].plot(kind='hist')
```

Out[ ]:

```
&lt;matplotlib.axes._subplots.AxesSubplot at 0x7fe648a608d0&gt;
```



Looks strange. The value of x that is the installments ranges from -1 to 1000. The installment cannot take values like 1000. That is not sensible.

So, let's check the value counts

In [ ]:

```
ht['installments'].value_counts()
```

Out[ ]:

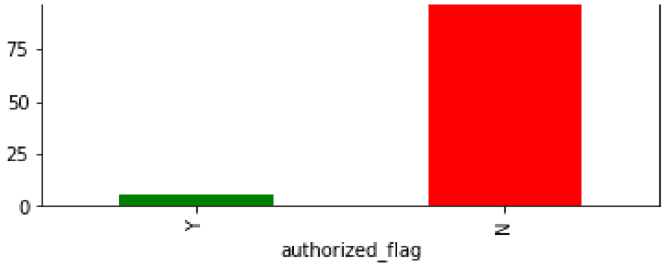
```
 0      15411747
 1      11677522
 2        666416
 3        538207
 4        179525
-1        178159
 6        132634
 10       118827
 5        116090
 12        55064
 8         20474
 7         10906
 9          5772
 11          830
 999         188
Name: installments, dtype: int64
```

Looks like the values -1 and 999 could be used to fill missing values. Or these values could be to denote a fraud transaction. Especially the value 999 could be to denote the unauthorized transactions.

Let's see how many transactions are authorized if the installment value is 999

In [ ]:

```
f = ht.groupby(['installments'])['authorized_flag'].value_counts()[999].sort_values(asce
nding=True).plot(kind='bar',color=['green','red'])
```



In [ ]:

```
ht.groupby(['installments'])['authorized_flag'].value_counts()[999]
```

Out[ ]:

```
authorized_flag
N    182
Y      6
Name: authorized_flag, dtype: int64
```

As we can see from the above plot there are only 3% of authorized transactions. So this value could be because the transaction is fraud. This could serve as a good feature

In [ ]:

```
f.head
```

Out[ ]:

```
<bound method NDFrame.head of installments  authorized_flag
-1            Y                   157794
              N                    20365
 0            Y                 14302589
              N                  1109158
 1            Y                 10591787
              N                  1085735
 2            Y                   589125
              N                    77291
 3            Y                   464071
              N                    74136
 4            Y                   147193
              N                    32332
 5            Y                    93938
              N                    22152
 6            Y                   103419
              N                    29215
 7            Y                     7560
              N                     3346
 8            Y                    14177
              N                     6297
 9            Y                     3831
              N                     1941
 10           Y                    83419
              N                    35408
 11           Y                      548
              N                      282
 12           Y                    35995
              N                    19069
 999          N                      182
              Y                        6
Name: authorized_flag, dtype: int64>
```

If we take a look at the value 999 from the above table out of all the 188 entries 182 almost 97% of transactions were not authorized. So it could mean fraud transaction. Let's remove it and plot the values.

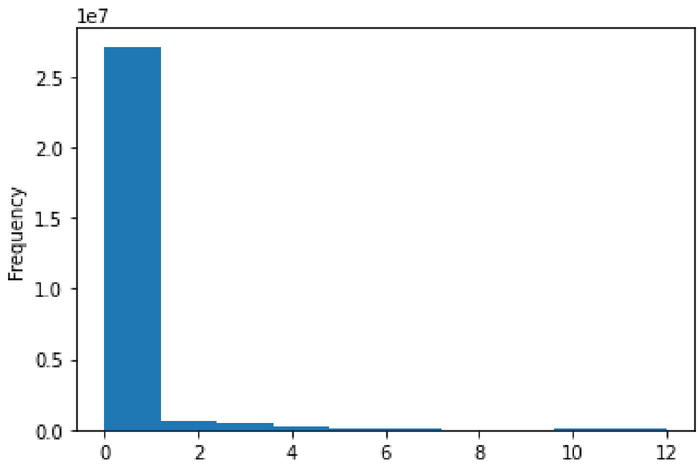
Let's plot the histogram with range(0,12) removing -1 and 999

In [ ]:

```
ht['installments'].plot(kind='hist', range=[0,12])
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe67948da58>
```



In [ ]:

```
ht['installments'].value_counts()
```

Out[ ]:

```
 0      15411747
 1      11677522
 2        666416
 3        538207
 4        179525
-1        178159
 6        132634
 10       118827
 5        116090
 12        55064
 8         20474
 7         10906
 9          5772
 11          830
 999         188
Name: installments, dtype: int64
```

Large number of the values are either 0 or 1. In most the cases there are no installments or an installment of 1 month.

In [ ]:

```
# getting MEMORY ERROR DON'T RUN IT
cols = ['authorized_flag','city_id','category_1','installments','category_3','merchant_c
ategory_id','month_lag','purchase_amount','category_2','state_id', 'subsector_id']
d = {'A':1,'B':2,'C':3}
x = {'Y':1,'N':0}

n = ht[cols]
n['category_3'] = n['category_3'].map(d)
n['category_1'] = n['category_1'].map(x)
n['authorized_flag'] = n['authorized_flag'].map(x)

n = n.dropna()

vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(n.iloc[:,:].values, i) for i in range(n.s
hape[1])]
vif["features"] = n.columns
vif
```

3.2.5) Let's check the features category_1, category_2, category_3

In [ ]:

```
# convert the column type to category so we can find the correlation between them
# by default it is in object.
col = ['category_1', 'category_2', 'category_3']
for c in col:
  if c in ht.columns:
    ht[c] = ht[c].astype('category')
```

In [ ]:

```
#https://stackoverflow.com/questions/48035381/correlation-among-multiple-categorical-vari
ables-pandas
corr = ht[['category_1','category_2','category_3']].apply(lambda x : pd.factorize(x)[0])
.corr(method='pearson', min_periods=1)
```

In [ ]:

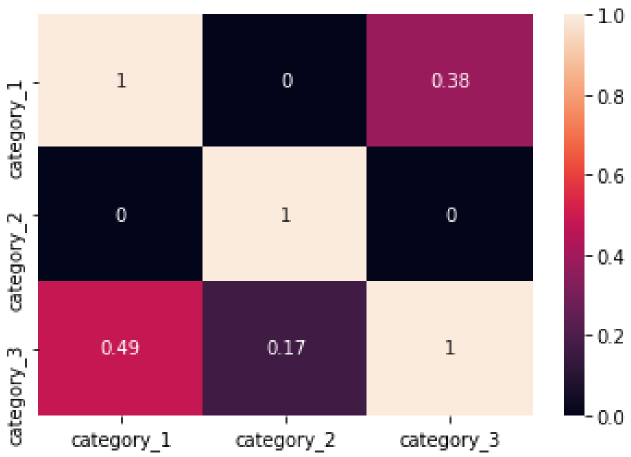
```
# normalizing the values so it between 0 and 1
corr_norm=np.round((corr-corr.min())/(corr.max()-corr.min()), 3)
```

In [ ]:

```
sns.heatmap(corr_norm,
        xticklabels=corr.columns,
        yticklabels=corr.columns, annot=True)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff3ae8b2390>
```



Looks like category3 and category1 are correlated. Since these features are anonyized we don't have a clear idea of what these features are?

So instead of dropping this strightaway we could train model with and without it and see the performance changes.

In [ ]:

```
ht['category_1'].value_counts()
```

Out[ ]:

```
N    27028332
Y     2084029
Name: category_1, dtype: int64
```

In [ ]:

```
ht['category_2'].value_counts()
```

Out[ ]:

```
1.0    15177199
3.0     3911795
```

```
5.0     3725915
4.0     2618053
2.0     1026535
Name: category_2, dtype: int64
```

In [ ]:

```
ht['category_3'].value_counts()
```

Out[ ]:

```
A    15411747
B    11677522
C     1844933
Name: category_3, dtype: int64
```

The category_1 takes two values Y, N

The category_2 takes five values 1,2,3,4,5

The category_3 takes three values A,B,C

3.3) Merchant

3.3.1) Overview

In [ ]:

```
print('The shape of the data is:', merchant.shape)
```

```
The shape of the data is: (334696, 22)
```

In [ ]:

```
merchant.head()
```

Out[ ]:

	merchant_id	merchant_group_id	merchant_category_id	subsector_id	numerical_1	numerical_2	category_1	most_rec
0	M_ID_838061e48c	8353	792	9	-0.057465	-0.057465	N	
1	M_ID_9339d880ad	3184	840	20	-0.057465	-0.057465	N	
2	M_ID_e726bbae1e	447	690	1	-0.057465	-0.057465	N	
3	M_ID_a70e9c5f81	5026	792	9	-0.057465	-0.057465	Y	
4	M_ID_64456c37ce	2228	222	21	-0.057465	-0.057465	Y	

In [ ]:

```
merchant.isnull().sum()
```

Out[ ]:

```
merchant_id                     0
merchant_group_id               0
merchant_category_id            0
subsector_id                    0
numerical_1                     0
numerical_2                     0
category_1                      0
most_recent_sales_range         0
most_recent_purchases_range     0
avg_sales_lag3                 13
avg_purchases_lag3              0
active_months_lag3              0
avg_sales_lag6                 13
```

```
avg_purchases_lag6                0
active_months_lag6                0
avg_sales_lag12                  13
avg_purchases_lag12               0
active_months_lag12               0
category_4                        0
city_id                           0
state_id                          0
category_2                    11887
dtype: int64
```

There are few null values in lag 3,6 and 12 and around 11887 in category_2..

Other than category_2 there are very few missing values. So we could drop them.

In [ ]:

```
print('Percentage of missing values in avg_sales_lag3:', merchant['avg_sales_lag3'].isnul
l().sum() / len(merchant['avg_sales_lag3']) * 100, '%')
print('Percentage of missing values in avg_sales_lag6:', merchant['avg_sales_lag6'].isnul
l().sum() / len(merchant['avg_sales_lag6'])* 100, '%')
print('Percentage of missing values in avg_sales_lag12:', merchant['avg_sales_lag12'].isn
ull().sum() / len(merchant['avg_sales_lag12'])* 100, '%')
print('Percentage of missing values in category_2:', merchant['category_2'].isnull().sum(
) / len(merchant['category_2'])* 100, '%')
```

```
Percentage of missing values in avg_sales_lag3: 0.0038841217104476898 %
Percentage of missing values in avg_sales_lag6: 0.0038841217104476898 %
Percentage of missing values in avg_sales_lag12: 0.0038841217104476898 %
Percentage of missing values in category_2: 3.551581136314745 %
```

In [ ]:

```
print('Percentage of missing values in the dataset:', merchant.isnull().sum().sum() / len
(merchant) * 100, '%')
```

```
Percentage of missing values in the dataset: 3.5632335014460885 %
```

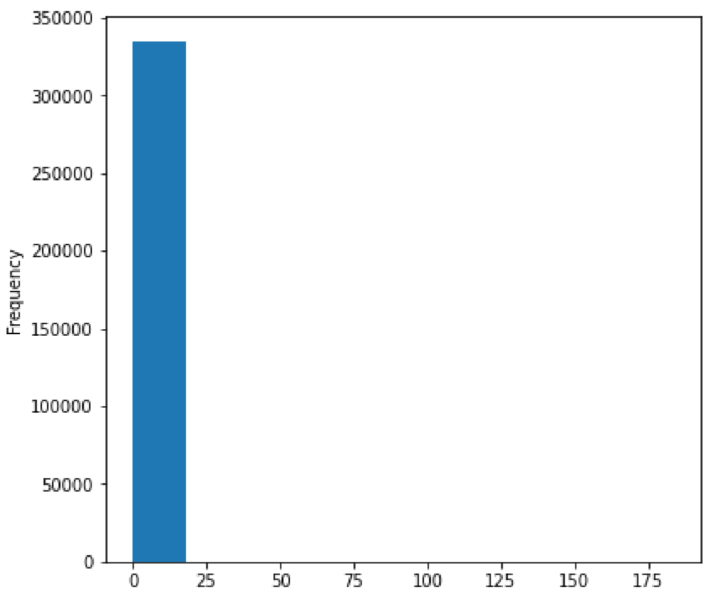
3.3.2) Exploring the features numerical1 and numerical2

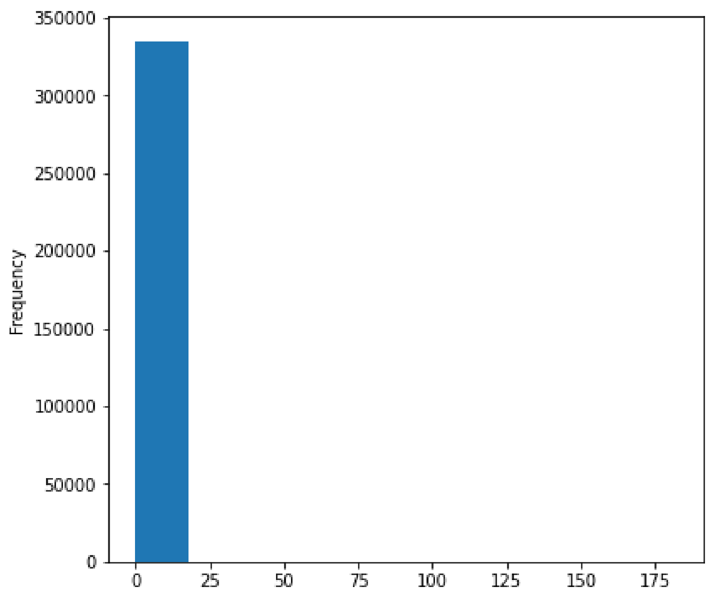
In [ ]:

```
fig, ax = plt.subplots(1, 2, figsize = (14, 6))
merchant['numerical_1'].plot(kind='hist', ax=ax[0])
merchant['numerical_2'].plot(kind='hist', ax=ax[1])
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff3a68b85c0>
```





The distribution of the values for both the columns looks same.

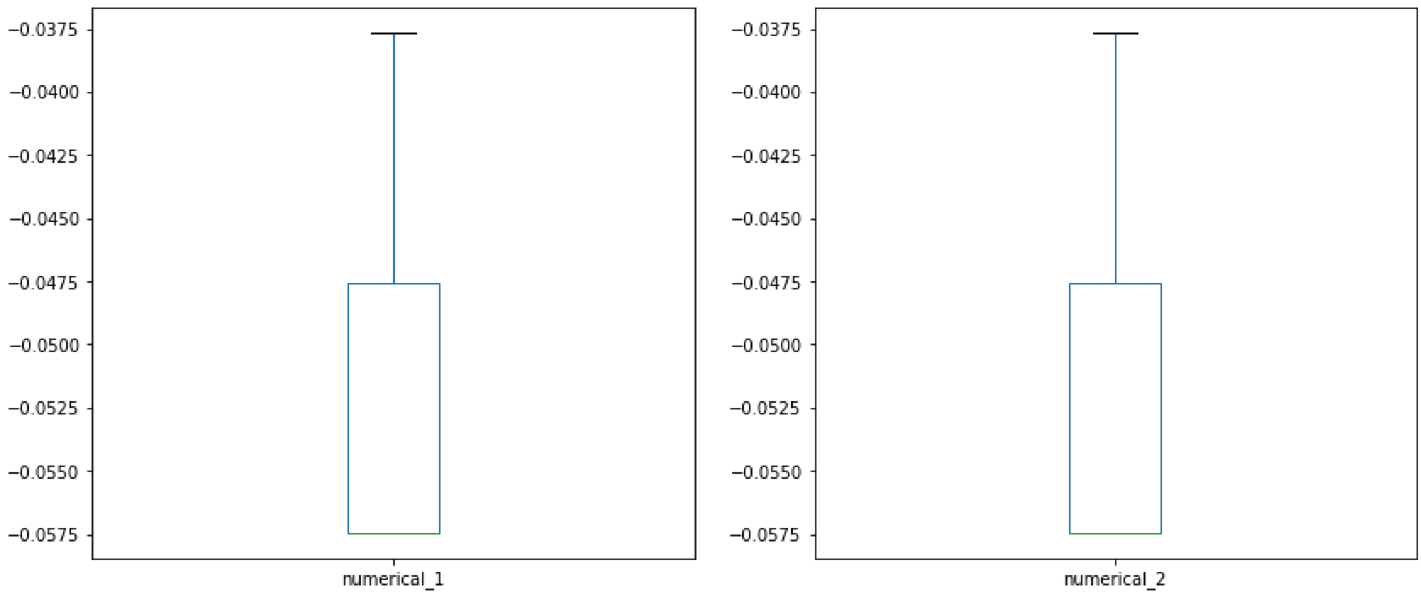
Let's confirm this with box plot.

In [ ]:

```
'''
box plot for the features numerical_1 and numerical_2
Here the showfliers is set to False i.e., it will not
show the outliers.
The reason to set it to False is if we set it to True
the plot becomes too congested and it kind of getting contracted to the middle.
However, since we are not performing outlier detection it is not important so
we can set it to False.
'''

fig, ax = plt.subplots(1, 2, figsize = (14, 6))
merchant['numerical_1'].plot(kind='box', showfliers=False, ax=ax[0])
merchant['numerical_2'].plot(kind='box',showfliers=False, ax=ax[1])
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff332257f60>
```



These two features looks exactly the same. Let's take a look at the five number summary

In [ ]:

```
print(merchant['numerical_1'].describe())
print(merchant['numerical_2'].describe())
```

```
count    334696.000000
mean          0.011482
std           0.000000
min          -0.057465
25%          -0.057465
50%          -0.057465
75%          -0.047546
max         183.750000
Name: numerical_1, dtype: float64
count    334696.000000
mean          0.008095
std           0.000000
min          -0.057465
25%          -0.057465
50%          -0.057465
75%          -0.047546
max         182.125000
Name: numerical_2, dtype: float64
```

The distribution of values for the two features looks identical. Both the features have the same 5 number stats like mean, std, max etcetera.

Let's find the value counts

In [ ]:

```
merchant['numerical_1'].value_counts()
```

Out[ ]:

```
-0.057465      228788
-0.047546       41528
-0.037628       15689
-0.027725        8297
-0.017807        5249
                ...  
 20.593750          1
 7.250000           1
 13.890625          1
 107.625000         1
 8.000000           1
Name: numerical_1, Length: 950, dtype: int64
```

The value -0.057465 occurs a lot of times. Let's find the percentage

In [ ]:

```
# calculating in what percentage the values are appearing in the dataset by dividing with the length of the data

merchant['numerical_1'].value_counts()/len(merchant)
```

Out[ ]:

```
-0.057465      0.683570
-0.047546      0.124077
-0.037628      0.046875
-0.027725      0.024790
-0.017807      0.015683
                 ...   
 20.593750     0.000003
 7.250000      0.000003
 13.890625     0.000003
 107.625000    0.000003
 8.000000      0.000003
Name: numerical_1, Length: 950, dtype: float64
```

The value -0.057465 occurs over 68%.

In [ ]:

```
# calculating in what percentage the values are appearing in the datasetby dividing with the length of the data
merchant['numerical_2'].value_counts()/len(merchant)
```

Out[ ]:

```
-0.057465     0.743104
-0.047546     0.099434
-0.037628     0.034790
-0.027725     0.018315
-0.017807     0.011811
                ...   
 10.039062    0.000003
 24.187500    0.000003
 5.593750     0.000003
 6.039062     0.000003
 16.015625    0.000003
Name: numerical_2, Length: 944, dtype: float64
```

Similarly, In numerical_2 feature too the value -0.057465 occurs around 74%.

Still not clear what these features are.

It looks more of a categorical feature rather than numerical. It takes only 950 unique values given there are over 300000 points.

However if it's a numerical I don't think these features will be useful since their variance is 0 and contains a lot of constant values.

Anyway let's move forward.

In [ ]:

```
print(len(merchant['numerical_1'].unique().tolist()))
print(len(merchant['numerical_2'].unique().tolist()))
print(len(merchant))
```

```
950
944
334696
```

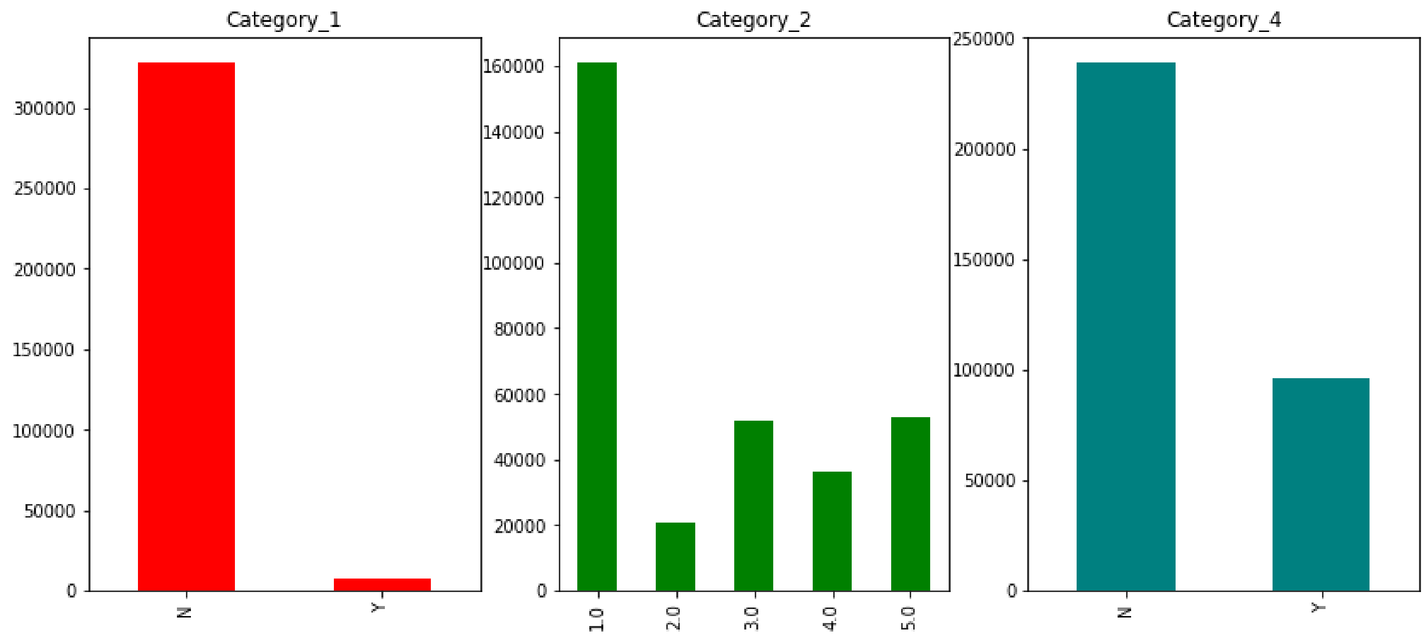
3.3.3) Now let's take a look at the anonymized Category_1,2 and 4 features

In [ ]:

```
fig, ax = plt.subplots(1, 3, figsize = (14, 6))
merchant['category_1'].value_counts().sort_index().plot(kind='bar', ax=ax[0], color='red
', title='Category_1')
merchant['category_2'].value_counts().sort_index().plot(kind='bar', ax=ax[1], color='gre
en', title='Category_2')
merchant['category_4'].value_counts().sort_index().plot(kind='bar', ax=ax[2], color='tea
l', title='Category_4')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f36a40bc588>
```



All the three features are caategorical.

1) Category_1 takes two values [Y, N]

2) Category_2 takes five values [1,2,3,4,5]

3) Category_3 takes two values [Y,N]

Should be one hot encoded as these are categoricaal

3.3.4) Average sales lag

In [ ]:

```
print(merchant['avg_sales_lag3'].describe())
print(merchant['avg_sales_lag6'].describe())
print(merchant['avg_sales_lag12'].describe())
```

```
count    334683.000000
mean         13.839176
std        2395.453369
min         -82.129997
25%           0.880000
50%           1.000000
75%           1.160000
max      851844.625000
Name: avg_sales_lag3, dtype: float64
count    3.346830e+05
mean     2.165529e+01
std      3.947046e+03
min     -8.213000e+01
25%      8.500000e-01
50%      1.010000e+00
75%      1.230000e+00
max      1.513959e+06
Name: avg_sales_lag6, dtype: float64
count    3.346830e+05
mean     2.523122e+01
std      5.251777e+03
min     -8.213000e+01
25%      8.500000e-01
50%      1.020000e+00
75%      1.290000e+00
max      2.567408e+06
Name: avg_sales_lag12, dtype: float64
```

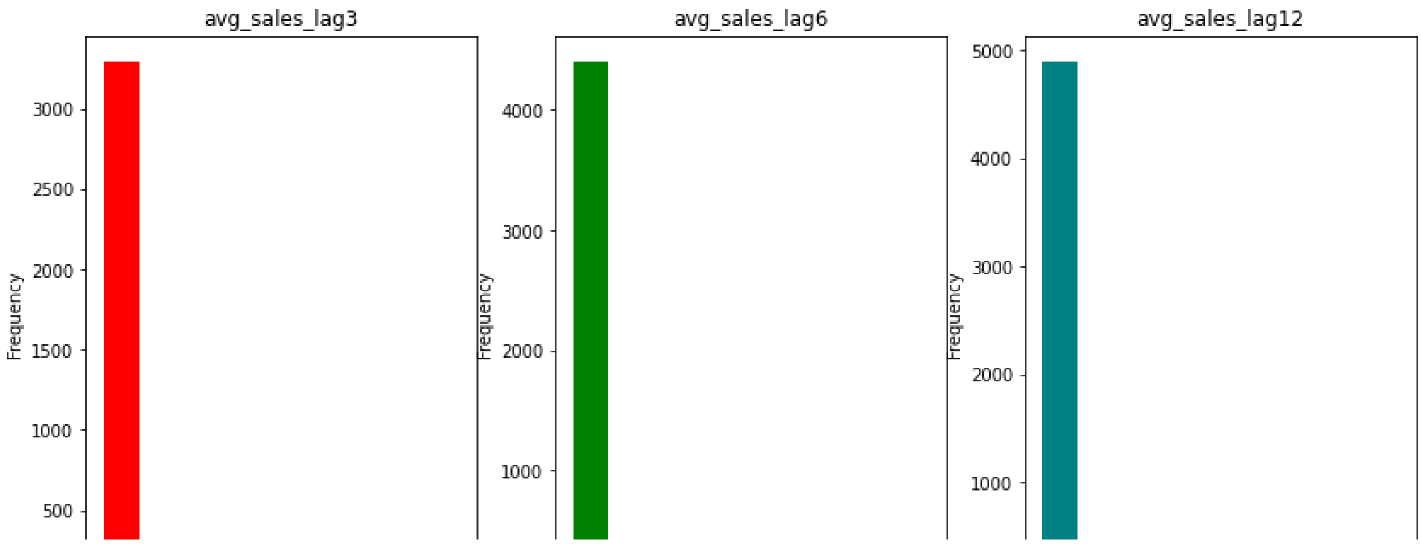
Let's plot the distribution of data

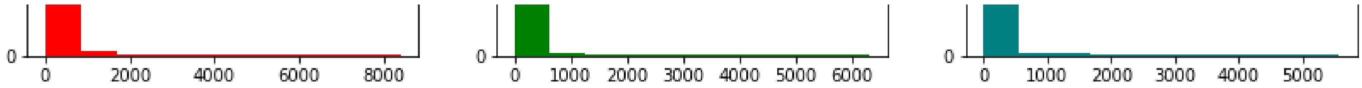
In [ ]:

```
fig, ax = plt.subplots(1, 3, figsize = (14, 6))
merchant['avg_sales_lag3'].value_counts().sort_index().plot(kind='hist', ax=ax[0], color
='red', title='avg_sales_lag3')
merchant['avg_sales_lag6'].value_counts().sort_index().plot(kind='hist', ax=ax[1], color
='green', title='avg_sales_lag6')
merchant['avg_sales_lag12'].value_counts().sort_index().plot(kind='hist', ax=ax[2], colo
r='teal', title='avg_sales_lag12')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f89ea2fa4e0>
```





Looks like all the three sales_lag distribution are similar. A lot of values are within may be 20. However as we can see from the plots there are also few extreme values which is greater than 8000.

To get a better picture of the values let's print the value counts and also will see the maximum value in each of the feature

In [ ]:

```
merchant['avg_sales_lag3'].value_counts()
```

Out[ ]:

```
1.000000       8411
0.980000       7953
0.990000       7891
0.970000       7663
0.960000       7572
               ...
48.070000         1
24.049999         1
48.119999         1
48.130001         1
2964.659912       1
Name: avg_sales_lag3, Length: 3372, dtype: int64
```

In [ ]:

```
max(merchant['avg_sales_lag3'])
```

Out[ ]:

```
851844.625
```

In [ ]:

```
merchant['avg_sales_lag6'].value_counts()
```

Out[ ]:

```
1.000000      6310
0.980000      5898
0.950000      5846
0.970000      5803
0.960000      5801
              ...
172.059998       1
43.009998        1
22.280001        1
54.490002        1
255.899994       1
Name: avg_sales_lag6, Length: 4507, dtype: int64
```

In [ ]:

```
merchant['avg_sales_lag12'].value_counts()
```

Out[ ]:

```
1.000000      5565
0.990000      5160
0.970000      5145
0.980000      5088
0.960000      5025
              ...
48.450001        1
96.839996        1
15.840000        1
12.100000        1
```

```
186.410004          1
Name: avg_sales_lag12, Length: 5009, dtype: int64
```

There are few extreme values in each feature might be outlier and should be handled appropriately.

In [ ]:

```
print(max(merchant['avg_sales_lag3']))
print(max(merchant['avg_sales_lag6']))
print(max(merchant['avg_sales_lag12']))
```

```
851844.625
1513959.0
2567408.0
```

3.3.5) Average purchase lags

In [ ]:

```
print(merchant['avg_purchases_lag3'].describe())
print(merchant['avg_purchases_lag6'].describe())
print(merchant['avg_purchases_lag12'].describe())
```

```
count    3.346960e+05
mean              inf
std               NaN
min      3.334953e-01
25%      9.236499e-01
50%      1.016667e+00
75%      1.146522e+00
max               inf
Name: avg_purchases_lag3, dtype: float64
count    3.346960e+05
mean              inf
std               NaN
min      1.670447e-01
25%      9.022475e-01
50%      1.026961e+00
75%      1.215575e+00
max               inf
Name: avg_purchases_lag6, dtype: float64
count    3.346960e+05
mean              inf
std               NaN
min      9.832954e-02
25%      8.983333e-01
50%      1.043361e+00
75%      1.266480e+00
max               inf
Name: avg_purchases_lag12, dtype: float64
```

For all the three features the max value is 'inf' and because of that the mean and the std values are getting goofed up.

So we need to take care of them. Let's investigate the inf values.

In [ ]:

```
# selecting rows which have value inf in the avg_purchases_lag3
merchant[merchant['avg_purchases_lag3']==np.inf]
```

Out[ ]:

	merchant_id	merchant_group_id	merchant_category_id	subsector_id	numerical_1	numerical_2	category_1	most_re
10	M_ID_492cfa500c	13462	369	27	-0.057465	-0.057465	N	
11	M_ID_73487fed26	17123	427	27	-0.057465	-0.057465	Y	
12	M_ID_7149162139	2118	63	27	-0.057465	-0.057465	N	

In [ ]:

```
# selecting rows which have value inf in the avg_purchases_lag6
merchant[merchant['avg_purchases_lag6']==np.inf]
```

Out[ ]:

	merchant_id	merchant_group_id	merchant_category_id	subsector_id	numerical_1	numerical_2	category_1	most_re
10	M_ID_492cfa500c	13462	369	27	-0.057465	-0.057465	N	
11	M_ID_73487fed26	17123	427	27	-0.057465	-0.057465	Y	
12	M_ID_7149162139	2118	63	27	-0.057465	-0.057465	N	

In [ ]:

```
# selecting rows which have value inf in the avg_purchases_lag12
merchant[merchant['avg_purchases_lag12']==np.inf]
```

Out[ ]:

	merchant_id	merchant_group_id	merchant_category_id	subsector_id	numerical_1	numerical_2	category_1	most_re
10	M_ID_492cfa500c	13462	369	27	-0.057465	-0.057465	N	
11	M_ID_73487fed26	17123	427	27	-0.057465	-0.057465	Y	
12	M_ID_7149162139	2118	63	27	-0.057465	-0.057465	N	

All the inf are for the same merchants

Since this value is calculated by Monthly average of transactions in last 3 months divided by transactions in last active month if the transactions in the last active month is 0 then this value could be inf. We can drop these 3 rows and proceed further

In [ ]:

```
'''
selecting rows where avg_purchases_lag3 is not inf and save it in a new DF.
So we can perform basic summary stats.
'''
merchant_no_inf = merchant[merchant['avg_purchases_lag3']!=np.inf]
merchant_no_inf.head()
```

Out[ ]:

	merchant_id	merchant_group_id	merchant_category_id	subsector_id	numerical_1	numerical_2	category_1	most_rec
0	M_ID_838061e48c	8353	792	9	-0.057465	-0.057465	N	
1	M_ID_9339d880ad	3184	840	20	-0.057465	-0.057465	N	
2	M_ID_e726bbae1e	447	690	1	-0.057465	-0.057465	N	
3	M_ID_a70e9c5f81	5026	792	9	-0.057465	-0.057465	Y	
4	M_ID_64456c37ce	2228	222	21	-0.057465	-0.057465	Y	

In [ ]:

```
print(merchant_no_inf['avg_purchases_lag3'].describe())
print(merchant_no_inf['avg_purchases_lag6'].describe())
print(merchant_no_inf['avg_purchases_lag12'].describe())
```

```
count    334693.000000
mean          1.590762
```

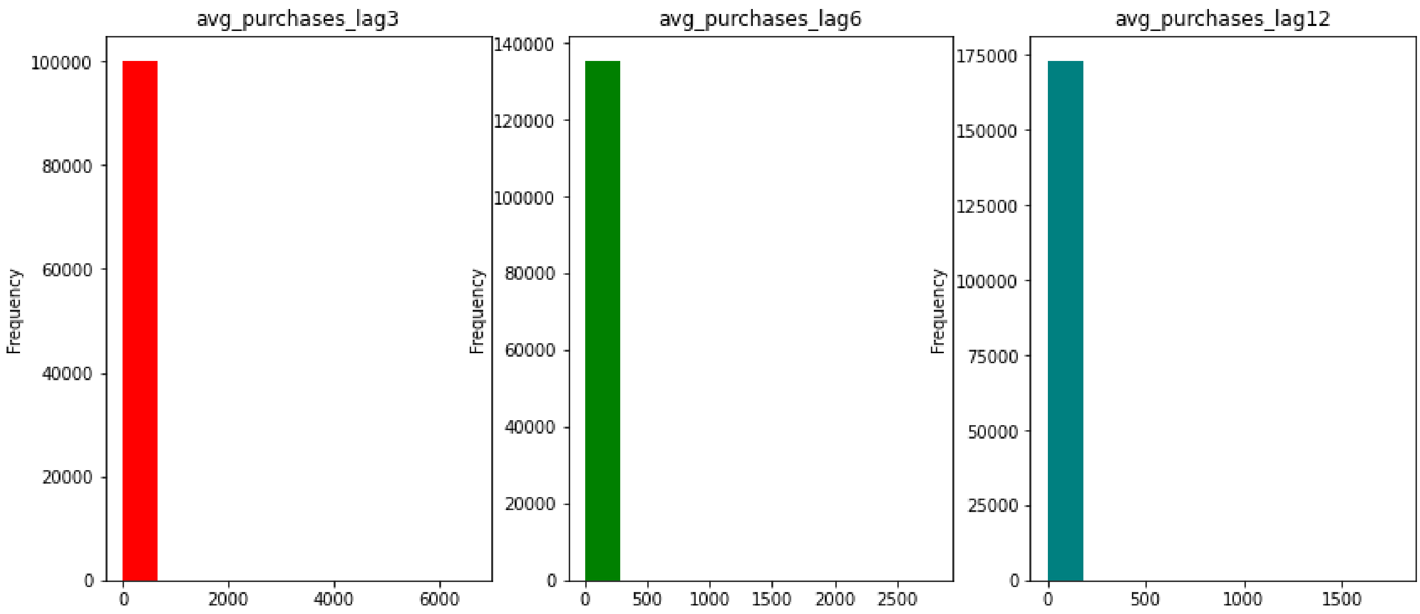
```
std         107.187059
min           0.333495
25%           0.923650
50%           1.016667
75%           1.146520
max       61851.333333
Name: avg_purchases_lag3, dtype: float64
count    334693.000000
mean          1.887568
std          97.862790
min           0.167045
25%           0.902245
50%           1.026961
75%           1.215556
max       56077.500000
Name: avg_purchases_lag6, dtype: float64
count    334693.000000
mean          2.079195
std          88.442384
min           0.098330
25%           0.898333
50%           1.043360
75%           1.266461
max       50215.555556
Name: avg_purchases_lag12, dtype: float64
```

In [ ]:

```
# plotting the distribution of data without the inf values
fig, ax = plt.subplots(1, 3, figsize = (14, 6))
merchant['avg_purchases_lag3'].value_counts().sort_index().plot(kind='hist', ax=ax[0], c
olor='red', title='avg_purchases_lag3')
merchant['avg_purchases_lag6'].value_counts().sort_index().plot(kind='hist', ax=ax[1], c
olor='green', title='avg_purchases_lag6')
merchant['avg_purchases_lag12'].value_counts().sort_index().plot(kind='hist', ax=ax[2],
color='teal', title='avg_purchases_lag12')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f89e2536438>
```



3.3.6) Let's explore sales range

In [ ]:

```
merchant['most_recent_sales_range'].value_counts()
```

Out[ ]:

```
E    177104
D    117475
```

```
C     34075
B      5037
A      1005
Name: most_recent_sales_range, dtype: int64
```

In [ ]:

```python
merchant['most_recent_purchases_range'].value_counts()
```

Out[ ]:

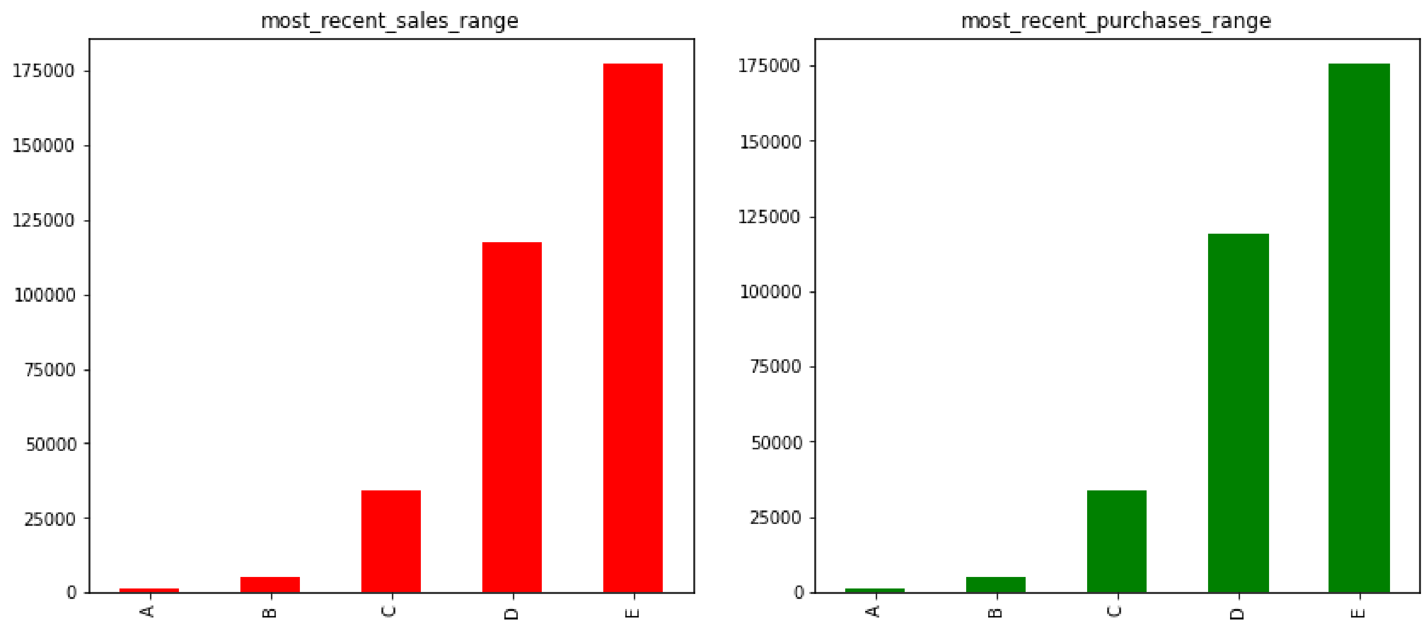
```
E    175309
D    119187
C     34144
B      5046
A      1010
Name: most_recent_purchases_range, dtype: int64
```

In [ ]:

```python
fig, ax = plt.subplots(1, 2, figsize = (14, 6))
merchant['most_recent_sales_range'].value_counts().sort_index().plot(kind='bar', ax=ax[0
], color='red', title='most_recent_sales_range')
merchant['most_recent_purchases_range'].value_counts().sort_index().plot(kind='bar', ax=
ax[1], color='green', title='most_recent_purchases_range')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f89e247f6d8>
```



No wonder they look the same. Because it is the recent sales and the recent purchases. As recent sales increase the recent purchase will also increase for that particular date

These two values could be correlated. Let's confirm it.

In [ ]:

```python
#selecting only the specified columns from the merchant and save it in a separate DF
cols = ['most_recent_sales_range', 'most_recent_purchases_range']
m = merchant[cols]
```

In [ ]:

```python
#encoding the cat variables with 1 to 5
d = {'A':1, 'B':2, 'C':3, 'D':4,'E':5}
m['most_recent_purchases_range'] = m['most_recent_purchases_range'].map(d)
m['most_recent_sales_range'] = m['most_recent_sales_range'].map(d)
```

In [ ]:

```
#calculating variance inflation factor for the two columns
from statsmodels.stats.outliers_influence import variance_inflation_factor

vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(m.iloc[:,:].values, i) for i in range(m.s
hape[1])]
vif["features"] = m.columns
vif
```

Out[ ]:

	VIF Factor	features
0	63.52409	most_recent_sales_range
1	63.52409	most_recent_purchases_range

The VIF value is around 64 denotes that these values are correlated. So we need to drop any one of the variable. Or we could do dummy variable encoding and check the score again.

Let's print the correlation matrix for all the features.

Correlation between variables

Variance Inflation Factor

In [ ]:

```
cols = ['active_months_lag3','active_months_lag6','active_months_lag12','numerical_1', 'n
umerical_2','avg_sales_lag3','avg_sales_lag6','avg_purchases_lag3','avg_sales_lag12','avg
_purchases_lag12','avg_purchases_lag6','category_2','state_id']
merchi = merchant[cols]
merchi = merchi.dropna()

from statsmodels.stats.outliers_influence import variance_inflation_factor

vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(merchi.iloc[:,:].values, i) for i in rang
e(merchi.shape[1])]
vif["features"] = merchi.columns
vif
```

Out[ ]:

	VIF Factor	features
0	517.963454	active_months_lag3
1	888.058008	active_months_lag6
2	148.780422	active_months_lag12
3	398.608414	numerical_1
4	398.584669	numerical_2
5	45.881735	avg_sales_lag3
6	252.021712	avg_sales_lag6
7	398.529760	avg_purchases_lag3
8	130.293237	avg_sales_lag12
9	580.462034	avg_purchases_lag12
10	1603.754005	avg_purchases_lag6
11	3.326845	category_2
12	5.577641	state_id

Looks like there are variables which are heavily correlated like active_months_lag6, numerical_1 and 2.

Looks like there are variables which are heavily correlated like active_months_lag3, numerical_1 and 2,
avg_purchase_lag6 and avg_sales_lag_6 and avg_purchase_lag12

Let's remove some of the variables and again we'll calculate the VIF

In [ ]:

```
cols = ['active_months_lag3','active_months_lag12','numerical_1', 'avg_sales_lag3','avg_p
urchases_lag3','avg_sales_lag12','avg_purchases_lag12','category_2','state_id']
merchi = merchant[cols]
merchi = merchi.dropna()

vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(merchi.iloc[:,:].values, i) for i in rang
e(merchi.shape[1])]
vif["features"] = merchi.columns
vif
```

Out[ ]:

	VIF Factor	features
0	77.199413	active_months_lag3
1	70.037085	active_months_lag12
2	1.000416	numerical_1
3	2.756102	avg_sales_lag3
4	79.868728	avg_purchases_lag3
5	1.801982	avg_sales_lag12
6	78.972876	avg_purchases_lag12
7	3.326589	category_2
8	5.576621	state_id

As we can see that after removing some of the correlated variables we can see that VIF score reduces significantly.

However there are still values which are above 10. However it is less than 100. So instead of removing them straightaway it needs further investigation.

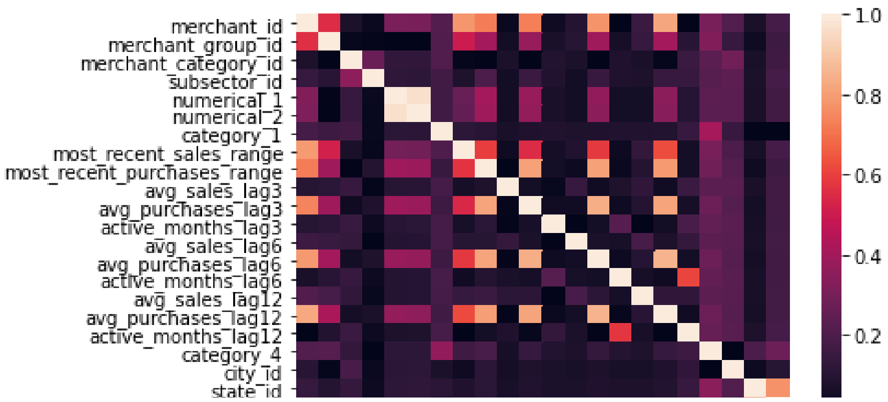
Let's plot the correlation matrix

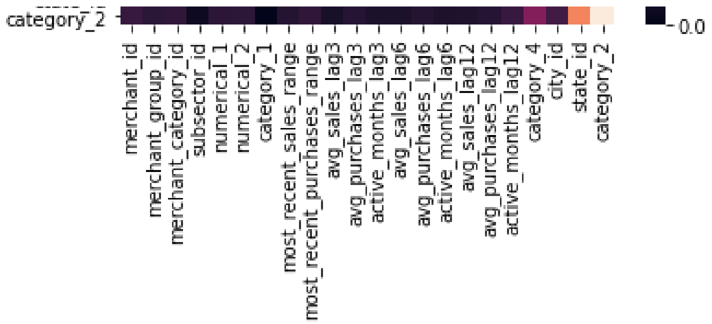
In [ ]:

```
corr = merchant.apply(lambda x : pd.factorize(x)[0]).corr(method='pearson', min_periods=
1)
corr_norm=np.round((corr-corr.min())/(corr.max()-corr.min()), 3)
sns.heatmap(corr_norm,
         xticklabels=corr.columns,
         yticklabels=corr.columns, annot=False)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff357b14fd0>
```





As we seen before as the numerical_1 and numerical_2 have similar values and distributions and they are correlated

The state_id is correlated with city_id and category_2. Since the anonymized feature category_2 is correlated with state_id I think it is somehow related to location.

There are notable correlations between purchase lags and month lags

merchant_id is correlated with merchant_group_id

3.4) new merchants data

3.4.1) Overview

In [ ]:

```
print('The shape of the data is:', new_merchant.shape)
```

```
The shape of the data is: (1963031, 14)
```

In [ ]:

```
new_merchant.head()
```

Out[ ]:

	authorized_flag	card_id	city_id	category_1	installments	category_3	merchant_category_id	merchant_id
0	Y	C_ID_415bb3a509	107	N	1	B	307	M_ID_b0c793002c
1	Y	C_ID_415bb3a509	140	N	1	B	307	M_ID_88920c89e8
2	Y	C_ID_415bb3a509	330	N	1	B	507	M_ID_ad5237ef6b
3	Y	C_ID_415bb3a509	-1	Y	1	B	661	M_ID_9e84cda3b1
4	Y	C_ID_ef55cf8d4b	-1	Y	1	B	166	M_ID_3c86fa3831

In [ ]:

```
new_merchant.isnull().sum()
```

Out[ ]:

```
authorized_flag             0
card_id                     0
city_id                     0
category_1                  0
installments                0
category_3              55922
merchant_category_id        0
merchant_id             26216
```

```
month_lag                    0
purchase_amount              0
purchase_date                0
category_2              111745
state_id                     0
subsector_id                 0
dtype: int64
```

In [ ]:

```
print('Percentage of missing values in category_3:', new_merchant['category_3'].isnull().
sum() / len(new_merchant['category_3']) * 100, '%')
print('Percentage of missing values in category_2:', new_merchant['category_2'].isnull().
sum() / len(new_merchant['category_2'])* 100, '%')
print('Percentage of missing values in merchant_id:', new_merchant['merchant_id'].isnull(
).sum() / len(new_merchant['merchant_id'])* 100, '%')
```

```
Percentage of missing values in category_3: 2.8487578647509895 %
Percentage of missing values in category_2: 5.692472508075522 %
Percentage of missing values in merchant_id: 1.3354857870303627 %
```

In [ ]:

```
print('Percentage of missing values in the dataset:', new_merchant.isnull().sum().sum() /
len(new_merchant) * 100, '%')
```

```
Percentage of missing values in the dataset: 9.876716159856874 %
```

There are null values in category3 and 2 and in merchant_id.

The categorical features can be lable encoded and we can create a separate category for the missing values.

In [ ]:

```
new_merchant['authorized_flag'].value_counts()
```

Out[ ]:

```
Y    1963031
Name: authorized_flag, dtype: int64
```

All the transactions are authorized so we don't have to worry about them.

Unlike the historical data where we have a fair share of unauthorized purchases here all the purchases are authorized so it doesn't make sense to create features separately for authorized and unauthorized as I proposed there.

Here since this column has a single value i.e., 1 for all the value this feature can be regarded as a constant feature and we can drop this.

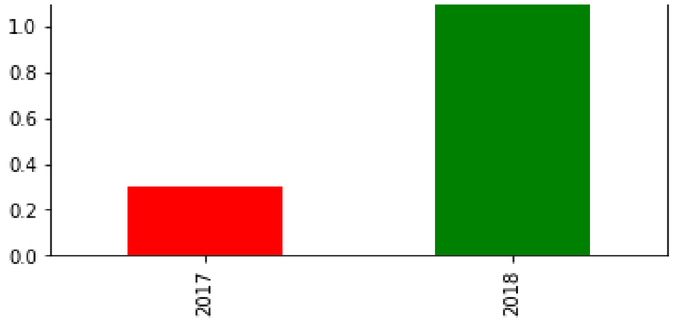
3.4.2) purchase date

In [ ]:

```
# get the year from the purchase date
year = pd.DatetimeIndex(new_merchant['purchase_date']).year

#getting the value count so that we can plot the number of purchases for each year
year.value_counts().sort_index().plot(kind='bar', color=['red','green'])
```

Out[ ]:

```
&lt;matplotlib.axes._subplots.AxesSubplot at 0x7f89e1f56a20&gt;
```



In [ ]:

```
year.value_counts()
```

Out[ ]:

```
2018    1659548
2017     303483
Name: purchase_date, dtype: int64
```

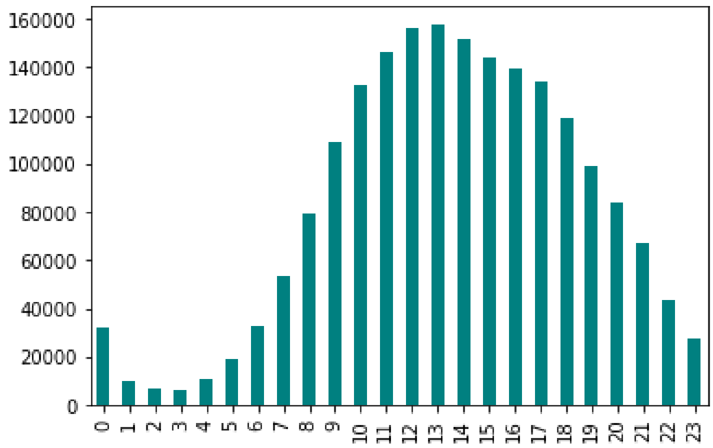
Maximum number of purchases are made in the year 2018

In [ ]:

```
'''
extract the part hour from the purchase date and
find the number of purchases for that particular hour using the
value counts so that we can visualize the number of purchases
made during particular time.
'''

hour = pd.DatetimeIndex(new_merchant['purchase_date']).hour
hour.value_counts().sort_index().plot(kind='bar', color='teal')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f89d8b0ccc0>
```



In [ ]:

```
hour.value_counts().sort_index()
```

Out[ ]:

```
0      32386
1       9657
2       6635
3       6493
4      10350
5      18876
6      33137
7      53529
8      79204
9     109094
10    132348
```

```
11     146332
12     156300
13     157810
14     151863
15     144363
16     139133
17     134024
18     119266
19      99387
20      84076
21      67341
22      43650
23      27777
Name: purchase_date, dtype: int64
```

A lot of purchases are made between 9am and 6pm.

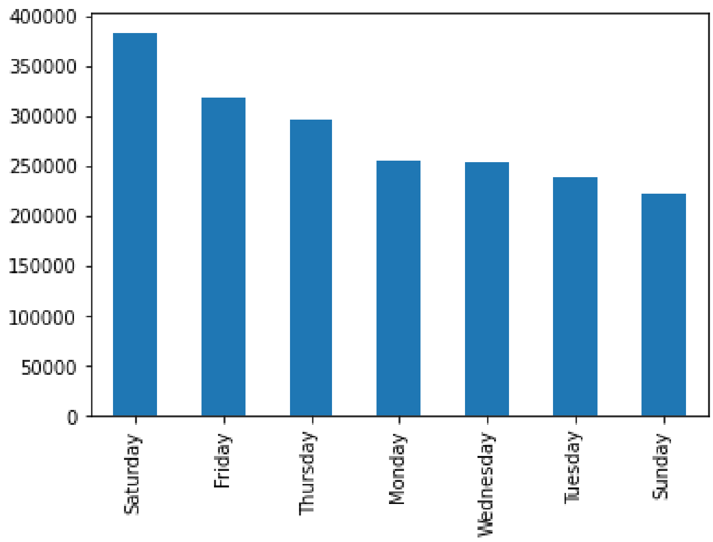
So creating a feature like hour of the day could be helpful.

In [ ]:

```
'''
extract the part day from the purchase date and
find the number of purchases for that particular day using the
value counts so that we can visualize the number of purchases
made during particular day.
'''

day_name = pd.DatetimeIndex(new_merchant['purchase_date']).day_name()
day_name.value_counts().sort_values(ascending=False).plot(kind='bar')
```

Out[ ]:

```
&lt;matplotlib.axes._subplots.AxesSubplot at 0x7f89c65f48d0&gt;
```



In [ ]:

```
day_name.value_counts()
```

Out[ ]:

```
Saturday     382769
Friday       317861
Thursday     295924
Monday       254158
Wednesday    253875
Tuesday      237702
Sunday       220742
Name: purchase_date, dtype: int64
```

Friday and Saturday there are lot of transactions.

Like the hour feature here we could create a feature like weekend or not as most of the purchases are made on

friday and saturdays.

In [ ]:

```
mont = pd.DatetimeIndex(new_merchant['purchase_date']).month_name()
mont
```

Out[ ]:

```
Index(['March', 'March', 'April', 'March', 'March', 'April', 'March', 'April',
       'April', 'March',
       ...
       'April', 'March', 'March', 'April', 'December', 'April', 'March',
       'March', 'March', 'April'],
      dtype='object', name='purchase_date', length=1963031)
```

In [ ]:

```
'''
Extracting the hour, day, month and year from the purchase date
to find out which month has maximum number of purchases
for each year.
'''
datet = new_merchant[['purchase_date']]
datet['hour']=pd.DatetimeIndex(datet['purchase_date']).hour
datet['mont']=pd.DatetimeIndex(datet['purchase_date']).month_name()
datet['year']=pd.DatetimeIndex(datet['purchase_date']).year
datet['day']=pd.DatetimeIndex(datet['purchase_date']).day_name()
datet
```

Out[ ]:

	purchase_date	hour	mont	year	day
0	2018-03-11 14:57:36	14	March	2018	Sunday
1	2018-03-19 18:53:37	18	March	2018	Monday
2	2018-04-26 14:08:44	14	April	2018	Thursday
3	2018-03-07 09:43:21	9	March	2018	Wednesday
4	2018-03-22 21:07:53	21	March	2018	Thursday
...	...	...	...	...	...
1963026	2018-04-06 14:36:52	14	April	2018	Friday
1963027	2018-03-07 13:19:18	13	March	2018	Wednesday
1963028	2018-03-05 12:04:56	12	March	2018	Monday
1963029	2018-03-09 14:47:05	14	March	2018	Friday
1963030	2018-04-11 07:59:46	7	April	2018	Wednesday

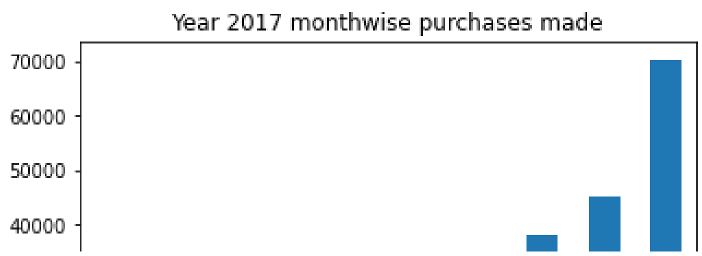
1963031 rows × 5 columns

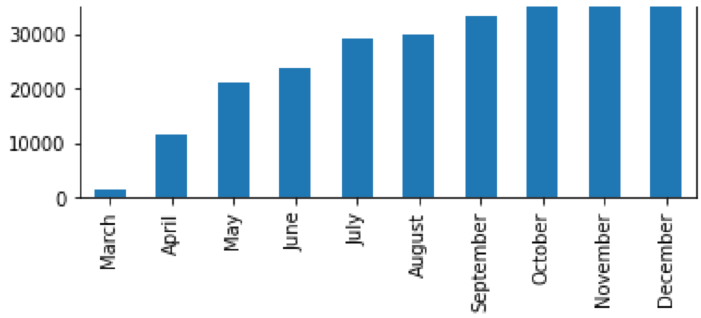
In [ ]:

```
# groupby year and get the months for the particular year
datet.groupby(['year']).get_group(2017)['mont'].value_counts().sort_values().plot(kind='
bar',title='Year 2017 monthwise purchases made')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f89b47299e8>
```





The month december has large number of purchases. May be because of christmas and new year.

We could create a feature whether the month is a festival month or not.

In [ ]:

```
#number of transactions made in 2017
datet.groupby(['year']).get_group(2017)['mont'].value_counts().sum()
```

Out[ ]:

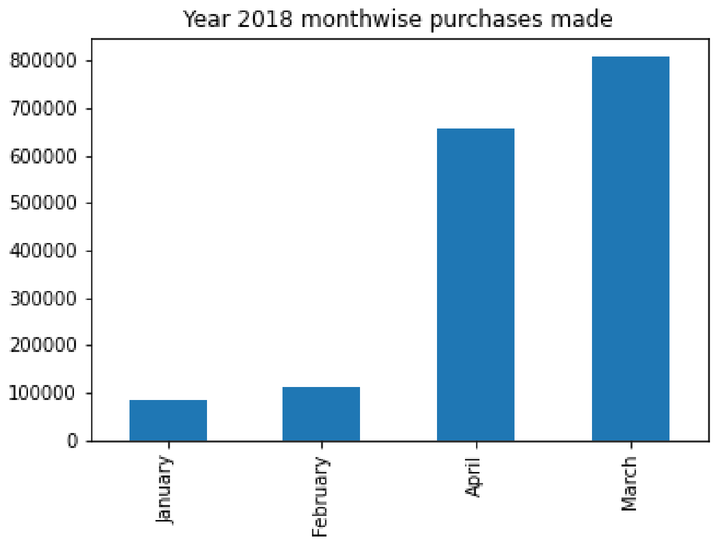
```
303483
```

In [ ]:

```
# groupby year and get the months for the particular year

datet.groupby(['year']).get_group(2018)['mont'].value_counts().sort_values().plot(kind='
bar',title='Year 2018 monthwise purchases made')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f89b4999080>
```



For the year 2018 we have data only for four months from january to march.

In addition to create feature like whether the month is festival month or not we could also create influencial month like people go shopping 2 months before any festival.

3.4.3) Exploring features category_1 ,2 and 3

In [ ]:

```
print(new_merchant['category_1'].value_counts())
print(new_merchant['category_2'].value_counts())
print(new_merchant['category_3'].value_counts())
```

```
N    1899935
Y      63096
Name: category_1, dtype: int64
```

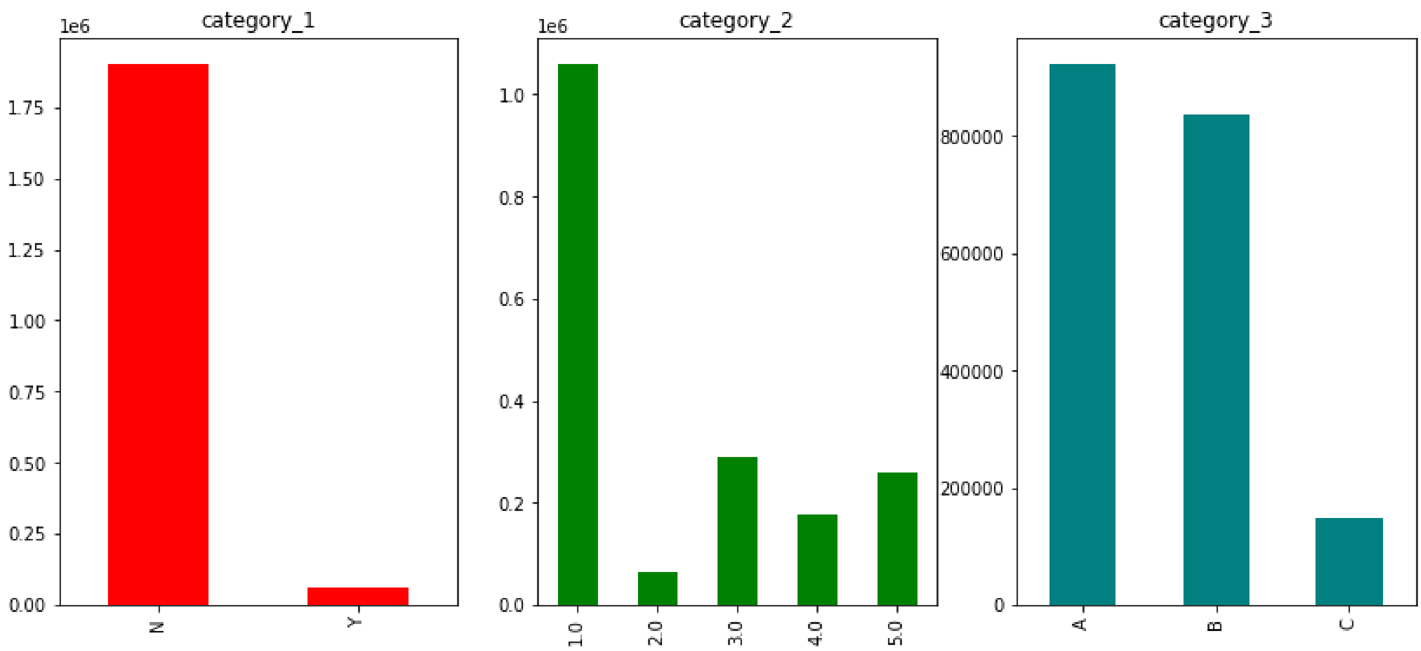
```
1.0    1058242
3.0     289525
5.0     259266
4.0     178590
2.0      65663
Name: category_2, dtype: int64
A    922244
B    836178
C    148687
Name: category_3, dtype: int64
```

In [ ]:

```
fig, ax = plt.subplots(1, 3, figsize = (14, 6))
new_merchant['category_1'].value_counts().sort_index().plot(kind='bar', ax=ax[0], color=
'red', title='category_1')
new_merchant['category_2'].value_counts().sort_index().plot(kind='bar', ax=ax[1], color=
'green', title='category_2')
new_merchant['category_3'].value_counts().sort_index().plot(kind='bar', ax=ax[2], color=
'teal', title='category_3')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7a9bf85198>
```



All the three are categorical features, category_1 takes two values Y or N

category_2 takes five values from 1 to 5

category_3 takes three values A, B or C

One hot encoding should be done.

Correlation using VIF

In [ ]:

```
d = {'A':1,'B':2,'C':3}
x = {'Y':1,'N':0}

cols = ['authorized_flag','category_3','month_lag','purchase_amount','state_id','subsect
or_id', 'category_2','installments']
n = new_merchant[cols]
n['category_3'] = n['category_3'].map(d)
n['authorized_flag'] = n['authorized_flag'].map(x)

n = n.dropna()
```

In [ ]:

```
print(new_merchant.columns)
```

```
Index(['authorized_flag', 'card_id', 'city_id', 'category_1', 'installments',
       'category_3', 'merchant_category_id', 'merchant_id', 'month_lag',
       'purchase_amount', 'purchase_date', 'category_2', 'state_id',
       'subsector_id'],
      dtype='object')
```

In [ ]:

```
vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(n.iloc[:,:].values, i) for i in range(n.s
hape[1])]
vif["features"] = n.columns
vif
```

Out[ ]:

	VIF Factor	features
0	31.880336	authorized_flag
1	1.493133	category_3
2	1.000173	month_lag
3	1.067400	purchase_amount
4	1.021572	state_id
5	1.010237	subsector_id
6	1.021653	category_2
7	1.528132	installments

The value for the authorized flag is somewhat higher, it is around 32 which indicates possible correlation. So this variable needs further investigation.

Other than the authorized flag the remaining variables doesn't look correlated. They are well under 2.

OBSERVATION

1) There are a total of 5 data files. train, test, new_merchant, merchant and historical transactions.

2) The features which are inherently in the train set is not much useful as the box plot of these features against the target variables. So we need to engineer some new features.

3) Plotting the distribution of the train data against the test data shows that both have the same distribution. So there's no need for time based splitting.

4) Except for the train and test data there are missing values in the remaining data. So it need to be filled with apposite method before modelling.

5) Since there are time feature like purchase date we can create new features which are extract of that feature. Like extracting days, months, weekend or check whether it is a holiday etcetera.

6) There are a lot of categorical features comparing to the numerical ones. The categorical features should be one hot encoded.

7) The features in the merchant dataset is highly correlated as we can see from the VIF scores. We can also see that after removing the correlated variables and again calculating the VIF score we can see reduce VIF scores.

8) In the historical transactions data theres is this feature called authorized_flag count which indicates whether the transaction is authorized or not. What we can do is we can separate our datasets as authorized or not and create features separately.

9) In the target variable even though it looks normally distributed around a central value there few outliers in the

data. However we cannot simply drop those data because those outliers may present in the test data also and we don't know that.

10) So what we can do is that we can create a new column whether it is outlier or not and let the model decide how to handle it.

11) Another thing we could do is create separate model for predicting outliers and the normal data.

Imputing using ML models

new_merchant.csv

partly inspired from: https://medium.com/towards-artificial-intelligence/handling-missing-data-for-advanced-machine-learning-b6eb89050357

In [57]:

```
new_merchant = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/new_merchant_tran
sactions.csv',parse_dates=["purchase_date"])
new_merchant = reduce_mem_usage(new_merchant)
```

```
Mem. usage decreased to 114.20 Mb (45.5% reduction)
```

In [ ]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

a = pd.DataFrame()
a['card_id'] = new_merchant['card_id']
a['merchant_id'] = new_merchant['merchant_id']
a['purchase_date'] = new_merchant['purchase_date']

new_merchant.drop(['card_id', 'merchant_id', 'purchase_date'], axis=1, inplace=True)
gc.collect()

feat = new_merchant.columns
cols = ['category_2', 'category_3']

#label encode the variables
new_merchant = lab_enc(new_merchant, ['authorized_flag','category_1'], prefix='new_merch
ant')

#list to hold the null values
no_nan = []

#select only columns which doesn't have any null values
for c in feat:
  if c not in cols:
    no_nan.append(c)

#create a test set by selecting only rows which are having null values
test = new_merchant[new_merchant['category_2'].isna()]

#create train set by selecting rows which doesn't have any null values
train = new_merchant.dropna()

#label encode the category 3 variables before feeding it to the model
d = {'A':1, 'B':2, 'C':3}
train['category_3'] = train['category_3'].map(d)
test['category_3'] = test['category_3'].map(d)

#fit the classifier to the train data
clf_cat2 = LogisticRegression()
clf_cat2.fit(train[no_nan], train['category_2'])
pickle.dump(clf_cat2, open('clf_cat2.sav', 'wb'))
```

```
#make prediction only for the rows with null value
new_merchant.loc[new_merchant['category_2'].isna(), 'category_2'] = clf_cat2.predict(tes
t[no_nan])

test = new_merchant[new_merchant['category_3'].isna()]
train = new_merchant.dropna()

clf_cat3 = LogisticRegression()
clf_cat3.fit(train[no_nan], train['category_3'])
pickle.dump(clf_cat3, open('clf_cat3.sav', 'wb'))

new_merchant.loc[new_merchant['category_3'].isna(), 'category_3'] = clf_cat3.predict(tes
t[no_nan])
```

In [ ]:

```
new_merchant.isna().sum()
```

Out[ ]:

```
authorized_flag         0
city_id                 0
category_1              0
installments            0
category_3              0
merchant_category_id    0
month_lag               0
purchase_amount         0
category_2              0
state_id                0
subsector_id            0
dtype: int64
```

In [ ]:

```
new_merchant['card_id'] = a['card_id']
new_merchant['merchant_id'] = a['merchant_id']
new_merchant['purchase_date'] = a['purchase_date']
```

In [ ]:

```
new_merchant.to_csv('new_merch_fill_na.csv')
!cp new_merch_fill_na.csv "/content/drive/My Drive/Colab Notebooks/ELO"
```

In [73]:

```
del a, new_merchant;gc.collect()
```

Out[73]:

```
123
```

historical transac

In [6]:

```
ht             = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/historical_trans
actions.csv',parse_dates=['purchase_date'])
ht             = reduce_mem_usage(ht)
```

```
Mem. usage decreased to 1749.11 Mb (43.7% reduction)
```

In [ ]:

```
ht.isna().sum()
```

Out[ ]:

```
authorized_flag               0
card_id                       0
city_id                       0
```

```
category_1                 0
installments               0
category_3            178159
merchant_category_id       0
merchant_id           138481
month_lag                  0
purchase_amount            0
purchase_date              0
category_2           2652864
state_id                   0
subsector_id               0
dtype: int64
```

In [ ]:

```
%%time
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

a = pd.DataFrame()
a['card_id'] = ht['card_id']
a['merchant_id'] = ht['merchant_id']
a['purchase_date'] = ht['purchase_date']

ht.drop(['card_id', 'merchant_id', 'purchase_date'], axis=1, inplace=True)
gc.collect()

feat = ht.columns
cols = ['category_2', 'category_3']

#laabel encode the variables
ht = lab_enc(ht, ['authorized_flag','category_1'], prefix='ht')

#list to hold the null values
no_nan = []

#select only columns which doesn't have any null values
for c in feat:
  if c not in cols:
    no_nan.append(c)

#create a test set by selecting only rows which are having null values
test = ht[ht['category_2'].isna()]

#create train set by selecting rows which doesn't have any null values
train = ht.dropna()

#label encode the category 3 variables before feeding it to the model
d = {'A':1, 'B':2, 'C':3}
train['category_3'] = train['category_3'].map(d)
test['category_3'] = test['category_3'].map(d)

#fit the classifier to the train data
ht_clf_cat2 = LogisticRegression()
ht_clf_cat2.fit(train[no_nan], train['category_2'])
pickle.dump(ht_clf_cat2, open('ht_clf_cat2.sav', 'wb'))

#make prediction only for the rows with null value
ht.loc[ht['category_2'].isna(), 'category_2'] = ht_clf_cat2.predict(test[no_nan])

#del clf;gc.collect()
```

```
CPU times: user 27min 48s, sys: 43.1 s, total: 28min 32s
Wall time: 27min 14s
```

In [ ]:

```
test = ht[ht['category_3'].isna()]
train = ht.dropna()

ht_clf_cat3 = LogisticRegression()
ht_clf_cat3.fit(train[no_nan], train['category_3'])
```

```
pickle.dump(ht_clf_cat3, open('ht_clf_cat3.sav', 'wb'))

ht.loc[ht['category_3'].isna(), 'category_3'] = ht_clf_cat3.predict(test[no_nan])
```

In [ ]:

```
ht.isna().sum()
```

Out[ ]:

```
authorized_flag         0
city_id                 0
category_1              0
installments            0
category_3              0
merchant_category_id    0
month_lag               0
purchase_amount         0
category_2              0
state_id                0
subsector_id            0
dtype: int64
```

In [ ]:

```
ht['card_id'] = a['card_id']
ht['merchant_id'] = a['merchant_id']
ht['purchase_date'] = a['purchase_date']
```

In [ ]:

```
ht.to_csv('ht_fill_na.csv')
!cp ht_fill_na.csv "/content/drive/My Drive/Colab Notebooks/ELO"
```

merchant.csv

In [15]:

```
merchant = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/merchants.csv')
merchant = reduce_mem_usage(merchant)
```

```
Mem. usage decreased to 30.32 Mb (46.0% reduction)
```

In [ ]:

```
from sklearn.neighbors import KNeighborsRegressor

#a = merchant.copy()
merchant = merchant[merchant['avg_purchases_lag3']!=np.inf]

tmp = pd.DataFrame()
tmp['merchant_id'] = merchant['merchant_id']
tmp['category_2'] = merchant['category_2']

merchant.drop(['merchant_id', 'category_2'], axis=1, inplace=True)

merchant = lab_enc(merchant, ['category_4','category_1','most_recent_sales_range','most_
recent_purchases_range'], prefix='merchant')

feat = merchant.columns
cols = ['avg_sales_lag3','avg_sales_lag6','avg_sales_lag12']
no_nan = []

for c in feat:
  if c not in cols:
    no_nan.append(c)

test = merchant[merchant['avg_sales_lag3'].isna()]
train = merchant.dropna()
```

```
merch_clf_knn = KNeighborsRegressor(n_neighbors=5)
merch_clf_knn.fit(train[no_nan], train['avg_sales_lag3'])
merchant.loc[merchant['avg_sales_lag3'].isna(), 'avg_sales_lag3'] = merch_clf_knn.predict
(test[no_nan])
pickle.dump(merch_clf_knn, open('merch_clf_knn.sav', 'wb'))

test = merchant[merchant['avg_sales_lag6'].isna()]
train = merchant.dropna()

merch_clf2_knn = KNeighborsRegressor(n_neighbors=5)
merch_clf2_knn.fit(train[no_nan], train['avg_sales_lag6'])
merchant.loc[merchant['avg_sales_lag6'].isna(), 'avg_sales_lag6'] = merch_clf2_knn.predic
t(test[no_nan])
pickle.dump(merch_clf2_knn, open('merch_clf2_knn.sav', 'wb'))

test = merchant[merchant['avg_sales_lag12'].isna()]
train = merchant.dropna()

merch_clf3_knn = KNeighborsRegressor(n_neighbors=5)
merch_clf3_knn.fit(train[no_nan], train['avg_sales_lag12'])
merchant.loc[merchant['avg_sales_lag12'].isna(), 'avg_sales_lag12'] = merch_clf3_knn.pred
ict(test[no_nan])
pickle.dump(merch_clf3_knn, open('merch_clf3_knn.sav', 'wb'))

merchant['category_2'] = tmp['category_2']

merchant.isna().sum()
```

Out[ ]:

```
merchant_group_id                 0
merchant_category_id              0
subsector_id                      0
numerical_1                       0
numerical_2                       0
category_1                        0
most_recent_sales_range           0
most_recent_purchases_range       0
avg_sales_lag3                    0
avg_purchases_lag3                0
active_months_lag3                0
avg_sales_lag6                    0
avg_purchases_lag6                0
active_months_lag6                0
avg_sales_lag12                   0
avg_purchases_lag12               0
active_months_lag12               0
category_4                        0
city_id                           0
state_id                          0
category_2                    11886
dtype: int64
```

In [ ]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression

feat = merchant.columns
cols = ['category_2']
no_nan = []

for c in feat:
  if c not in cols:
    no_nan.append(c)

test = merchant[merchant['category_2'].isna()]
train = merchant.dropna()

merch_clf_cat2 = LogisticRegression()
merch_clf_cat2.fit(train[no_nan], train['category_2'])
merchant.loc[merchant['category_2'].isna(), 'category_2'] = merch_clf_cat2.predict(test[n
```

```
o_nan])
pickle.dump(merch_clf_cat2, open('merch_clf_cat2.sav', 'wb'))
```

In [ ]:

```
merchant['merchant_id'] = tmp['merchant_id']
```

In [ ]:

```
merchant.isna().sum()
```

Out[ ]:

```
merchant_group_id              0
merchant_category_id           0
subsector_id                   0
numerical_1                    0
numerical_2                    0
category_1                     0
most_recent_sales_range        0
most_recent_purchases_range    0
avg_sales_lag3                 0
avg_purchases_lag3             0
active_months_lag3             0
avg_sales_lag6                 0
avg_purchases_lag6             0
active_months_lag6             0
avg_sales_lag12                0
avg_purchases_lag12            0
active_months_lag12            0
category_4                     0
city_id                        0
state_id                       0
category_2                     0
merchant_id                    0
dtype: int64
```

In [ ]:

```
merchant.to_csv('merch_fill_na.csv')
!cp merch_fill_na.csv "/content/drive/My Drive/Colab Notebooks/ELO"
```

In [ ]:

```

```

Pre-processing

In [ ]:

```
new_merchant = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/new_merch_fill_na
.csv')
new_merchant = reduce_mem_usage(new_merchant)

ht = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/ht_fill_na.csv')
ht = reduce_mem_usage(ht)
```

```
Mem. usage decreased to 95.48 Mb (57.5% reduction)
Mem. usage decreased to 1471.48 Mb (55.8% reduction)
```

In [ ]:

```
#label encoding the features
ht = lab_enc(ht, ['category_3'], prefix='ht')
new_merchant = lab_enc(new_merchant, ['category_3'], prefix='new_merchant')

gc.collect()
```

Out[ ]:

```
0
```

0

In [ ]:

```
# historical transactions one hot encoding
# tried with pandas get_dummies but as I am working in colab the kernel is crashing stati
ng memory exhausted

mont = [0,-1,-2,-3,-4,-5,-6]
cat_2 = [1.,2.,3.,4.,5.]
cat_3 = [0,1,2,3]

for val in mont:
  ht['month_lag={}'.format(val)] = (ht['month_lag'] == val).astype(int)

for val in cat_2:
  ht['category_2={}'.format(int(val))] = (ht['category_2'] == val).astype(int)

for val in cat_3:
  ht['category_3={}'.format(int(val))] = (ht['category_3'] == val).astype(int)
gc.collect()
```

Out[ ]:

```
60
```

In [ ]:

```
# new merchant one hot encoding
cat_2 = [1.,2.,3.,4.,5.]
cat_3 = [0,1,2,3]
mont = [1,2]

for val in mont:
  new_merchant['month_lag={}'.format(val)] = (new_merchant['month_lag'] == val).astype(i
nt)

for val in cat_2:
  new_merchant['category_2={}'.format(int(val))] = (new_merchant['category_2'] == val).a
stype(int)

for val in cat_3:
  new_merchant['category_3={}'.format(int(val))] = (new_merchant['category_3'] == val).a
stype(int)
gc.collect()
```

Out[ ]:

```
60
```

In [ ]:

```
#creating a reference month
ht['purchase_month'] = ht['purchase_date'].astype(str)
#for reference month keep the date constant and subtract the month lag from the month fie
ld
#inspired from a kaggle kernel(unable to find the discussion thread)
ht['reference_month'] = pd.to_datetime(ht['purchase_month'].apply(lambda x: x[:7] + '-28
')) - \
                                        ht['month_lag'].apply(lambda x: np.timedelta64(x
, 'M'));gc.collect()
```

Out[ ]:

```
0
```

In [ ]:

```
#extract only month from the reference month column
ht['reference_month'] = [x[:7] for x in ht['reference_month'].astype(str)]
del ht['purchase_month'];gc.collect()
```

In [ ]:

```
#do the same with the new_merchant data
new_merchant['reference_month'] = (pd.to_datetime(pd.DatetimeIndex(new_merchant['purchas
e_date']).date) - \
                                    new_merchant['month_lag'].apply(lambda x: np.timedelt
a64(x, 'M')))
new_merchant['reference_month'] = [x[:7] for x in new_merchant['reference_month'].astype
(str)]
```

In [ ]:

```
#save the file
new_merchant.to_csv('new_merchant_processed_fill_na.csv', index=False)
#!cp new_merchant_processed.csv "/content/drive/My Drive/Colab Notebooks/ELO"
ht.to_csv('ht_processed_fill_na.csv', index=False)
#!cp ht_processed.csv "/content/drive/My Drive/Colab Notebooks/ELO"
```

FEATURE ENGINEERING

Creating features based on the time features based on the historical transactiond and new merchants.

We'll create features grouped by the card_id as the card_id has some duplicate values and there are no null values.

In [ ]:

```
#loading the new_merchant dataset
new = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/new_merch_fill_na_processe
d.csv')
new_merchant_feats = pd.DataFrame(new.groupby(['card_id']).size()).reset_index()
new_merchant_feats.columns = ['card_id', 'new_transac_count']
#the purchase amount given to us is normalized. It does not make any sense if we look at
it.
#Credits to the user radar he somehow deanonymize the data and give the below formula to
transform the purchase
#amount which will make much sense
# kaggle.com/raddar/towards-de-anonymizing-the-data-some-insights
new['purchase_amount'] = np.round(new['purchase_amount'] / 0.00150265118 + 497.06, 2)

#loading the historical transactions data and group it by the column card_id
ht = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/ht_processed_fill_na.csv')
historical_trans_features = pd.DataFrame(ht.groupby(['card_id']).size()).reset_index()
historical_trans_features.columns = ['card_id', 'hist_transac_count']
#transforming the purchase amount
ht['purchase_amount'] = np.round(ht['purchase_amount'] / 0.00150265118 + 497.06, 2)
```

In the following code snippet we will find the nunique value. What nunique will return is the number of unique observations in the columns.

The columns in which the operation is going to be performed is denoted using the cols variable.

In [ ]:

```
#find the value nuniq for the cols specified
cols = ['city_id', 'state_id', 'merchant_category_id', 'subsector_id', 'merchant_id']
new_merchant_feats = find_single_val(new_merchant_feats, new, col=cols, grpby='card_id',\
                                     op=['nunique'], prefix='new_transac', use_col=True)

#find the value nuniq for the cols specified
cols = ['city_id', 'state_id', 'merchant_category_id', 'subsector_id', 'merchant_id']
historical_trans_features = find_single_val(historical_trans_features, new, col=cols, grp
by='card_id',\
                                     op=['nunique'], prefix='hist_transac', use_col=True)
```

Using the category_1 feature which takes binary value 1 or 0 we'll find the sum of this feature grouped by the card_id.

Once we calculate the sum of this feature what we have at hand is the number of 1's occuring. To find the number of 0's we'll subtract the sum from the total transaction count.

In [ ]:

```
#find the val sum of 1 for the cols specified
# since this is a binary feat (0 or 1)subtracting from the tot count to find the count of 0
new_merchant_feats = find_single_val(new_merchant_feats, new, col=['category_1'], grpby='card_id',\
                                     op=['sum'], prefix='new_transac', use_col=True)
new_merchant_feats['new_transac_category_0_sum'] = new_merchant_feats['new_transac_count'].values - new_merchant_feats.iloc[:, -1].values

#find the val sum of 1 for the cols specified
# since this is a binary feat (0 or 1)subtracting from the tot count to find the count of 0
historical_trans_features = find_single_val(historical_trans_features, new, col=['category_1'], grpby='card_id',\
                                     op=['sum'], prefix='hist_transac', use_col=True)
historical_trans_features['hist_transac_category_0_sum'] = historical_trans_features['hist_transac_count'].values - \
                                                                          historical_trans_features.iloc[:, -1].values
```

Using the feature we just created we'll find other numerical features like calculating the mean and standard deviation, min, max and skew.

In [ ]:

```
#find the val mean and std for the specified col
new_merchant_feats = find_single_val(new_merchant_feats, new, col=['category_1'], grpby='card_id',\
                                     op=['mean','std'], prefix='new_transac', use_col=True)

#aggregate feturess like mean, sum, max, min for the col specified
new_merchant_feats = s_agg(new_merchant_feats, new, col='installments', grpby='card_id',\
                                    op=['mean', 'sum', 'max', 'min', 'std', 'skew'], prefix='new_transac_')

#find the val mean and std for the specified col
historical_trans_features = find_single_val(historical_trans_features, new, col=['category_1'], grpby='card_id',\
                                                    op=['mean','std'], prefix='hist_transac', use_col=True)

#aggregate feturess like mean, sum, max, min for the col specified
historical_trans_features = s_agg(historical_trans_features, new, col='installments', grpby='card_id', \
                                              op=['mean', 'sum', 'max', 'min', 'std', 'skew'], \
                  prefix='hist_transac')
```

The feature category_2 takes 5 values. This feature is one hot encoded. So what we'll do is we'll find the sum and mean for each of the one hot encoded columns.

The feature category_2=1 indicates that the category_2 takes value 1 like that.

The same goes for the category_3 also

In [ ]:

```
#find val mean and sum for the col specified
cols = ['category_2=1', 'category_2=2', 'category_2=3', 'category_2=4', 'category_2=5',
        'category_3=0', 'category_3=1', 'category_3=2', 'category_3=3']
new_merchant_feats = find_single_val(new_merchant_feats, new, col=cols, grpby='card_id',\
                                              op=['mean','sum'], prefix='new_transac', use_col=True)
```

```
#find val mean and sum for the col specified
cols = ['category_2=1', 'category_2=2', 'category_2=3', 'category_2=4', 'category_2=5',
        'category_3=0', 'category_3=1', 'category_3=2', 'category_3=3']
historical_trans_features = find_single_val(historical_trans_features, new, col=cols, grp
by='card_id',\
                                            op=['mean','sum'], prefix='new_transac', us
e_col=True)
```

In the following snippet we'll create new features based off of the month_lag column which is inherently present in the given data. For that first we'll group the data by card_id.

Once the data is grouped then we'll calculate the values like sum, mean and standard deviation for the purchase amount column.

In the following we have calculated the above values for the purchase amount.

For the authorized flag feature we'll do the same as we did for the category_1 feature like finding sum and subtracting the sum from the total transaction count to get the sum of 0.

In [ ]:

```
grpby_lag = ['card_id', 'month_lag']
grpby_id  = ['card_id', 'merchant_id']
#get basic month stats
historical_trans_features = get_monthlag_stat(historical_trans_features, new, grpby=grpby
_lag, op='count', \
                                           col='purchase_amount', prefix='hist_transac',
name=['count_std','count_max'])

#authorized column mean and count features
historical_trans_features = find_single_val(historical_trans_features, new, col=['authori
zed_flag'], grpby='card_id',\
                                            op=['sum', 'mean'], prefix='hist_transac',
use_col=True)
#get authorized column denied count by subtracting authorized colum sum from total transa
ction count
historical_trans_features['hist_transac_denied_count'] = historical_trans_features['hist_
transac_count'].values - \
                                                         historical_trans_features.iloc
[:, -1].values

#find mean of the count of the transac for merchant id
historical_trans_features['hist_transac_merchant_id_count_mean'] = historical_trans_featu
res['hist_transac_count'].values \
                                                                   / historical_trans_fea
tures['hist_transac_merchant_id_nunique'].values

#basic features from authorized column like max, ratio, std
historical_trans_features['hist_transac_merchant_count_max'] = ht.groupby(grpby_id).size(
).reset_index()
historical_trans_features['hist_transac_merchant_count_max'] = groupby(['card_id'])[0].m
ax().values #tiup

#get basic month stats grouped by card_id and month_lag
new_merchant_feats = get_monthlag_stat(new_merchant_feats, new, grpby=grpby_lag, op='coun
t', \
                                       col='purchase_amount', prefix='new_transac_', nam
e=['count_std','count_max'])
```

Calculating simple ratios by dividing the already calculted values.

In [ ]:

```
historical_trans_features['hist_transac_merchant_ratio'] = historical_trans_features.iloc
[:, -1].values \
                                                                      / historical_tran
s_features['hist_transac_count'].values
historical_trans_features['hist_transac_merchant_id_ratio'] = historical_trans_features.i
loc[:, -2].values \
```

```
                                                                                / historical_
trans_features['hist_transac_merchant_id_count_mean'].values
historical_trans_features['hist_transac_merchant_count_std'] = ht.groupby(['card_id', 'me
rchant_id']).size().reset_index().\
                                                                    groupby(['card_id'])[0].
std().values
#save the created features
historical_trans_features.to_csv('hist_transac_info_fill_na.csv')
new_merchant_feats.to_csv('new_merch_info_fillna.csv')
```

Amount

Once the data is loaded we'll group them by the card_id

Next we'll calculate simple statistics like min, max, mean, medan and std for the purchase amount column.

The transaction amount difference feature is the difference between the maximum and minimum purchase amount for each card_id

Then we'll find the difference between the purchase amount max and min for each card_id

In [ ]:

```
new = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/new_merchant_processed_fil
l_na.csv')
new_merchant_feats = pd.DataFrame(new.groupby(['card_id']).size()).reset_index()
new_merchant_feats.columns = ['card_id', 'new_transac_count']
#the purchase amount given to us is normalized. It does not make any sense if we look at
it.
#Credits to the user radar he somehow deanonymize the data and give the below formula to
transform the purchase
#amount which will make much sense
# kaggle.com/raddar/towards-de-anonymizing-the-data-some-insights
new['purchase_amount'] = np.round(new['purchase_amount'] / 0.00150265118 + 497.06, 2)

ht = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/ht_processed_fill_na.csv')
historical_trans_features = pd.DataFrame(ht.groupby(['card_id']).size()).reset_index()
historical_trans_features.columns = ['card_id', 'hist_transac_count']
ht['purchase_amount'] = np.round(ht['purchase_amount'] / 0.00150265118 + 497.06, 2)

#crete agg features based on the purchase amount
op = ['sum', 'mean', 'max', 'min', 'median', 'std', 'skew']
new_merchant_feats = s_agg(new_merchant_feats, new, op=op, prefix='new_transac_', col='pu
rchase_amount', grpby='card_id')
#finding the difference between the maximum and minmum purchase amount
new_merchant_feats['new_transac_amount_diff'] = new_merchant_feats['new_transac_purchase_
amount_max'].values - \
                                                  new_merchant_feats['new_transac_purchase
_amount_min'].values

#create basic agg features from the purchase amount column grouped by card id
op = ['sum', 'mean', 'max', 'min', 'median', 'std', 'skew']
historical_trans_features = s_agg(historical_trans_features, ht, op=op, prefix='hist_tra
nsac_', \
                   col='purchase_amount', grpby='card_id')
#finding the difference between the purchase amount max and min
historical_trans_features['hist_transac_amount_diff'] = historical_trans_features['hist_t
ransac_purchase_amount_max'].values - \
                                         historical_trans_features['hist_transac_purchase
_amount_min'].values
```

Calculating the monthlag features for the purchase amount. The month lag take two values 1 and 2 we are finding the sum for these two month lags.

Finally we'll calculate the ratio by dividing the purchase amount for both the monthlags.

In [ ]:

```
#basic month features
```

```
new_merchant_feats = get_monthlag_stat(new_merchant_feats, new, grpby=['card_id','month_l
ag'], op='sum',\
                                        col='purchase_amount', prefix='new_transac_', \
                                        name=['1_amount','2_amount'])
# dividing monthlag2 by 1 to find the ratio
new_merchant_feats['new_transac_monthlag_ratio'] = (new_merchant_feats.iloc[:, -1] / new_
merchant_feats.iloc[:, -2])\
                                                          .replace([np.inf, -np.inf], n
p.nan)
#create another feature by taking the log of the ratio
new_merchant_feats['new_transac_monthlag_log_ratio'] = np.log2(new_merchant_feats.iloc[:,
-1])
```

The following feature successive aggregates is inspired from a kaggle kernel.

what the below code snippet does is that it group the data twice by different column for each groupby and find basic aggregate values.

First it will goup by card_id and the field1 for the second groupby it will group the data by card_id and the field2. We can mention the field1 and field2 in the method arguments. Then it will find the agg values like mean, min, max and std for the columns we have specified.

In [ ]:

```
#successive agg features
#create a temp DF ADD to hold the new features
add = successive_aggregates(ht, field1='category_1', field2='purchase_amount')
col = ['installments', 'city_id', 'merchant_category_id', 'merchant_id',\
       'subsector_id','category_2','category_3']

#for each column commpute the agg and merge with the temp DF
for c in col:
  add = add.merge(successive_aggregates(ht, c, 'purchase_amount'), \
            on=['card_id'], how='left')
#merge the temp DF with our feature set
new_merchant_feats = new_merchant_feats.merge(add, on=['card_id'], how='left')

#successive agg features
#create a temp DF ADD to hold the new features
add = successive_aggregates(new, 'category_1', 'purchase_amount')
col = ['installments', 'city_id', 'merchant_category_id', 'merchant_id',\
       'subsector_id','category_2','category_3']

#for each column commpute the agg and merge with the temp DF
for c in col:
  add = add.merge(successive_aggregates(new, c, 'purchase_amount'), \
            on=['card_id'], how='left')
#merge the temp DF with our feature set
historical_trans_features = historical_trans_features.merge(add, on=['card_id'], how='lef
t')

#save the created features
new_merchant_feats.to_csv('new_merch_amount_fillna.csv', index=False)
historical_trans_features.to_csv('hist_transac_amount_fill_na.csv', index=False)
```

Time

Loading the dataset new merchant and historical transactions and group by card_id

In [ ]:

```
new = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/new_merchant_processed_fil
l_na.csv')
new_merchant_feats = pd.DataFrame(new.groupby(['card_id']).size()).reset_index();gc.colle
ct()
new_merchant_feats.columns = ['card_id', 'new_transac_count']

ht = pd.read_csv('/content/drive/My Drive/Colab Notebooks/ELO/ht_processed_fill_na.csv')
historical_trans_features = pd.DataFrame(ht.groupby(['card_id']).size()).reset_index();g
```

```
c.collect()
historical_trans_features.columns = ['card_id', 'hist_transac_count']
ht['purchase_amount'] = np.round(ht['purchase_amount'] / 0.00150265118 + 497.06, 2)
```

Finding simple stats values mean, std and max for the column monthlag.

Then based on these newly created features create feat like difference and ratio by dividing the values

In [ ]:

```
#agg feat like mean, std, max for the column monthlag grouped by card_id
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'max'], prefix='n
ew_transac_', grpby='card_id', col='month_lag')

#get agg feats like min, mean, std for the col specified
historical_trans_features = s_agg(historical_trans_features, ht, ['nunique', 'mean', 'st
d', 'min', 'skew'], 'hist_transac_', 'card_id', 'month_lag')

#get values like min and max values from the col purchase_date
new_merchant_feats = find_single_val(new_merchant_feats, new, col=['purchase_date'], grpb
y='card_id', op=['max','min'], prefix='new_transac', use_col=True)
#based on the min and max find difference and ratio
new_merchant_feats['purchase_date_diff'] = (pd.to_datetime(new_merchant_feats.iloc[:, -2]
) -
                                            pd.to_datetime(new_merchant_feats.iloc[:, -1
])).dt.days.values
new_merchant_feats['purchase_count_ratio'] = new_merchant_feats['new_transac_count'].valu
es / (1. + new_merchant_feats.iloc[:, -1].values)

#get values like min and max values from the col purchase_date
historical_trans_features = find_single_val(historical_trans_features, ht, col=['purchase
_date'], grpby='card_id',\
                             op=['max','min'], prefix='hist_transac', use_col=True)
#create feats like difference and ratio between the first and last purchases made for a c
ard_id
historical_trans_features['hist_purchase_date_diff'] = (pd.to_datetime(historical_trans_f
eatures.iloc[:, -2]) - \
                                                        pd.to_datetime(historical_trans
_features.iloc[:, -1])).dt.days.values
historical_trans_features['hist_purchase_count_ratio'] = historical_trans_features['hist_
transac_count'].values / (1. + historical_trans_features.iloc[:, -1].values)
```

Now we'll create features based on whether a particular day is a weekend or not.

We'll mark a day as weekend if it is either saturday or sunday.

Since is_weekend is a numeric feature now we can find values like sum, mean etcetera. This will tells us that how many times a particular card_id made purchase during weekends.

Next we'll create a feature month difference calculated by subtracting the purchase date from the reference month.

Once we have the feature month_diff then we can find sum and mean of the feat. Since the date has a dtype of timedelta we cannot directly calculate the month difference as the timedelta has no attribute to calc month. We first need to find the days then we can divide it by 30 to get the month value.

In [ ]:

```
reference_date = '2018-12-31'
#features based on if the particular day is a weekend
new['is_weekend'] = (pd.DatetimeIndex(new['purchase_date']).dayofweek)
#>5 to check whether the day is sat or sunday if it is then assign a val 1 else 0
new['is_weekend'] = new['is_weekend'].apply(lambda x: 1 if x >= 5 else 0).values
#get the values of mean and sum grouped by card_id for the weekend feature
new_merchant_feats = find_single_val(new_merchant_feats, new, col=['is_weekend'], grpby='
card_id', name='purchase_weekend_count',\
                            op=['sum'], prefix='new_transac')
new_merchant_feats = find_single_val(new_merchant_feats, new, col=['is_weekend'], grpby='
card_id', name='purchase_weekend_mean',\
```

```
                            op=['mean'], prefix='new_transac')

#features based on if the particular day is a weekend
#day is termed as weekend if it is either sat or sunday
ht['is_weekend'] = (pd.DatetimeIndex(ht['purchase_date']).dayofweek)
#>5 to check whether the day is sat or sunday if it is then assign a val 1 else 0
ht['is_weekend'] = ht['is_weekend'].apply(lambda x: 1 if x >= 5 else 0).values
#get the values of mean and sum grouped by card_id for the weekend feature
# find purchases made in weekend sum
historical_trans_features = find_single_val(historical_trans_features, ht, col=['is_weeke
nd'], grpby='card_id', \
                                            name='purchase_weekend_count', op=['sum'],
prefix='hist_transac')
#find purchases made in weekend mean
historical_trans_features = find_single_val(historical_trans_features, ht, col=['is_weeke
nd'], grpby='card_id', \
                                            name='purchase_weekend_mean', op=['mean'],
prefix='hist_transac')
historical_trans_features = historical_trans_features.merge(ht[['card_id', 'reference_mon
th']]\
.drop_duplicates(), on='card_id', how='left')
historical_trans_features['reference_month'] = pd.to_datetime(historical_trans_features[
'reference_month'])

purchase_date = pd.to_datetime(new['purchase_date'])
reference_date = pd.to_datetime(reference_date)
# We need to find the difference in days then we can divide by 30 to convert it into mont
hs.
# as timedelta doesn't have attribute to directly get months.
new['month_diff'] = (reference_date - purchase_date).dt.days
new['month_diff'] = new['month_diff'] // 30 + new['month_lag']
new['month_diff'].head()
new_merchant_feats = find_single_val(new_merchant_feats, new, col=['month_diff'], grpby='
card_id', \
                                     name='new_month_diff_mean', op=['mean'])

purchase_date = pd.to_datetime(ht['purchase_date'])
reference_date = pd.to_datetime(reference_date)
# We need to find the difference in days then we can divide by 30 to convert it into mont
hs.
# as timedelta doesn't have attribute to directly get months.
ht['month_diff'] = (reference_date - purchase_date).dt.days
ht['month_diff'] = ht['month_diff'] // 30 + ht['month_lag']
ht['month_diff'].head()
historical_trans_features = s_agg(historical_trans_features, ht, op=['mean', 'std', 'min
', 'max'], \
                   col='month_diff', grpby='card_id', prefix='hist_')
```

calculating the month ratio for the column purchase amount. Once we have the month ration feature we'll perform aggregate operations like finding mean, std, min, max based off of the month ratio feature.

The amount month ratio is calculated by dividing the purchase amount by the month difference feature we created in the last part.

The 1 is added in the denominator to nullify the division by zero error.

In [ ]:

```
new['amount_month_ratio'] = new['purchase_amount'].values / (1. + new['month_diff'].valu
es)
#agg feat based on the cols created in the last part
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'min', 'max', 'sk
ew'], \
                          prefix='new_transac_', grpby='card_id', col='duration')
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'min', 'max', 'sk
ew'], \
                          prefix='new_transac_', grpby='card_id', col='amount_month_rat
io')
#find sum and mean of the col monthlag col grouped by card_id
new_merchant_feats = find_single_val(new_merchant_feats, new, col=['month_lag=1', 'month_
```

```
lag=2'], grpby='card_id',\
                              op=['sum','mean'], prefix='new_transac', use_col=True)

ht['amount_month_ratio'] = ht['purchase_amount'].values / (1. + ht['month_diff'].values)
#agg feat based on the cols created in the last part
historical_trans_features = s_agg(historical_trans_features, ht, ['mean', 'std', 'min',
'max', 'skew'], \
                   prefix='hist_transac_', grpby='card_id', col='duration')
historical_trans_features = s_agg(historical_trans_features, ht, ['mean', 'std', 'min',
'max', 'skew'], \
                   prefix='hist_transac_', grpby='card_id', col='amount_month_ratio')
#find sum and mean of the col monthlag col grouped by card_id
historical_trans_features = find_single_val(historical_trans_features, ht, col=['month_la
g=0', 'month_lag=-1', 'month_lag=-2'],\
                                            grpby='card_id', op=['sum','mean'], prefix=
'hist_transac', use_col=True)
```

In the below cell we'll extract the week, day and hour from the column purchase_date.

Once we have the extracted values from the date column now we can create basic aggregates like mean, min and max for all the three features.

Since we are grouping by card_id this feature will tells us information about each card what is the mean purchase hour, at which week of the year they made a purchase etcetera.

In [ ]:

```
#extract week, day, and hour from the date column then
#create agg features like mean, min, max for each of the
#features separately
ht['week'] = pd.DatetimeIndex(ht['purchase_date']).week.values
ht['day'] = pd.DatetimeIndex(ht['purchase_date']).dayofweek.values
ht['hour'] = pd.DatetimeIndex(ht['purchase_date']).hour.values
#get aggregate values from the cols week, day and hour
gc.collect()
historical_trans_features = s_agg(historical_trans_features, ht, op=['nunique', 'mean',
'min', 'max'], \
                   col='week', grpby='card_id', prefix='hist_transac')
historical_trans_features = s_agg(historical_trans_features, ht, op=['nunique', 'mean',
'min', 'max'], \
                   col='day', grpby='card_id', prefix='hist_transac')
historical_trans_features = s_agg(historical_trans_features, ht, op=['nunique', 'mean',
'min', 'max'], \
                   col='hour', grpby='card_id', prefix='hist_transac')
```

In the following code snippet we are calculating the difference in days, seconds, minutes between the current and previous purchase for a particular card_id.

To do that we'll first group the data by card_id and shift the purchase date value. Then we'll find the difference between the days, seconds, minutes etcetera. This feature will gives us information about the difference in days between purchases for each card_id.

Once we have these numerical features then we can apply the functin s_agg to calculate the aggregate values like mean, std etcetera.

In [ ]:

```
#calculating the ratio between the two monthlags
new_merchant_feats['new_transac_month_lag=1_2_ratio'] = new_merchant_feats['new_transac_m
onth_lag=1_sum'].values \
                                                          / (1. + new_merchant_feats['new
_transac_month_lag=2_sum'].values)
#get basic time feat and create agg features based on the created cols
new = get_basic_time_feat(new, 'card_id', 'purchase_date', 2)
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'max', 'min'], pr
efix='new_transac_', \
                           grpby='card_id', col='purchase_date_diff_1_seconds')
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'max', 'min'], pr
efix='new_transac_', \
```

```
                              grpby='card_id', col='purchase_date_diff_1_days')
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'max', 'min'], pr
efix='new_transac_', \
                              grpby='card_id', col='purchase_date_diff_1_hours')
#get basic time feat and create agg features based on the created cols
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'max', 'min'], pr
efix='new_transac_', \
                              grpby='card_id', col='purchase_date_diff_2_seconds')
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'max', 'min'], pr
efix='new_transac_', \
                              grpby='card_id', col='purchase_date_diff_2_days')
new_merchant_feats = s_agg(new_merchant_feats, new, op=['mean', 'std', 'max', 'min'], pr
efix='new_transac_', \
                              grpby='card_id', col='purchase_date_diff_2_hours')
```

based on the features created above now find values like ratio, sum etcetera

In [ ]:

```
#find the ratio between the monthlag cols
historical_trans_features['hist_transac_monthlag_0_-1_ratio'] = historical_trans_features
.iloc[:, -6].values \
                                                                    / (1. + historical_trans_
features.iloc[:, -4].values)
historical_trans_features['hist_transac_monthlag_0_-2_ratio'] = historical_trans_features
.iloc[:, -7].values \
                                                                    / (1. + historical_trans_
features.iloc[:, -3].values)
#create a feature of the sum of all the three monthlag sum
#crete a temp dataframe which holds the three cols
col = ['hist_transac_month_lag=0_sum', 'hist_transac_month_lag=-1_sum', 'hist_transac_mon
th_lag=-2_sum']
tmp = historical_trans_features[col]
#perform sum operation over the cols
historical_trans_features['hist_transac_3mon_sum'] = tmp.sum(axis=1)
del tmp;gc.collect()
historical_trans_features['hist_transac_3mon_ratio'] = historical_trans_features.iloc[:,
-1].values \
                                                            / (1. + historical_trans_features
['hist_transac_count'].values)
```

like we created new features by shifting the values of the purchase date for the new merchants data we are gonna do the same for the historical transactions as well

Sort the purchase date and shift the values of the purchase date column.

Then find the difference between the seconds, days etcetera. This will give us the diffrence between the last and current purchase date.

Once we have the new columns of difference in purchase date based on days, minutes and seconds we can now perform operations like min, max, mean on the columns to create new aggregate features.

In [ ]:

```
#if it gives an error use ht['purchase_date'] = pd.to_datetime(ht['purchase_date'])
ht = ht.sort_values('purchase_date')
#get basic time feat and create agg features based on the created cols
ht = get_basic_time_feat(ht, 'card_id', 'purchase_date', 1)
#get basic time feat and create agg features based on the created cols
historical_trans_features = s_agg(historical_trans_features, ht, op=['mean', 'std', 'max
', 'min'], prefix='hist_transac_', \
                                     grpby='card_id', col='purchase_date_diff_1_seconds')
historical_trans_features = s_agg(historical_trans_features, ht, op=['mean', 'std', 'max
', 'min'], prefix='hist_transac_', \
                                     grpby='card_id', col='purchase_date_diff_1_days')
historical_trans_features = s_agg(historical_trans_features, ht, op=['mean', 'std', 'max
', 'min'], prefix='hist_transac_', \
                                     grpby='card_id', col='purchase_date_diff_1_hours')
```

In the following cell we will create influential days features.

What the below code does is that it will find whether a purchase is made 100 days before a festival. If it is so then it will be consider as an influential day.

Based on this feature now we'll create new features by finding basic stats like finding the mean for each of the holiday columns like christmas, fathers day etcetera.

In [ ]:

```
#create influential day features. If a purchase is made withing 100 days
#before or after a festival then it is called as influential days.
holiday = ['ChristmasDay_2017', 'FathersDay_2017', 'ChildrenDay_2017', 'BlackFriday_2017'
, 'ValentineDay_2017', 'MothersDay_2018']
date = ['2017-12-25', '2017-08-13', '2017-10-12', '2017-11-24', '2017-06-12', '2018-05-13
']

for idx, day in enumerate(holiday):
  new = get_influential(new, day, date[idx])

#loop through all the created features and add it to the DataFrame
for c in holiday:
    gc.collect()
    new_merchant_feats['new_transac_{}_mean'.format(c)] = new.groupby(['card_id'])[c]\
                                                           .mean().values
new_merchant_feats.drop(['new_transac_count'], axis=1, inplace=True)

#create influential day features. If a purchase is made withing 100 days
#before or after a festival then it is called as influential days.
ht['purchase_date'] = pd.to_datetime(ht['purchase_date'])
for idx, day in enumerate(holiday):
  ht = get_influential(ht, day, date[idx])

#loop through all the created features and add it to the DataFrame
for c in holiday:
  gc.collect()
  historical_trans_features['hist_transac_{}_mean'.format(c)] = ht.groupby(['card_id'])[
c]\
                                                                 .mean().values
historical_trans_features.drop(['hist_transac_count'],axis=1,inplace=True)

#save the created features
new_merchant_feats.to_csv('new_merch_time_fillna.csv', index=False)
historical_trans_features.to_csv('hist_transac_time_fill_na.csv', index=False)
```

FEATURES CREATED

QUICK SUMMARY OF WHAT ARE ALL THE FEATURES CREATED:

1) Creating features out of the date columns like adding whether it is a weekday, weekend, any special festive day or holiday, difference between dates, first and last registered dates. Could also engineer a feature like if a purchase is made within days before or after a festival then we can call it an influential day for making a purchase.

2) Transaction count (count)/success and failure count for each card_id (classified by authorized_flag)

3) category_1/ category_2/ category_3, are subjected to one-hot conversion and findding the mean and sum for each column separately.

4)Count/ max of each card_id under different month_lag

5)Sum/ mean/ max/ min/ median/ std of the transaction amount of each card_id

6) The statistical features like finding min, max, difference, average, percentiles of the time difference of the card_id transaction.

7) Creating new features by feature interaction like summing two features, taking ratio etcetera.

8)The ratio of weekends and working days for each card_id transaction Creating agg. features grouped by card_id and like finding the count of the purchases a particular card_id made during weekend etcetera and perform typical aggregate features like avg, min, max.

9) Features out of categorical features. For instance, there is this feature called authorized_flag takes two values ('Y' or 'N') which indicate whether a transaction is authorized or not. We can map the Y or N to 1 and 0 and we can find the sum of that column grouped by card_id. This will give us how many transactions are authorized for a particular card_id

FEATURE SELECTION USING RECURSIVE FEATURE ELIMINATION

Loading the dataset

In [11]:

```
hist_transac_amount = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/hist_t
ransac_amount_fill_na.csv')
hist_transac_info = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/hist_tra
nsac_info_fill_na.csv')
hist_transac_time = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/hist_tra
nsac_time_fill_na.csv')

new_merch_amount = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/new_merch
_amount_fillna.csv')
new_merch_info = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/new_merch_i
nfo_fillna.csv')
new_merch_time = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/new_merch_t
ime_fill_na.csv')
```

In [12]:

```
hist_transac_info.drop('Unnamed: 0', axis=1, inplace=True)
new_merch_info.drop('Unnamed: 0', axis=1, inplace=True)
new_merch_time.drop('Unnamed: 0', axis=1, inplace=True)
```

In [13]:

```
hist_feats = hist_transac_info.merge(hist_transac_amount, on='card_id', how='left')
hist_feats = hist_feats.merge(hist_transac_time, on='card_id', how='left')
```

In [14]:

```
del hist_transac_info, hist_transac_amount, hist_transac_time
gc.collect()
```

Out[14]:

```
0
```

In [15]:

```
new_feats = new_merch_info.merge(new_merch_amount, on='card_id', how='left')
new_feats = new_feats.merge(new_merch_time, on='card_id', how='left')
```

In [16]:

```
del new_merch_info, new_merch_amount, new_merch_time
gc.collect()
```

Out[16]:

```
0
```

In [17]:

```
train_df = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/train.csv')
test_df = pd.read_csv('/content/drive/My Drive/case study/upload 15mis/test.csv')
```

merging the historical feat and new_merchant feat with the train and test data

In [18]:

```
train_df = train_df.merge(hist_feats, on=['card_id'], how='left')
test_df = test_df.merge(hist_feats, on=['card_id'], how='left')

train_df = train_df.merge(new_feats, on=['card_id'], how='left')
test_df = test_df.merge(new_feats, on=['card_id'], how='left')
```

In [19]:

```
train_df['outliers'] = 0
train_df.loc[train_df['target'] < -30, 'outliers'] = 1
```

The train dataset has a date column first_active_month. We'll just create simple time features based on time like difference max etcetera

In [20]:

```
#creating basic time features from the train set
act_date = pd.to_datetime('2018-12-31')

for df in [train_df, test_df]:
    #converting the col ref_month an first_act_month to datetime type
    reference_month = pd.to_datetime(df['reference_month'])
    first_act_month = pd.to_datetime(df['first_active_month'])
    #extracting the year and month from the first_act_month
    df['year'] = pd.DatetimeIndex(df['first_active_month']).year.values
    df['month'] = pd.DatetimeIndex(df['first_active_month']).month.values
    df['month_diff'] = (reference_month - \
                                        first_act_month).dt.days.values

    df['elapsed_days'] = (act_date - reference_month).dt.days.values

    df['hist_purchase_active_diff'] = (pd.to_datetime(df['hist_transac_purchase_date_min
'].astype(str)\
                                         .apply(lambda x: x[:7])) - first_act_month).dt.
days.values
    df['hist_purchase_recency'] = (act_date - pd.to_datetime(df['hist_transac_purchase_d
ate_max'])).dt.days.values
    df['new_purchase_recency'] = (act_date - pd.to_datetime(df['new_transac_purchase_dat
e_max'])).dt.days.values
```

In [23]:

```
train_df.head()
```

Out[23]:

	first_active_month	card_id	feature_1	feature_2	feature_3	target	hist_transac_count_x	hist_transac_city_id_nu
0	2017-06	C_ID_92a2005557	5	2	1	-0.820283	260	
1	2017-01	C_ID_3d0044924f	4	1	0	0.392913	350	
2	2016-08	C_ID_d639edf6cd	2	2	0	0.688056	43	
3	2017-09	C_ID_186d6a6901	4	3	0	0.142495	77	
4	2017-11	C_ID_cdbd2c0db2	1	3	0	-0.159749	133	

5 rows × 336 columns

In [22]:

```
train_df = lab_enc(train_df, ['year', 'month'], prefix='train_df')
test_df = lab_enc(test_df, ['year', 'month'], prefix='test_df')
```

In [ ]:

```
train_cols = [c for c in train_df.columns if c not in ['hist_transac_purchase_date_max',
'hist_transac_purchase_date_min', 'new_transac_purchase_date_max', 'new_transac_purchase_
date_min',\
'hist_purchase_date_last', 'hist_purchase_date_first', 'reference_month', 'hist_purchase_
a_date_last', 'hist_purchase_a_date_first', 'new_purchase_date_last', 'new_purchase_date_
first','card_id', 'first_active_month','first_active_month', 'target','outliers','feature
_1','feature_2','feature_3','refernce_month','ref_first_month_diff_days']]
target = train_df['target']
del train_df['target']
```

In [ ]:

```
train_df[train_cols].shape
#train_df
```

Out[ ]:

```
(201917, 323)
```

In [ ]:

```
import gc
import logging
import datetime
import warnings
import numpy as np
import pandas as pd
import seaborn as sns
import lightgbm as lgb
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import StratifiedKFold
```

In [ ]:

```
outliers = train_df['outliers']
```

In [ ]:

```
from lightgbm import LGBMRegressor
from sklearn.feature_selection import RFECV

clf = LGBMRegressor(boosting_type='gbdt', objective='regression', num_iteration=10000,num
_leaves=120,
                        min_data_in_leaf=90,max_depth=8, learning_rate=0.01, feature_fra
ction= 0.7,
                        bagging_freq= 1,bagging_fraction= 0.9,data_random_seed= 11,metri
c= 'rmse',lambda_l1=0.4,
                        verbosity= -1,random_state= 4950)

rfe = RFECV(estimator=clf, step=3, cv=StratifiedKFold(n_splits=2, random_state=42) \
              .split(train_df[train_cols],outliers.values), \
               n_jobs=1, verbose=2)

rfe.fit(train_df[train_cols],target)
```

```
Fitting estimator with 323 features.
Fitting estimator with 320 features.
Fitting estimator with 317 features.
Fitting estimator with 314 features.
Fitting estimator with 311 features.
Fitting estimator with 308 features.
Fitting estimator with 305 features.
Fitting estimator with 302 features.
Fitting estimator with 299 features.
Fitting estimator with 296 features.
Fitting estimator with 293 features.
Fitting estimator with 290 features.
Fitting estimator with 287 features.
```

```
Fitting estimator with 284 features.
Fitting estimator with 281 features.
Fitting estimator with 278 features.
Fitting estimator with 275 features.
Fitting estimator with 272 features.
Fitting estimator with 269 features.
Fitting estimator with 266 features.
Fitting estimator with 263 features.
Fitting estimator with 260 features.
Fitting estimator with 257 features.
Fitting estimator with 254 features.
Fitting estimator with 251 features.
Fitting estimator with 248 features.
Fitting estimator with 245 features.
Fitting estimator with 242 features.
Fitting estimator with 239 features.
Fitting estimator with 236 features.
Fitting estimator with 233 features.
Fitting estimator with 230 features.
Fitting estimator with 227 features.
Fitting estimator with 224 features.
Fitting estimator with 221 features.
Fitting estimator with 218 features.
Fitting estimator with 215 features.
Fitting estimator with 212 features.
Fitting estimator with 209 features.
Fitting estimator with 206 features.
Fitting estimator with 203 features.
Fitting estimator with 200 features.
Fitting estimator with 197 features.
Fitting estimator with 194 features.
Fitting estimator with 191 features.
Fitting estimator with 188 features.
Fitting estimator with 185 features.
Fitting estimator with 182 features.
Fitting estimator with 179 features.
Fitting estimator with 176 features.
Fitting estimator with 173 features.
Fitting estimator with 170 features.
Fitting estimator with 167 features.
Fitting estimator with 164 features.
Fitting estimator with 161 features.
Fitting estimator with 158 features.
Fitting estimator with 155 features.
Fitting estimator with 152 features.
Fitting estimator with 149 features.
Fitting estimator with 146 features.
Fitting estimator with 143 features.
Fitting estimator with 140 features.
Fitting estimator with 137 features.
Fitting estimator with 134 features.
Fitting estimator with 131 features.
Fitting estimator with 128 features.
Fitting estimator with 125 features.
Fitting estimator with 122 features.
Fitting estimator with 119 features.
Fitting estimator with 116 features.
Fitting estimator with 113 features.
Fitting estimator with 110 features.
Fitting estimator with 107 features.
Fitting estimator with 104 features.
Fitting estimator with 101 features.
Fitting estimator with 98 features.
Fitting estimator with 95 features.
Fitting estimator with 92 features.
Fitting estimator with 89 features.
Fitting estimator with 86 features.
Fitting estimator with 83 features.
Fitting estimator with 80 features.
Fitting estimator with 77 features.
Fitting estimator with 74 features.
Fitting estimator with 71 features.
```

```
Fitting estimator with 68 features.
Fitting estimator with 65 features.
Fitting estimator with 62 features.
Fitting estimator with 59 features.
Fitting estimator with 56 features.
Fitting estimator with 53 features.
Fitting estimator with 50 features.
Fitting estimator with 47 features.
Fitting estimator with 44 features.
Fitting estimator with 41 features.
Fitting estimator with 38 features.
Fitting estimator with 35 features.
Fitting estimator with 32 features.
Fitting estimator with 29 features.
Fitting estimator with 26 features.
Fitting estimator with 23 features.
Fitting estimator with 20 features.
Fitting estimator with 17 features.
Fitting estimator with 14 features.
Fitting estimator with 11 features.
Fitting estimator with 8 features.
Fitting estimator with 5 features.
Fitting estimator with 2 features.
Fitting estimator with 323 features.
Fitting estimator with 320 features.
Fitting estimator with 317 features.
Fitting estimator with 314 features.
Fitting estimator with 311 features.
Fitting estimator with 308 features.
Fitting estimator with 305 features.
Fitting estimator with 302 features.
Fitting estimator with 299 features.
Fitting estimator with 296 features.
Fitting estimator with 293 features.
Fitting estimator with 290 features.
Fitting estimator with 287 features.
Fitting estimator with 284 features.
Fitting estimator with 281 features.
Fitting estimator with 278 features.
Fitting estimator with 275 features.
Fitting estimator with 272 features.
Fitting estimator with 269 features.
Fitting estimator with 266 features.
Fitting estimator with 263 features.
Fitting estimator with 260 features.
Fitting estimator with 257 features.
Fitting estimator with 254 features.
Fitting estimator with 251 features.
Fitting estimator with 248 features.
Fitting estimator with 245 features.
Fitting estimator with 242 features.
Fitting estimator with 239 features.
Fitting estimator with 236 features.
Fitting estimator with 233 features.
Fitting estimator with 230 features.
Fitting estimator with 227 features.
Fitting estimator with 224 features.
Fitting estimator with 221 features.
Fitting estimator with 218 features.
Fitting estimator with 215 features.
Fitting estimator with 212 features.
Fitting estimator with 209 features.
Fitting estimator with 206 features.
Fitting estimator with 203 features.
Fitting estimator with 200 features.
Fitting estimator with 197 features.
Fitting estimator with 194 features.
Fitting estimator with 191 features.
Fitting estimator with 188 features.
Fitting estimator with 185 features.
Fitting estimator with 182 features.
Fitting estimator with 179 features.
```

```
Fitting estimator with 176 features.
Fitting estimator with 173 features.
Fitting estimator with 170 features.
Fitting estimator with 167 features.
Fitting estimator with 164 features.
Fitting estimator with 161 features.
Fitting estimator with 158 features.
Fitting estimator with 155 features.
Fitting estimator with 152 features.
Fitting estimator with 149 features.
Fitting estimator with 146 features.
Fitting estimator with 143 features.
Fitting estimator with 140 features.
Fitting estimator with 137 features.
Fitting estimator with 134 features.
Fitting estimator with 131 features.
Fitting estimator with 128 features.
Fitting estimator with 125 features.
Fitting estimator with 122 features.
Fitting estimator with 119 features.
Fitting estimator with 116 features.
Fitting estimator with 113 features.
Fitting estimator with 110 features.
Fitting estimator with 107 features.
Fitting estimator with 104 features.
Fitting estimator with 101 features.
Fitting estimator with 98 features.
Fitting estimator with 95 features.
Fitting estimator with 92 features.
Fitting estimator with 89 features.
Fitting estimator with 86 features.
Fitting estimator with 83 features.
Fitting estimator with 80 features.
Fitting estimator with 77 features.
Fitting estimator with 74 features.
Fitting estimator with 71 features.
Fitting estimator with 68 features.
Fitting estimator with 65 features.
Fitting estimator with 62 features.
Fitting estimator with 59 features.
Fitting estimator with 56 features.
Fitting estimator with 53 features.
Fitting estimator with 50 features.
Fitting estimator with 47 features.
Fitting estimator with 44 features.
Fitting estimator with 41 features.
Fitting estimator with 38 features.
Fitting estimator with 35 features.
Fitting estimator with 32 features.
Fitting estimator with 29 features.
Fitting estimator with 26 features.
Fitting estimator with 23 features.
Fitting estimator with 20 features.
Fitting estimator with 17 features.
Fitting estimator with 14 features.
Fitting estimator with 11 features.
Fitting estimator with 8 features.
Fitting estimator with 5 features.
Fitting estimator with 2 features.
Fitting estimator with 323 features.
Fitting estimator with 320 features.
Fitting estimator with 317 features.
Fitting estimator with 314 features.
Fitting estimator with 311 features.
Fitting estimator with 308 features.
Fitting estimator with 305 features.
Fitting estimator with 302 features.
Fitting estimator with 299 features.
Fitting estimator with 296 features.
Fitting estimator with 293 features.
Fitting estimator with 290 features.
Fitting estimator with 287 features.
```

```
Fitting estimator with 284 features.
Fitting estimator with 281 features.
Fitting estimator with 278 features.
Fitting estimator with 275 features.
Fitting estimator with 272 features.
Fitting estimator with 269 features.
Fitting estimator with 266 features.
Fitting estimator with 263 features.
Fitting estimator with 260 features.
Fitting estimator with 257 features.
CPU times: user 6h 40min 33s, sys: 2min 12s, total: 6h 42min 46s
```

In [ ]:

```python
print(rfe.n_features_)
```

```
254
```

After the feature selection by RFE we are left with 254 features. Let's print the features

In [ ]:

```python
rfe.support_
```

Out[ ]:

```
array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True, False,  True,  True,  True,  True,
        True, False,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True, False, False,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True, False, False,  True,  True, False,  True, False,  True,
        True,  True,  True,  True, False,  True, False,  True, False,
        True, False,  True, False,  True, False,  True,  True,  True,
       False,  True, False,  True, False,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True, False,  True, False,  True, False,  True,  True,  True,
        True,  True,  True,  True,  True, False,  True,  True, False,
       False,  True,  True, False,  True,  True,  True,  True,  True,
        True,  True, False,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True, False,  True, False, False,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True, False,  True,  True,  True, False,  True,  True,  True,
        True,  True,  True, False, False, False, False, False, False,
       False,  True,  True, False, False, False, False, False,  True,
       False,  True,  True, False, False, False, False, False, False,
        True, False,  True,  True,  True, False,  True, False, False,
       False, False,  True, False,  True, False,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True, False,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True, False,  True,  True, False,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True, False, False,  True, False,
        True,  True,  True,  True,  True,  True,  True,  True, False,
       False, False,  True,  True,  True,  True, False,  True,  True,
       False,  True,  True,  True, False,  True,  True,  True])
```

The support attribute returns a boolean array. If true then the feature at that index is selected. Let's print the actual feature names

In [ ]:

```python
# storing the actual features and the features selected by rfe (boolean list).
all_feat = train_df[train_cols].columns
```

```
sel = rfe.support_
```

Let's write a for loop to check whether a feature is selected or not.

First we'll enumerate through the original features.

If that features value is True in the array returned by RFE then we'll add it to our list of final features.

In [ ]:

```
#list to store the final selected features
rfe_final_feats = []
#loop through all the features in the train set
for idx, feat in enumerate(all_feat):
  #for that feature if the rfe value is True then add it to the final feature list.
  if sel[idx] == True:
    rfe_final_feats.append(feat)
```

Save the features

In [ ]:

```
np.save('rfe_final_feats.npy', rfe_final_feats)
!cp rfe_final_feats.npy '/content/drive/My Drive/case study/upload 15mis/'
```

Let's load the saved features and print it.

In [ ]:

```
final_fecat = np.load('/content/drive/My Drive/case study/upload 15mis/final_feat_rfe.npy
', allow_pickle=True)
final_feat
```

Quick Summary of What has been done till now.

1) Started off with exploratory data analysis. There are a total of 5 data files. train, test, new_merchant, merchant and historical transactions.

2) Then while exploring we found out some interestings things about the data, like outliers in the target, train and test have same distribution, missing data that needs to be taken care of etcetera.

3) While investigating the VIF value we can see that there are few variables that are correlated and the correlation matrix also confirms that. So we can remove those variables.

4) We also found out that the variable installment has value 999 which could be used to denote un-authourized transaction as the number of transaction with that value is un-authourized.

5) Then as far as feature engineering is concerned a total of around 330 features were created.

6) Some of the main features created are time based features like whether a day is a weekend, a purchase is made on a holiday etcetera. Then aggregate features like mean, finding sum, maximum and minimum value etcetera.

7) category_1/ category_2/ category_3, are subjected to one-hot conversion and findding the mean and sum for each column separately.

8) Since there are more than 330 features feature selection is done using recursive feature elimination. Finally after reccursive feature elimination we are left with 254 features. So for model building we can use these 254 features.

MODEL BUILDING

HYPERPARAMETER TUNING OF LGBM USING OPTUNA

Ref: https://github.com/optuna/optuna/blob/master/examples/lightgbm_simple.py

In [ ]:

```
import lightgbm
def objective(trial):

    lgbm_train = lightgbm.Dataset(train_df[train_cols], target, free_raw_data=False)

    params = {
              'objective': 'regression', 'metric': 'rmse',
              'verbosity': -1,  "learning_rate": 0.01,
              'device': 'cpu',  'seed': 326,
              'boosting_type': 'gbdt', 'n_jobs': 8,
              'num_leaves': trial.suggest_int('num_leaves', 16, 64), 'colsample_bytree':
trial.suggest_uniform('colsample_bytree', 0.001, 1),
              'subsample': trial.suggest_uniform('subsample', 0.001, 1), 'max_depth': tr
ial.suggest_int('max_depth', 1, 12),
              'reg_alpha': trial.suggest_uniform('reg_alpha', 0, 10), 'reg_lambda': tria
l.suggest_uniform('reg_lambda', 0, 10),
              'min_split_gain': trial.suggest_uniform('min_split_gain', 0, 10), 'min_chi
ld_weight': trial.suggest_uniform('min_child_weight', 0, 45),
              'min_data_in_leaf': trial.suggest_int('min_data_in_leaf', 16, 64)
              }


    folds = StratifiedKFold(n_splits=3, shuffle=True, random_state=326)

    clf = lightgbm.cv(params=params, train_set=lgbm_train, metrics=['rmse'], nfold=3, \
                      folds=folds.split(train_df[train_cols], outlier.values), \
                      num_boost_round=10000, early_stopping_rounds=200, verbose_eval=100
, seed=47)
    gc.collect()
    return clf['rmse-mean'][-1]

if __name__ == '__main__':
    study = optuna.create_study()
    study.optimize(objective, n_trials=100);gc.collect()
    print('Number of finished trials: {}'.format(len(study.trials)))
    print('Best trial:')
    trial = study.best_trial;gc.collect()
    print('  Value: {}'.format(trial.value))
    print('  Params: ')
    for key, value in trial.params.items():
        print('    {}: {}'.format(key, value))
```

```
Best trial:
  Value: 3.578537206836289
  Params:
    num_leaves: 60
    colsample_bytree: 0.35120050902766814
    subsample: 0.7640712594837535
    max_depth: 6
    reg_alpha: 4.054941002128766
    reg_lambda: 9.687137168616093
    min_split_gain: 2.160075593248262
    min_child_weight: 16.067553828016983
    min_data_in_leaf: 17
```

Training LGBM with features selected by RFE

In [ ]:

```
# Loading the features selected by RFE
feats = np.load('final_feat_rfe.npy', allow_pickle=True)
len(feats)
```

```
254
```

In [ ]:

```
%%time

param = {
    'objective'          : 'regression',
    'boosting_type'      : 'gbdt',
    'metric'             : 'rmse',
    'learning_rate'      : 0.01,
    'num_leaves': 60,
    'colsample_bytree': 0.35120050902766814,
    'subsample':0.7640712594837535,
    'max_depth': 6,
    'reg_alpha': 4.054941002128766,
    'reg_lambda': 9.687137168616093,
    'min_split_gain': 2.160075593248262,
    'min_child_weight': 16.067553828016983,
    'min_data_in_leaf': 17,
    'nthread'            : 8

}

#prepare fit model with cross-validation
folds = StratifiedKFold(n_splits=9, shuffle=True, random_state=2019)
oof = np.zeros(len(train_df))
predictions = np.zeros(len(test_df))


for fold_, (trn_idx, val_idx) in enumerate(folds.split(train_df,outliers.values)):
    print("fold {}".format(fold_))
    trn_data = lgb.Dataset(train_df.iloc[trn_idx][feats], label=target.iloc[trn_idx])
    val_data = lgb.Dataset(train_df.iloc[val_idx][feats], label=target.iloc[val_idx])

    num_round = 10000
    clf = lgb.train(param, trn_data, num_round, valid_sets = [trn_data, val_data], \
                    verbose_eval=200, early_stopping_rounds = 150)
    oof[val_idx] = clf.predict(train_df.iloc[val_idx][feats], num_iteration=clf.best_ite
ration)

    #make the predictions
    predictions = predictions + clf.predict(test_df[feats], num_iteration=clf.best_itera
tion) / folds.n_splits

print("".format(np.sqrt(mean_squared_error(oof, target))))
```

```
fold 0
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.619 valid_1's rmse: 3.66852
[400] training's rmse: 3.54656 valid_1's rmse: 3.64776
[600] training's rmse: 3.49872 valid_1's rmse: 3.64273
[800] training's rmse: 3.45995 valid_1's rmse: 3.64129
[1000] training's rmse: 3.42973 valid_1's rmse: 3.64027
[1200] training's rmse: 3.40295 valid_1's rmse: 3.63998
Early stopping, best iteration is:
[1165] training's rmse: 3.40708 valid_1's rmse: 3.63977
fold 1
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.61644 valid_1's rmse: 3.68745
[400] training's rmse: 3.54419 valid_1's rmse: 3.66546
[600] training's rmse: 3.4986 valid_1's rmse: 3.65978
[800] training's rmse: 3.46176 valid_1's rmse: 3.65753
[1000] training's rmse: 3.4296 valid_1's rmse: 3.65579
[1200] training's rmse: 3.40096 valid_1's rmse: 3.6559
Early stopping, best iteration is:
[1123] training's rmse: 3.41114 valid_1's rmse: 3.65548
fold 2
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.61792 valid_1's rmse: 3.68513
[400] training's rmse: 3.54608 valid_1's rmse: 3.66602
[600] training's rmse: 3.50038 valid_1's rmse: 3.65957
[800] training's rmse: 3.4654 valid_1's rmse: 3.65665
[1000] training's rmse: 3.43649 valid_1's rmse: 3.65527
[1200] training's rmse: 3.40847 valid_1's rmse: 3.65477
[1400] training's rmse: 3.38147 valid_1's rmse: 3.65416
```

```
[1400] training's rmse: 3.38147 valid_1's rmse: 3.65416
[1600] training's rmse: 3.35624 valid_1's rmse: 3.65408
[1800] training's rmse: 3.33395 valid_1's rmse: 3.65389
Early stopping, best iteration is:
[1717] training's rmse: 3.34262 valid_1's rmse: 3.65362
fold 3
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.6164 valid_1's rmse: 3.68381
[400] training's rmse: 3.54333 valid_1's rmse: 3.66433
[600] training's rmse: 3.49515 valid_1's rmse: 3.66058
[800] training's rmse: 3.45685 valid_1's rmse: 3.659
[1000] training's rmse: 3.42583 valid_1's rmse: 3.65837
[1200] training's rmse: 3.39745 valid_1's rmse: 3.65849
Early stopping, best iteration is:
[1164] training's rmse: 3.40187 valid_1's rmse: 3.65826
fold 4
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.61933 valid_1's rmse: 3.67733
[400] training's rmse: 3.54815 valid_1's rmse: 3.65243
[600] training's rmse: 3.49996 valid_1's rmse: 3.64593
[800] training's rmse: 3.4629 valid_1's rmse: 3.64184
[1000] training's rmse: 3.43187 valid_1's rmse: 3.64077
[1200] training's rmse: 3.40323 valid_1's rmse: 3.64094
Early stopping, best iteration is:
[1066] training's rmse: 3.42203 valid_1's rmse: 3.64038
fold 5
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.62181 valid_1's rmse: 3.66521
[400] training's rmse: 3.55076 valid_1's rmse: 3.63771
[600] training's rmse: 3.50497 valid_1's rmse: 3.6288
[800] training's rmse: 3.46923 valid_1's rmse: 3.62495
[1000] training's rmse: 3.44078 valid_1's rmse: 3.62195
[1200] training's rmse: 3.41471 valid_1's rmse: 3.6204
[1400] training's rmse: 3.39107 valid_1's rmse: 3.61951
[1600] training's rmse: 3.36826 valid_1's rmse: 3.61903
[1800] training's rmse: 3.3458 valid_1's rmse: 3.6182
[2000] training's rmse: 3.32349 valid_1's rmse: 3.61774
[2200] training's rmse: 3.30357 valid_1's rmse: 3.61682
[2400] training's rmse: 3.28373 valid_1's rmse: 3.61657
Early stopping, best iteration is:
[2424] training's rmse: 3.28066 valid_1's rmse: 3.61655
fold 6
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.61556 valid_1's rmse: 3.68425
[400] training's rmse: 3.54242 valid_1's rmse: 3.66396
[600] training's rmse: 3.4971 valid_1's rmse: 3.65854
[800] training's rmse: 3.46233 valid_1's rmse: 3.65535
[1000] training's rmse: 3.43191 valid_1's rmse: 3.65384
[1200] training's rmse: 3.40464 valid_1's rmse: 3.65354
[1400] training's rmse: 3.37809 valid_1's rmse: 3.65315
[1600] training's rmse: 3.35482 valid_1's rmse: 3.6528
[1800] training's rmse: 3.33441 valid_1's rmse: 3.65259
Early stopping, best iteration is:
[1753] training's rmse: 3.33903 valid_1's rmse: 3.65233
fold 7
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.61966 valid_1's rmse: 3.67713
[400] training's rmse: 3.54794 valid_1's rmse: 3.65513
[600] training's rmse: 3.50085 valid_1's rmse: 3.64814
[800] training's rmse: 3.46507 valid_1's rmse: 3.6444
[1000] training's rmse: 3.43408 valid_1's rmse: 3.64318
[1200] training's rmse: 3.40651 valid_1's rmse: 3.64184
[1400] training's rmse: 3.3818 valid_1's rmse: 3.64168
Early stopping, best iteration is:
[1294] training's rmse: 3.39408 valid_1's rmse: 3.64109
fold 8
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.61498 valid_1's rmse: 3.69348
[400] training's rmse: 3.54096 valid_1's rmse: 3.6775
[600] training's rmse: 3.49359 valid_1's rmse: 3.67539
Early stopping, best iteration is:
[639] training's rmse: 3.48555 valid_1's rmse: 3.6751
```

```
CPU times: user 41min 49s, sys: 28.5 s, total: 42min 18s
Wall time: 5min 33s
```

In [ ]:

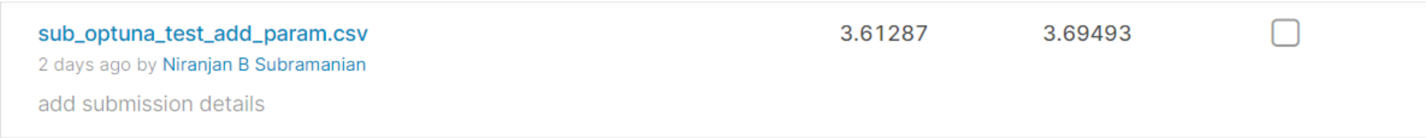
```
sub_df = pd.DataFrame({"card_id":card_id.values})
sub_df["target"] = predictions
sub_df.to_csv("sub_optuna_test_add_param.csv", index=False)
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/Colab Notebooks/ELO/MODEL/lgb_rfe.PNG")
```

```
Score in Kaggle
```

Out[ ]:



Training a LGBM model with all the features

In [ ]:

```
#selecting all the features except some basic ones which were used just to create new fea
tures
train_cols = [c for c in train_df.columns if c not in ['first_active_month', 'target','o
utliers','feature_1','feature_2','feature_3','refernce_month','hist_transac_purchase_date
_max', 'hist_transac_purchase_date_min', 'new_transac_purchase_date_max', 'new_transac_pu
rchase_date_min',\
'hist_purchase_date_last', 'hist_purchase_date_first', 'reference_month', 'hist_purchase_
a_date_last', 'hist_purchase_a_date_first', 'new_purchase_date_last', 'new_purchase_date_
first','card_id', \
'first_active_month','ref_first_month_diff_days']]
```

In [ ]:

```
%%time
param =  {
                    'colsample_bytree': 0.35120050902766814,
                    'subsample': 0.7640712594837535,
                    'max_depth': 6,
                    'reg_alpha': 4.054941002128766,
                    'reg_lambda': 9.687137168616093,
                    'min_split_gain': 2.160075593248262,
                    'min_child_weight': 16.067553828016983,
                    'min_data_in_leaf': 17,
                    'objective'          : 'regression',
                    'boosting_type'      : 'gbdt',
                    'metric'             : 'rmse',
                    'learning_rate'      : 0.01,
                    'num_leaves'         : 60,
                    'data_random_seed'   : 2019,
                    'max_bin'            : 255,
                    'nthread'            : 8
                }
#prepare fit model with cross-validation
folds = StratifiedKFold(n_splits=9, shuffle=True, random_state=2019)
oof = np.zeros(len(train_df))
predictions = np.zeros(len(test_df))
feature_importance_df = pd.DataFrame()
#run model

for fold_, (trn_idx, val_idx) in enumerate(
                                            folds.split(train_df,outliers.values)):
    #strLog = "fold {}".format(fold_)
```

```
    print("fold {}".format(fold_))
    trn_data = lgb.Dataset(train_df.iloc[trn_idx][train_cols], \
                           label=target.iloc[trn_idx])#, categorical_feature=cat)
    val_data = lgb.Dataset(train_df.iloc[val_idx][train_cols], \
                           label=target.iloc[val_idx])#, categorical_feature=cat)

    num_round = 10000
    clf = lgb.train(param, trn_data, num_round, valid_sets = [trn_data, val_data], \
                    verbose_eval=200, early_stopping_rounds = 150)
    oof[val_idx] = clf.predict(train_df.iloc[val_idx][train_cols], num_iteration=clf.bes
t_iteration)

    predictions += clf.predict(test_df[train_cols], num_iteration=clf.best_iteration) /
folds.n_splits

strRMSE = "".format(np.sqrt(mean_squared_error(oof, target)))
print(strRMSE)
```

```
fold 0
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.60827 valid_1's rmse: 3.65873
[400] training's rmse: 3.5373 valid_1's rmse: 3.64238
[600] training's rmse: 3.495 valid_1's rmse: 3.64029
[800] training's rmse: 3.46504 valid_1's rmse: 3.63953
Early stopping, best iteration is:
[761] training's rmse: 3.47071 valid_1's rmse: 3.63921
fold 1
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.60612 valid_1's rmse: 3.67788
[400] training's rmse: 3.53479 valid_1's rmse: 3.6613
[600] training's rmse: 3.4959 valid_1's rmse: 3.65731
[800] training's rmse: 3.46344 valid_1's rmse: 3.65547
[1000] training's rmse: 3.43454 valid_1's rmse: 3.65463
[1200] training's rmse: 3.40882 valid_1's rmse: 3.65402
[1400] training's rmse: 3.38446 valid_1's rmse: 3.65378
Early stopping, best iteration is:
[1351] training's rmse: 3.38928 valid_1's rmse: 3.65375
fold 2
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.60712 valid_1's rmse: 3.67644
[400] training's rmse: 3.53762 valid_1's rmse: 3.6597
[600] training's rmse: 3.49769 valid_1's rmse: 3.65532
[800] training's rmse: 3.47078 valid_1's rmse: 3.6533
[1000] training's rmse: 3.44522 valid_1's rmse: 3.65207
[1200] training's rmse: 3.4203 valid_1's rmse: 3.65166
Early stopping, best iteration is:
[1181] training's rmse: 3.42305 valid_1's rmse: 3.65151
fold 3
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.6058 valid_1's rmse: 3.674
[400] training's rmse: 3.53472 valid_1's rmse: 3.66002
[600] training's rmse: 3.49378 valid_1's rmse: 3.65728
[800] training's rmse: 3.46162 valid_1's rmse: 3.65628
[1000] training's rmse: 3.43362 valid_1's rmse: 3.65565
Early stopping, best iteration is:
[1016] training's rmse: 3.4314 valid_1's rmse: 3.65543
fold 4
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.60893 valid_1's rmse: 3.66734
[400] training's rmse: 3.53899 valid_1's rmse: 3.64485
[600] training's rmse: 3.49679 valid_1's rmse: 3.63944
[800] training's rmse: 3.46491 valid_1's rmse: 3.63717
Early stopping, best iteration is:
[840] training's rmse: 3.45931 valid_1's rmse: 3.63701
fold 5
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.6111 valid_1's rmse: 3.65403
[400] training's rmse: 3.54298 valid_1's rmse: 3.6297
[600] training's rmse: 3.50516 valid_1's rmse: 3.6233
[800] training's rmse: 3.47632 valid_1's rmse: 3.61992
[1000] training's rmse: 3.45046 valid_1's rmse: 3.61731
[1200] training's rmse: 3.42569 valid_1's rmse: 3.6158
```

```
[1400] training's rmse: 3.40072 valid_1's rmse: 3.61458
[1600] training's rmse: 3.37719 valid_1's rmse: 3.61422
Early stopping, best iteration is:
[1634] training's rmse: 3.3736 valid_1's rmse: 3.61407
fold 6
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.6052 valid_1's rmse: 3.68279
[400] training's rmse: 3.53572 valid_1's rmse: 3.66445
[600] training's rmse: 3.49794 valid_1's rmse: 3.66096
[800] training's rmse: 3.469 valid_1's rmse: 3.65969
[1000] training's rmse: 3.44296 valid_1's rmse: 3.65872
[1200] training's rmse: 3.4178 valid_1's rmse: 3.6579
[1400] training's rmse: 3.39336 valid_1's rmse: 3.65725
[1600] training's rmse: 3.36932 valid_1's rmse: 3.65686
Early stopping, best iteration is:
[1554] training's rmse: 3.37495 valid_1's rmse: 3.65672
fold 7
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.60758 valid_1's rmse: 3.67263
[400] training's rmse: 3.53704 valid_1's rmse: 3.65126
[600] training's rmse: 3.49568 valid_1's rmse: 3.64661
[800] training's rmse: 3.46609 valid_1's rmse: 3.64436
[1000] training's rmse: 3.43758 valid_1's rmse: 3.64315
[1200] training's rmse: 3.41167 valid_1's rmse: 3.6427
[1400] training's rmse: 3.38785 valid_1's rmse: 3.64207
Early stopping, best iteration is:
[1389] training's rmse: 3.38945 valid_1's rmse: 3.64201
fold 8
Training until validation scores don't improve for 150 rounds
[200] training's rmse: 3.60405 valid_1's rmse: 3.68792
[400] training's rmse: 3.53251 valid_1's rmse: 3.67487
[600] training's rmse: 3.48771 valid_1's rmse: 3.67223
[800] training's rmse: 3.45568 valid_1's rmse: 3.67198
Early stopping, best iteration is:
[812] training's rmse: 3.45382 valid_1's rmse: 3.67182

CPU times: user 55min 14s, sys: 36.4 s, total: 55min 50s
Wall time: 7min 21s
```

In [ ]:

```python
sub_df = pd.DataFrame({"card_id":card_id.values})
sub_df["target"] = predictions
sub_df.to_csv("sub_lgb_323_feat.csv", index=False)
```

In [ ]:

```python
import pickle
pickle.dump(clf, open('lgb_final_323_tune.sav', 'wb'))
```

In [ ]:

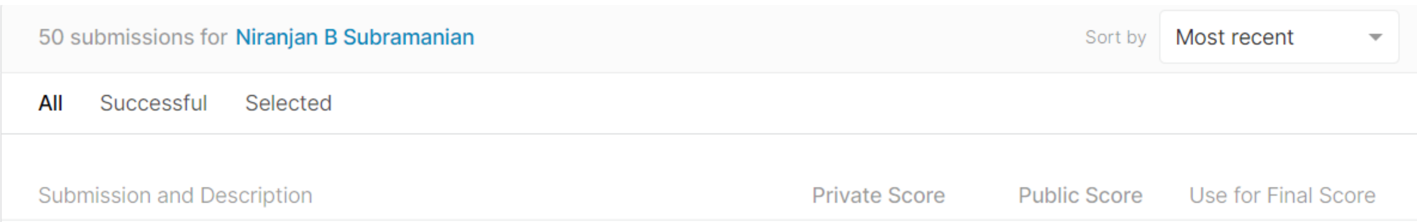
```python
np.save('oof_lgb_323_feat.npy', oof)
np.save('pred_lgb_323_feat.npy', predictions)
```

In [ ]:

```python
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/Colab Notebooks/ELO/MODEL/lgb final score.PNG")
```

Score in Kaggle

Out[ ]:



sub_lgb_tuned_323_feat_no_bsun_rerun.csv 2 minutes ago by Niranjan B Subramanian add submission details	3.61084	3.69359	☐

Using all the features does increase the performance of the model dramatically. The score went from 3.61287 to 3.61084 which is a huge improvement.

So, we'll use all the features to train the XGB model.

XGBOOST

HYPERPARAMETER TUNING USING RANDOMIZED SEARCHCV

In [ ]:

```
%%time
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint as sp_randint
import xgboost as xgb
from sklearn.model_selection import KFold, StratifiedKFold
from sklearn.metrics import mean_squared_error

xgb = xgb.XGBRegressor(learning_rate=0.01, n_estimators=100, objective= 'reg:linear', \
                       eval_metric ='rmse', silent=True, nthread=1, tree_method='gpu_his
t')

parameters = {
         'num_boost_round': [10, 25, 50], 'eta': [0.05, 0.1, 0.3],
         'max_depth': [3, 4, 5, 6,], 'subsample':[i/10.0 for i in range(6,10)],
         'colsample_bytree':[i/10.0 for i in range(6,10)], "min_samples_split": sp_randi
nt(2, 11),
         "min_samples_leaf": sp_randint(1, 11), "min_child_weight": range(1,6,2),
         'gamma':[i/10.0 for i in range(0,5)], 'reg_alpha':[1e-5, 1e-2, 0.1, 1, 100]
     }

random_search = RandomizedSearchCV(xgb, param_distributions=parameters, \
                cv=StratifiedKFold(n_splits=9, random_state=42).split(train_df,outliers.
values),\
                 n_jobs=-1, n_iter=30, verbose=3)

random_search.fit(train_df, target)
```

```
Fitting 9 folds for each of 30 candidates, totalling 270 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  28 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 124 tasks      | elapsed:  5.4min
[Parallel(n_jobs=-1)]: Done 270 out of 270 | elapsed: 11.5min finished
```

```
CPU times: user 4.76 s, sys: 8.2 s, total: 13 s
Wall time: 11min 33s
```

In [ ]:

```
random_search.best_params_
```

Out[ ]:

```
{'colsample_bytree': 0.9,
 'eta': 0.3,
 'gamma': 0.3,
 'max_depth': 6,
 'min_child_weight': 1,
 'min_samples_leaf': 7,
 'min_samples_split': 7,
 'num_boost_round': 25,
 'reg_alpha': 1e-05,
 'subsample': 0.8}
```

XGBOOST WITH BEST PARAMETERS

In [ ]:

```
import xgboost as xgb
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

xgb_params = {
              'eta': 0.3, 'max_depth': 6, 'subsample': 0.8, 'colsample_bytree': 0.9, \
              'learning_rate':0.01, 'gamma':0.3, 'min_samples_leaf' : 7, 'min_samples_sp
lit': 7, \
              'num_boost_round': 25, 'reg_alpha': 1e-05,'objective': 'reg:linear', 'eval
_metric': 'rmse', \
              'silent': True, 'tree_method':'gpu_hist'
              }


FOLDs = KFold(n_splits=9, shuffle=True, random_state=1989)

oof_xgb = np.zeros(len(train_df))
predictions_xgb = np.zeros(len(test_df))


for fold_, (trn_idx, val_idx) in enumerate(FOLDs.split(train_df,outliers.values)):
    trn_data = xgb.DMatrix(data=train_df.iloc[trn_idx], label=target.iloc[trn_idx])
    val_data = xgb.DMatrix(data=train_df.iloc[val_idx], label=target.iloc[val_idx])
    watchlist = [(trn_data, 'train'), (val_data, 'valid')]
    print("xgb " + str(fold_) + "-" * 50)
    num_round = 10000
    xgb_model = xgb.train(xgb_params, trn_data, num_round, watchlist, \
                      early_stopping_rounds=200, verbose_eval=200)
    oof_xgb[val_idx] = xgb_model.predict(xgb.DMatrix(train_df.iloc[val_idx]), \
                                     ntree_limit=xgb_model.best_ntree_limit+50)

    predictions_xgb = predictions_xgb + xgb_model.predict(xgb.DMatrix(test_df), \
                            ntree_limit=xgb_model.best_ntree_limit+50) / FOLDs.n_splits

np.sqrt(mean_squared_error(oof_xgb, target))
```

```
xgb 0--------------------------------------------------
[0] train-rmse:3.92744 valid-rmse:4.10747
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.56353 valid-rmse:3.85181
[400] train-rmse:3.47456 valid-rmse:3.8299
[600] train-rmse:3.41372 valid-rmse:3.82396
[800] train-rmse:3.36164 valid-rmse:3.82258
[1000] train-rmse:3.31169 valid-rmse:3.82155
[1200] train-rmse:3.26657 valid-rmse:3.82073
Stopping. Best iteration:
[1107] train-rmse:3.28661 valid-rmse:3.82069

xgb 1--------------------------------------------------
[0] train-rmse:3.96001 valid-rmse:3.84891
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.59158 valid-rmse:3.59259
[400] train-rmse:3.5022 valid-rmse:3.5818
[600] train-rmse:3.43727 valid-rmse:3.57979
[800] train-rmse:3.38594 valid-rmse:3.57893
Stopping. Best iteration:
[759] train-rmse:3.39634 valid-rmse:3.57865

xgb 2--------------------------------------------------
[0] train-rmse:3.94451 valid-rmse:3.9735
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
```

```
[200] train-rmse:3.57485 valid-rmse:3.74218
[400] train-rmse:3.48543 valid-rmse:3.73009
[600] train-rmse:3.42348 valid-rmse:3.72797
[800] train-rmse:3.37005 valid-rmse:3.72648
[1000] train-rmse:3.32107 valid-rmse:3.72563
[1200] train-rmse:3.27229 valid-rmse:3.72491
Stopping. Best iteration:
[1119] train-rmse:3.29252 valid-rmse:3.72446

xgb 3--------------------------------------------------
[0] train-rmse:3.9637 valid-rmse:3.81587
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.58891 valid-rmse:3.58358
[400] train-rmse:3.50244 valid-rmse:3.57418
[600] train-rmse:3.44162 valid-rmse:3.57384
[800] train-rmse:3.38834 valid-rmse:3.57317
Stopping. Best iteration:
[706] train-rmse:3.41283 valid-rmse:3.5726

xgb 4--------------------------------------------------
[0] train-rmse:3.93859 valid-rmse:4.02174
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.57423 valid-rmse:3.75775
[400] train-rmse:3.48798 valid-rmse:3.73531
[600] train-rmse:3.42976 valid-rmse:3.72973
[800] train-rmse:3.37856 valid-rmse:3.72804
[1000] train-rmse:3.32868 valid-rmse:3.72708
[1200] train-rmse:3.2834 valid-rmse:3.72724
Stopping. Best iteration:
[1027] train-rmse:3.32293 valid-rmse:3.72696

xgb 5--------------------------------------------------
[0] train-rmse:3.94845 valid-rmse:3.94206
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.58507 valid-rmse:3.66558
[400] train-rmse:3.49639 valid-rmse:3.64648
[600] train-rmse:3.43706 valid-rmse:3.6417
[800] train-rmse:3.38336 valid-rmse:3.63947
[1000] train-rmse:3.33181 valid-rmse:3.63882
Stopping. Best iteration:
[986] train-rmse:3.33501 valid-rmse:3.63857

xgb 6--------------------------------------------------
[0] train-rmse:3.94216 valid-rmse:3.99105
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.57788 valid-rmse:3.72274
[400] train-rmse:3.48961 valid-rmse:3.7034
[600] train-rmse:3.43103 valid-rmse:3.69867
[800] train-rmse:3.37869 valid-rmse:3.69745
[1000] train-rmse:3.33136 valid-rmse:3.69728
[1200] train-rmse:3.28411 valid-rmse:3.69726
Stopping. Best iteration:
[1049] train-rmse:3.31964 valid-rmse:3.69695

xgb 7--------------------------------------------------
[0] train-rmse:3.95164 valid-rmse:3.91673
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.58451 valid-rmse:3.66445
[400] train-rmse:3.49464 valid-rmse:3.6474
[600] train-rmse:3.43058 valid-rmse:3.64373
[800] train-rmse:3.37879 valid-rmse:3.64244
[1000] train-rmse:3.33089 valid-rmse:3.64237
```

```
Stopping. Best iteration:
[945] train-rmse:3.34398 valid-rmse:3.64187

xgb 8------------------------------------------------
[0] train-rmse:3.95265 valid-rmse:3.90925
Multiple eval metrics have been passed: 'valid-rmse' will be used for early stopping.

Will train until valid-rmse hasn't improved in 200 rounds.
[200] train-rmse:3.58951 valid-rmse:3.64716
[400] train-rmse:3.50098 valid-rmse:3.62662
[600] train-rmse:3.44018 valid-rmse:3.62196
[800] train-rmse:3.38734 valid-rmse:3.62111
[1000] train-rmse:3.33851 valid-rmse:3.61945
[1200] train-rmse:3.29118 valid-rmse:3.61854
Stopping. Best iteration:
[1131] train-rmse:3.30726 valid-rmse:3.61826
```

Out[ ]:

```
3.6699344183630043
```

In [ ]:

```
import pickle
pickle.dump(xgb_model, open('/content/drive/My Drive/case study/upload 15mis/xgb_final_32
3_tune.sav', 'wb'))
```

In [ ]:

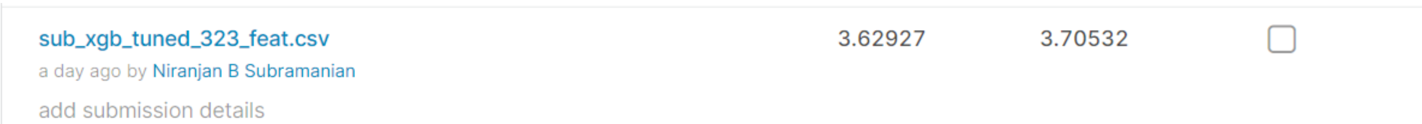
```
np.save('oof_xgb_323_feat.npy', oof_xgb)
np.save('pred_xgb_323_feat.npy', predictions_xgb)
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/Colab Notebooks/ELO/MODEL/xgb final.PNG")
```

Score in Kaggle

Out[ ]:



The XGBOOST model is not as good as the LightGBM

STACKING

SIMPLE BLENDING WITH 80% WEIGHT TO LGBM PREDICTIONS AND 20% WEIGHT TO XGB PREDICTIONS

In [ ]:

```
fianl_pred =0.8*pred_lgb + 0.2*pred_xgb
sub_df = pd.DataFrame({"card_id":card_id.values})
sub_df["target"] = fianl_pred
sub_df.to_csv("0.8lgb_0.2xgb_blend.csv", index=False)
!cp 0.8lgb_0.2xgb_blend.csv '/content/drive/My Drive/case study/upload 15mis/'
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/Colab Notebooks/ELO/MODEL/0.8LGB_00.2XGB.PNG")
```

Score in Kaggle

Out[ ]:

0.8lgb_0.2xgb_blend.csv 7 hours ago by Niranjan B Subramanian add submission details	3.61184	3.69340	☐

Simple weight based blending of LGBM AND XGB is better than the latter score alone.

In the following we'll create a simple bagging model then will be stacked with XGBOOST as a metalearner.

First the data is split into 80-20 for train and test. We'll call this x_test. This will be used to evaluate our model.

Then again we split the 80% data into 50-50 x1_train and x1_test.

The x1_train will be used to train the model and x1_test will be used to tune the hyperparameters.

Once we have the best model with best hyperparameter we'll train the model on the whole dataset and test on the kaggle submission file.

In [ ]:

```
%%time
from tqdm import tqdm
from sklearn.linear_model import Ridge
#number of base learners
bl = [100, 200, 300, 500]
#list to score rmse
score = []
#loop through all the no of base learners
for estimator in tqdm(bl):

  #list to store the base leaarners
  base_learners = []
  #loop through the list of base learners
  for i in range(estimator):
    max = len(x1_train)

    #create bootstrap data by randomly choosing pts with replacement
    idx = np.random.randint(0, max, size=50000)
    sample_x = x1_train.iloc[idx]
    sample_y = y1_train.iloc[idx]

    #instantiate the classifier
    clf = DecisionTreeRegressor(max_depth=5)

    #fit the classifier with the sampled data
    clf.fit(sample_x, sample_y)

    #append the classifier to the list of base learners
    base_learners.append(clf)

  #store the predictions of x1_test 50% of data
  df = pd.DataFrame()
  for idx, dt in enumerate(base_learners):
    pred = dt.predict(x1_test)
    df['clf{}'.format(idx+1)] = pred

  #training metalearner XGB
  num_round = 5000
  train = xgb.DMatrix(data=df, label=y1_test)
  watchlist=train

  xgb_model = xgb.train(train, num_round, verbose_eval=200)

  #predict on the test data 20% of the original data
```

```
  test = pd.DataFrame()
  for idx, dt in enumerate(base_learners):
    pred = dt.predict(x_test)
    test['clf{}'.format(idx+1)] = pred

  #calculate the RMSE for the x_test
  prediction = xgb_model.predict(xgb.DMatrix(test))
  s = np.sqrt(mean_squared_error(prediction, y_test))
  print('RMSE:', s)

  #append the RMSE to the list score
  score.append(s)
```

```
  0%|          | 0/4 [00:00<?, ?it/s]
 25%|██▌       | 1/4 [09:21<28:03, 561.23s/it]
```

```
RMSE: 3.74588409339133
```

```
 50%|█████     | 2/4 [28:02<24:18, 729.25s/it]
```

```
RMSE: 3.7547066784900025
```

```
 75%|███████▌  | 3/4 [56:09<16:56, 1016.47s/it]
```

```
RMSE: 3.768748080992725
```

```
100%|██████████| 4/4 [1:42:47<00:00, 1541.78s/it]
```

```
RMSE: 3.7888090596986275
CPU times: user 1h 42min 39s, sys: 4.38 s, total: 1h 42min 43s
Wall time: 1h 42min 47s
```

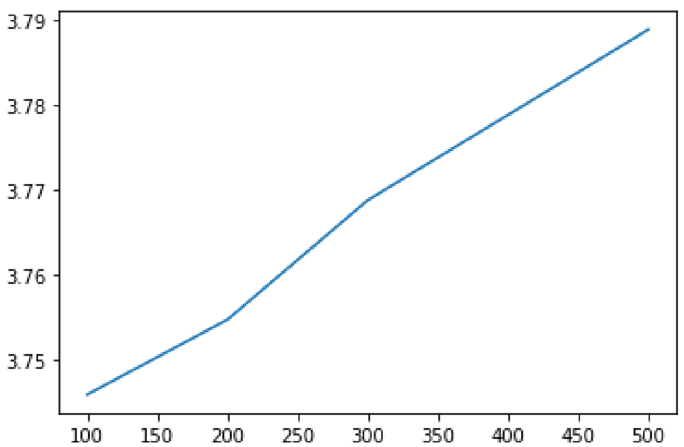
Let's plot the number of estimators on the x-axis against the RMSE values on the y-axis.

In [ ]:

```
import matplotlib.pyplot as plt
plt.plot(bl, score)
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x7f6ac5252898>]
```



Looks like 100 base learners has the lowest RMSE. So, we'll use 100 base learners to build the model.

In [ ]:

```
#number of base learners
bl = 100
#list to store all the base learners
base_learners = []
#loop through the no of base learners
for i in range(bl):
    max = len(x1_train)
    idx = np.random.randint(0, max, size=50000)
    #create bootstrap data by randomly choosing pts with replacement
```

```python
    sample_x = x1_train.iloc[idx]
    sample_y = y1_train.iloc[idx]
    #instantiate a classifier
    clf = DecisionTreeRegressor(max_depth=5)
    #fit the classifier on the bootstrapped data
    clf.fit(sample_x, sample_y)
    #add the base learner to the list
    base_learners.append(clf)

#create a DF to store the predictions
df = pd.DataFrame()
#loop through all the base learners and predict the test data 50%
for idx, dt in enumerate(base_learners):
  pred = dt.predict(x1_test)
  df['clf{}'.format(idx+1)] = pred
df.head()
```

Out[ ]:

	clf1	clf2	clf3	clf4	clf5	clf6	clf7	clf8	clf9	clf10	clf11	clf12	
0	0.494199	0.520884	0.215122	0.507670	0.548381	-0.101815	0.333711	0.427894	0.326794	0.508397	0.801646	0.707188	0.4
1	-0.833065	-0.928526	-0.648493	-0.441737	-0.314416	-0.788799	-0.279563	-0.363044	-0.709027	-0.309565	-1.159525	-1.189003	0.
2	0.494199	0.520884	0.036589	0.507670	0.055180	-0.190275	0.333711	0.427894	0.326794	0.508397	0.801646	0.707188	0.4
3	-0.919025	-0.554937	-0.908352	-1.323837	-0.518661	-0.125638	-0.239234	-1.251950	-1.065314	-1.405810	-0.216282	-0.243376	0.
4	-0.271775	-0.374137	0.036589	-0.441737	-0.314416	-0.190275	-0.279563	-0.363044	-0.235112	-0.309565	-1.159525	-1.189003	0.

5 rows × 100 columns

Now we'll train the meta model which is XGBOOST.

In [ ]:

```python
import xgboost as xgb
xgb_params = {'eta': 0.001, 'max_depth': 7, 'subsample': 0.8, 'colsample_bytree': 0.8,
          'objective': 'reg:linear', 'eval_metric': 'rmse', 'silent': True, 'tree_method
':'hist'}

num_round = 5000
train = xgb.DMatrix(data=df, label=y1_test)
watchlist=train

xgb_model = xgb.train(xgb_params, train, num_round, verbose_eval=200)

#make predictions on the test set 20% data
test = pd.DataFrame()
for idx, dt in enumerate(base_learners):
  pred = dt.predict(x_test)
  test['clf{}'.format(idx+1)] = pred
```

In [ ]:

```python
#calculate the RMSE value
prediction = xgb_model.predict(xgb.DMatrix(test))
s = np.sqrt(mean_squared_error(prediction, y_test))
print('RMSE:', s)
score.append(s)
```

```
RMSE: 3.7587417865524997
```

making final prediction - training on whole data

In [ ]:

```
train_df = train_df.fillna(0)
test_df = test_df.fillna(0)
```

In [ ]:

```
#number of base learners
bl = 100
#list to store all the base learners
base_learners = []
#loop through the no of base learners
for i in range(bl):
    #training on the whole data
    max = len(train_df[train_cols])
    idx = np.random.randint(0, max, size=50000)
    #create bootstrap data by randomly choosing pts with replacement
    sample_x = train_df[train_cols].iloc[idx]
    sample_y = target.iloc[idx]
    #instantiate a classifier
    clf = DecisionTreeRegressor(max_depth=5)
    #fit the classifier on the bootstrapped data
    clf.fit(sample_x, sample_y)
    #add the base learner to the list
    base_learners.append(clf)

#create a DF to store the predictions
df = pd.DataFrame()
#loop through all the base learners all predict the test data
for idx, dt in enumerate(base_learners):
  pred = dt.predict(train_df[train_cols])
  df['clf{}'.format(idx+1)] = pred
df.head()
```

Out[ ]:

	clf1	clf2	clf3	clf4	clf5	clf6	clf7	clf8	clf9	clf10	clf11	clf12	
0	-0.225156	-0.320996	-0.246741	-0.223829	-0.266014	-0.407913	-0.186145	-0.592907	-0.241513	-0.315529	-0.296225	-0.436041	0.
1	-1.857445	-1.999245	0.143004	0.089154	-1.243992	0.099734	-2.228633	-0.105974	-1.807319	-2.360468	0.002989	0.087405	1.
2	0.448579	0.282516	0.645393	-0.001087	0.612684	0.362219	0.566765	0.141078	0.422373	0.509077	0.268495	0.087405	0.
3	-0.225156	0.042503	-0.246741	-0.223829	-0.266014	-0.407913	-0.186145	-0.105974	-0.241513	-0.029063	-0.296225	-0.107935	0.
4	-0.806985	-0.543710	-0.246741	-0.223829	-0.266014	-0.407913	-0.827029	-0.270912	-0.241513	-0.315529	-0.296225	-0.107935	0.

5 rows × 100 columns

In [ ]:

```
xgb_params = {'eta': 0.001, 'max_depth': 7, 'subsample': 0.8, 'colsample_bytree': 0.8,
          'objective': 'reg:linear', 'eval_metric': 'rmse', 'silent': True, 'tree_method
':'hist'}

num_round = 5000
train = xgb.DMatrix(data=df, label=target)
watchlist=train

xgb_model = xgb.train(xgb_params, train, num_round, verbose_eval=200)
```

In [ ]:

```
#kaggle submission test data
df = pd.DataFrame()
```

```
for idx, dt in enumerate(base_learners):
  pred = dt.predict(test_df[train_cols])
  df['clf{}'.format(idx+1)] = pred
df.head()
```

Out[ ]:

	clf1	clf2	clf3	clf4	clf5	clf6	clf7	clf8	clf9	clf10	clf11	clf12	
0	-1.857445	-1.999245	-2.267977	-2.455072	-3.008357	-4.591418	-2.228633	-3.308297	-1.807319	-2.360468	-1.000437	-3.972314	-5.
1	-0.225156	0.042503	-0.246741	-0.223829	-0.266014	-0.407913	-0.186145	-0.270912	-0.241513	-0.315529	-0.296225	-0.107935	-0.
2	-0.806985	-1.283993	-1.108650	-0.839833	-0.266014	-0.407913	-0.827029	-0.901260	-0.890726	-1.278321	-0.929387	-1.165142	-0.
3	-0.225156	0.042503	-0.246741	-0.223829	-0.266014	-0.407913	-0.186145	-0.270912	-0.241513	-0.315529	-0.296225	-0.436041	-0.
4	-0.806985	-0.543710	-1.108650	-0.839833	-1.004097	-0.407913	-0.827029	-0.901260	-0.890726	-1.430156	-0.929387	-1.514833	-0.

5 rows × 100 columns

In [ ]:

```
#save the predictions
p = xgb_model.predict(xgb.DMatrix(df))
sub_df = pd.DataFrame({"card_id":test_df['card_id'].values})
sub_df["target"] = p
sub_df.to_csv("new_arch_xgb_trfuda.csv", index=False)
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/case study/upload 15mis/new_arch_xgb.PNG")
```

Score in Kaggle

Out[ ]:

Submission and Description	Private Score	Public Score	Use for Final Score
new_arch_xgb_100_trfuda.csv just now by Niranjan B Subramanian add submission details	3.68270	3.76989	☐

The score gets worsen. Let's try a stacked model with ridge as a metalearner.

STACKED MODEL WITH RIDGE REGRESSION AS META LEARNER

HYPERPARAMETER TUNING THE RIDGE

In [ ]:

```
#loading the predictions of the trained lgb and xgb models to feed as the input to the me
ta learner
oof_xgb = np.load('/content/drive/My Drive/case study/upload 15mis/oof_xgb_323_feat.npy')
pred_xgb = np.load('/content/drive/My Drive/case study/upload 15mis/pred_xgb_323_feat.npy
')
oof_lgb = np.load('/content/drive/My Drive/case study/upload 15mis/oof_lgb_tuned_323_fina
l_stack.npy')
pred_lgb = np.load('/content/drive/My Drive/case study/upload 15mis/pred_lgb_tuned_323.np
y_final_stack.npy')
```

```
train_stack = np.vstack([oof_xgb, oof_lgb]).transpose()
test_stack = np.vstack([pred_xgb, pred_lgb]).transpose()
```

In [ ]:

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

clf = Ridge()
alphas = np.array([1,0.1,0.01,0.001,0.0001,0])

grid_search = GridSearchCV(clf, param_grid=dict(alpha=alphas), \
                n_jobs=-1, verbose=3)

grid_search.fit(train_stack, target)
```

```
Fitting 5 folds for each of 6 candidates, totalling 30 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=-1)]: Done  30 out of  30 | elapsed:    2.0s finished
```

Out[ ]:

```
GridSearchCV(cv=None, error_score=nan,
             estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=None, normalize=False, random_state=None,
                             solver='auto', tol=0.001),
             iid='deprecated', n_jobs=-1,
             param_grid={'alpha': array([1.e+00, 1.e-01, 1.e-02, 1.e-03, 1.e-04, 0.e+00])
},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=3)
```

In [ ]:

```
grid_search.best_estimator_
```

Out[ ]:

```
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

TRAINING WITH BEST HYPERPARAMETER

In [ ]:

```
#loading the predictions of the trained lgb and xgb models to feed as the input to the me
ta learner
oof_xgb = np.load('/content/drive/My Drive/case study/upload 15mis/oof_xgb_323_feat.npy')
pred_xgb = np.load('/content/drive/My Drive/case study/upload 15mis/pred_xgb_323_feat.npy
')
oof_lgb = np.load('/content/drive/My Drive/case study/upload 15mis/oof_lgb_tuned_323_fina
l_stack.npy')
pred_lgb = np.load('/content/drive/My Drive/case study/upload 15mis/pred_lgb_tuned_323.np
y_final_stack.npy')

train_stack = np.vstack([oof_xgb, oof_lgb]).transpose()
test_stack = np.vstack([pred_xgb, pred_lgb]).transpose()
```

In [ ]:

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import StratifiedKFold
folds = StratifiedKFold(n_splits=5, shuffle=True, random_state=15)
oof_stacked = np.zeros(train_stack.shape[0])
predictions_stacked = np.zeros(test_stack.shape[0])

for fold_, (trn_idx, val_idx) in enumerate(folds.split(train_stack, outliers.values)):
    print("Starting fold n={}".format(fold_))
    trn_data, trn_y = train_stack[trn_idx], target.iloc[trn_idx].values
```

```
    val_data, val_y = train_stack[val_idx], target.iloc[val_idx].values

    clf = Ridge(alpha=1)
    clf.fit(trn_data, trn_y)

    oof_stacked[val_idx] = clf.predict(val_data)
    predictions_stacked = predictions_stacked + clf.predict(test_stack) / folds.n_splits

sub_df = pd.DataFrame({"card_id":card_id.values})
sub_df["target"] = predictions_stacked
sub_df.to_csv("sub_full_stack_final_xgb_lgb.csv", index=False)
!cp sub_full_stack_final_xgb_lgb.csv '/content/drive/My Drive/case study/upload 15mis/'
```

```
Starting fold n=0
Starting fold n=1
Starting fold n=2
Starting fold n=3
Starting fold n=4
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/Colab Notebooks/ELO/MODEL/stack final.PNG")
```

```
Score in Kaggle
```

Out[ ]:

51 submissions for Niranjan B Subramanian　　Sort by Most recent

All　Successful　Selected

Submission and Description	Private Score	Public Score	Use for Final Score
sub_full_stack_final_xgb_lgb.csv 10 minutes ago by Niranjan B Subramanian add submission details	3.61046	3.69093	☐

In [ ]:

```

```

Neural Network based models

ANN 5 Layer Dropout

In [ ]:

```
from sklearn.datasets import load_boston
from keras.models import Sequential
from keras.layers import Dense, Conv1D, Flatten, Dropout, MaxPooling1D, BatchNormalizati
on
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from keras import backend as K
from keras.callbacks import EarlyStopping

early_stop = EarlyStopping(monitor='loss',patience=6, verbose=1, mode='auto')

def root_mean_squared_error(y_true, y_pred):
        return K.sqrt(K.mean(K.square(y_pred - y_true)))

model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(train_df[train_cols].shape[1],)))
model.add(Dense(32, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(16, activation='relu'))
```

```
model.add(BatchNormalization())
model.add(Dense(8, activation='relu'))
model.add(Dense(1))
model.compile(loss=root_mean_squared_error, optimizer="adam")
```

In [ ]:

```
model.fit(train_df[train_cols], target, batch_size=64, epochs=50, verbose=2, callbacks=[
early_stop])
```

```
Epoch 1/50
 - 14s - loss: 3.3997
Epoch 2/50
 - 14s - loss: 3.4025
Epoch 3/50
 - 14s - loss: 3.3988
Epoch 4/50
 - 14s - loss: 3.4072
Epoch 5/50
 - 14s - loss: 3.3868
Epoch 6/50
 - 14s - loss: 3.3909
Epoch 7/50
 - 14s - loss: 3.4028
Epoch 8/50
 - 14s - loss: 3.4082
Epoch 9/50
 - 14s - loss: 3.3993
Epoch 10/50
 - 14s - loss: 3.3934
Epoch 11/50
 - 14s - loss: 3.4043
Epoch 00011: early stopping
```

Out[ ]:

```
<keras.callbacks.callbacks.History at 0x7fa491911fd0>
```

In [ ]:

```
ypred = model.predict(test_df[train_cols])
```

In [ ]:

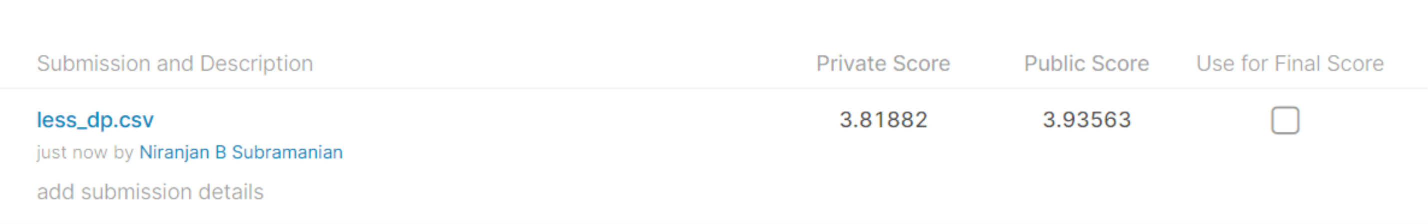
```
sub_df = pd.DataFrame({"card_id":test_df['card_id'].values})
sub_df["target"] = ypred
sub_df.to_csv("less_dp.csv", index=False)
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/case study/upload 15mis/less_Dp.PNG")
```

```
Score in Kaggle
```

Out[ ]:



Submission and Description	Private Score	Public Score	Use for Final Score
less_dp.csv just now by Niranjan B Subramanian add submission details	3.81882	3.93563	

Some what similar performance to the previous model even after increasing the layers and reducing the dropout rate.

Convolutional 1D Model:

In [ ]:

```
train_cnn = train_df[train_cols].values.reshape(train_df[train_cols].shape[0], train_df[
train_cols].shape[1], 1)
test_cnn = test_df[train_cols].values.reshape(test_df[train_cols].shape[0], test_df[trai
n_cols].shape[1], 1)
```

In [ ]:

```
train_cnn.shape
```

```
(201917, 323, 1)
```

In [ ]:

```
from sklearn.datasets import load_boston
from keras.models import Sequential
from keras.layers import Dense, Conv1D, Flatten, Dropout, MaxPooling1D
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from keras import backend as K
from keras.callbacks import EarlyStopping

early_stop = EarlyStopping(monitor='loss',patience=4, verbose=1, mode='auto')

def root_mean_squared_error(y_true, y_pred):
        return K.sqrt(K.mean(K.square(y_pred - y_true)))

model = Sequential()

model.add(Conv1D(64, 2, activation="relu", input_shape=(323, 1)))
model.add(Conv1D(32, 2, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.6))

model.add(Conv1D(16, 2, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Dropout(0.5))
model.add(Flatten())

model.add(Dense(64, activation="relu"))
model.add(Dense(1))

model.compile(loss=root_mean_squared_error, optimizer="adam")
```

In [ ]:

```
model.fit(train_cnn, target, batch_size=12, epochs=30, verbose=2, callbacks=[early_stop]
)
```

```
Epoch 1/30
16827/16827 - 96s - loss: 302.4445
Epoch 2/30
16827/16827 - 96s - loss: 2.7283
Epoch 3/30
16827/16827 - 96s - loss: 2.8471
Epoch 4/30
16827/16827 - 94s - loss: 2.6732
Epoch 5/30
16827/16827 - 97s - loss: 2.8528
Epoch 6/30
16827/16827 - 96s - loss: 2.6224
Epoch 7/30
16827/16827 - 94s - loss: 2.7189
Epoch 8/30
16827/16827 - 94s - loss: 2.6761
Epoch 9/30
16827/16827 - 96s - loss: 2.6309
Epoch 10/30
16827/16827 - 94s - loss: 2.6834
Epoch 00010: early stopping
```

Out[ ]:

```
<tensorflow.python.keras.callbacks.History at 0x7f5b42049f10>
```

In [ ]:

```
ypred = model.predict(test_cnn)
sub_df = pd.DataFrame({"card_id":test_df['card_id'].values})
sub_df["target"] = ypred
sub_df.to_csv("cnn_30.csv", index=False)
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/case study/upload 15mis/cnn.PNG")
```

```
Score in Kaggle
```

Out[ ]:

Submission and Description	Private Score	Public Score	Use for Final Score
cnn_30.csv just now by Niranjan B Subramanian add submission details	3.77332	3.89143	☐

CNN + LSTM

with maxpooling

In [ ]:

```
from keras.layers import Embedding
from keras.models import Sequential
from keras.layers import Dense, Conv1D, Flatten, Dropout, MaxPooling1D
from keras.layers import BatchNormalization, LSTM
from sklearn.metrics import mean_squared_error
from keras import backend as K
from keras.callbacks import EarlyStopping, ModelCheckpoint

def root_mean_squared_error(y_true, y_pred):
        return K.sqrt(K.mean(K.square(y_pred - y_true)))

filepath = '/content/drive/My Drive/case study/upload 15mis/weight/weights-{epoch:02d}-{l
oss:.2f}.hdf5'
early_stop = EarlyStopping(monitor='loss',patience=6, verbose=1, mode='auto')
model_ckpt = ModelCheckpoint(monitor='loss', save_best_only=True, verbose=1, mode='auto'
,filepath=filepath)

model = Sequential()
model.add(Embedding(324, 128, input_length=324))
model.add(Dropout(0.25))
model.add(Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(MaxPooling1D(2))

model.add(LSTM(50))
model.add(Dense(1))
model.compile(optimizer='adam', loss=root_mean_squared_error)
```

```
Using TensorFlow backend.
```

In [ ]:

```
model.fit(train_df[train_cols], target, batch_size=2048, epochs=50, callbacks=[model_ckp
t, early_stop])
```

```
Epoch 1/50
201917/201917 [==============================] - 41s 201us/step - loss: 3.8198
```

```
Epoch 00001: loss improved from inf to 3.81983, saving model to /content/drive/My Drive/c
ase study/upload 15mis/weight/weights-01-3.82.hdf5
Epoch 2/50
201917/201917 [==============================] - 33s 161us/step - loss: 3.7844

Epoch 00002: loss improved from 3.81983 to 3.78442, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-02-3.78.hdf5
Epoch 3/50
201917/201917 [==============================] - 34s 166us/step - loss: 3.7653

Epoch 00003: loss improved from 3.78442 to 3.76533, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-03-3.77.hdf5
Epoch 4/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.7521

Epoch 00004: loss improved from 3.76533 to 3.75206, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-04-3.75.hdf5
Epoch 5/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.7451

Epoch 00005: loss improved from 3.75206 to 3.74510, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-05-3.75.hdf5
Epoch 6/50
201917/201917 [==============================] - 33s 162us/step - loss: 3.7368

Epoch 00006: loss improved from 3.74510 to 3.73680, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-06-3.74.hdf5
Epoch 7/50
201917/201917 [==============================] - 34s 166us/step - loss: 3.7319

Epoch 00007: loss improved from 3.73680 to 3.73186, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-07-3.73.hdf5
Epoch 8/50
201917/201917 [==============================] - 33s 165us/step - loss: 3.7238

Epoch 00008: loss improved from 3.73186 to 3.72382, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-08-3.72.hdf5
Epoch 9/50
201917/201917 [==============================] - 34s 171us/step - loss: 3.7141

Epoch 00009: loss improved from 3.72382 to 3.71405, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-09-3.71.hdf5
Epoch 10/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.7034

Epoch 00010: loss improved from 3.71405 to 3.70340, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-10-3.70.hdf5
Epoch 11/50
201917/201917 [==============================] - 33s 165us/step - loss: 3.6891

Epoch 00011: loss improved from 3.70340 to 3.68909, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-11-3.69.hdf5
Epoch 12/50
201917/201917 [==============================] - 34s 170us/step - loss: 3.6903

Epoch 00012: loss did not improve from 3.68909
Epoch 13/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.6678

Epoch 00013: loss improved from 3.68909 to 3.66782, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-13-3.67.hdf5
Epoch 14/50
201917/201917 [==============================] - 33s 166us/step - loss: 3.6782

Epoch 00014: loss did not improve from 3.66782
Epoch 15/50
201917/201917 [==============================] - 33s 165us/step - loss: 3.6432

Epoch 00015: loss improved from 3.66782 to 3.64320, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-15-3.64.hdf5
Epoch 16/50
```

```
201917/201917 [==============================] - 34s 168us/step - loss: 3.6346

Epoch 00016: loss improved from 3.64320 to 3.63458, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-16-3.63.hdf5
Epoch 17/50
201917/201917 [==============================] - 34s 169us/step - loss: 3.6156

Epoch 00017: loss improved from 3.63458 to 3.61559, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-17-3.62.hdf5
Epoch 18/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.6032

Epoch 00018: loss improved from 3.61559 to 3.60324, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-18-3.60.hdf5
Epoch 19/50
201917/201917 [==============================] - 34s 169us/step - loss: 3.5979

Epoch 00019: loss improved from 3.60324 to 3.59792, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-19-3.60.hdf5
Epoch 20/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.5631

Epoch 00020: loss improved from 3.59792 to 3.56309, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-20-3.56.hdf5
Epoch 21/50
201917/201917 [==============================] - 34s 169us/step - loss: 3.5460

Epoch 00021: loss improved from 3.56309 to 3.54604, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-21-3.55.hdf5
Epoch 22/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.5160

Epoch 00022: loss improved from 3.54604 to 3.51604, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-22-3.52.hdf5
Epoch 23/50
201917/201917 [==============================] - 33s 162us/step - loss: 3.4968

Epoch 00023: loss improved from 3.51604 to 3.49681, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-23-3.50.hdf5
Epoch 24/50
201917/201917 [==============================] - 33s 164us/step - loss: 3.5214

Epoch 00024: loss did not improve from 3.49681
Epoch 25/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.5069

Epoch 00025: loss did not improve from 3.49681
Epoch 26/50
201917/201917 [==============================] - 34s 166us/step - loss: 3.4552

Epoch 00026: loss improved from 3.49681 to 3.45522, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-26-3.46.hdf5
Epoch 27/50
201917/201917 [==============================] - 34s 169us/step - loss: 3.4542

Epoch 00027: loss improved from 3.45522 to 3.45423, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-27-3.45.hdf5
Epoch 28/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.4127

Epoch 00028: loss improved from 3.45423 to 3.41270, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-28-3.41.hdf5
Epoch 29/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.3868

Epoch 00029: loss improved from 3.41270 to 3.38681, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-29-3.39.hdf5
Epoch 30/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.3827

Epoch 00030: loss improved from 3.38681 to 3.38272, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-30-3.38.hdf5
```

```
Epoch 31/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.3564

Epoch 00031: loss improved from 3.38272 to 3.35639, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-31-3.36.hdf5
Epoch 32/50
201917/201917 [==============================] - 34s 169us/step - loss: 3.3473

Epoch 00032: loss improved from 3.35639 to 3.34732, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-32-3.35.hdf5
Epoch 33/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.3033

Epoch 00033: loss improved from 3.34732 to 3.30326, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-33-3.30.hdf5
Epoch 34/50
201917/201917 [==============================] - 34s 170us/step - loss: 3.3119

Epoch 00034: loss did not improve from 3.30326
Epoch 35/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.3224

Epoch 00035: loss did not improve from 3.30326
Epoch 36/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.2929

Epoch 00036: loss improved from 3.30326 to 3.29290, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-36-3.29.hdf5
Epoch 37/50
201917/201917 [==============================] - 35s 175us/step - loss: 3.3022

Epoch 00037: loss did not improve from 3.29290
Epoch 38/50
201917/201917 [==============================] - 35s 174us/step - loss: 3.2817

Epoch 00038: loss improved from 3.29290 to 3.28170, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-38-3.28.hdf5
Epoch 39/50
201917/201917 [==============================] - 35s 171us/step - loss: 3.2628

Epoch 00039: loss improved from 3.28170 to 3.26284, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-39-3.26.hdf5
Epoch 40/50
201917/201917 [==============================] - 34s 169us/step - loss: 3.2841

Epoch 00040: loss did not improve from 3.26284
Epoch 41/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.2276

Epoch 00041: loss improved from 3.26284 to 3.22755, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-41-3.23.hdf5
Epoch 42/50
201917/201917 [==============================] - 34s 168us/step - loss: 3.2283

Epoch 00042: loss did not improve from 3.22755
Epoch 43/50
201917/201917 [==============================] - 34s 169us/step - loss: 3.2344

Epoch 00043: loss did not improve from 3.22755
Epoch 44/50
201917/201917 [==============================] - 35s 172us/step - loss: 3.2263

Epoch 00044: loss improved from 3.22755 to 3.22630, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-44-3.23.hdf5
Epoch 45/50
201917/201917 [==============================] - 35s 172us/step - loss: 3.2161

Epoch 00045: loss improved from 3.22630 to 3.21606, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-45-3.22.hdf5
Epoch 46/50
201917/201917 [==============================] - 36s 176us/step - loss: 3.2198
```

```
Epoch 00046: loss did not improve from 3.21606
Epoch 47/50
201917/201917 [==============================] - 34s 167us/step - loss: 3.1989

Epoch 00047: loss improved from 3.21606 to 3.19891, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-47-3.20.hdf5
Epoch 48/50
201917/201917 [==============================] - 34s 166us/step - loss: 3.2204

Epoch 00048: loss did not improve from 3.19891
Epoch 49/50
201917/201917 [==============================] - 35s 174us/step - loss: 3.2304

Epoch 00049: loss did not improve from 3.19891
Epoch 50/50
201917/201917 [==============================] - 33s 166us/step - loss: 3.2049

Epoch 00050: loss did not improve from 3.19891
```

Out[ ]:

```
<keras.callbacks.callbacks.History at 0x7f7815f3e908>
```

In [ ]:

```
model = Sequential()
model.add(Embedding(324, 128, input_length=324))
model.add(Dropout(0.25))
model.add(Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(MaxPooling1D(2))

model.add(LSTM(50))
model.add(Dense(1))
model.compile(optimizer='adam', loss=root_mean_squared_error)
#loading best weights
model.load_weights('/content/drive/My Drive/case study/upload 15mis/weight/weights-47-3.2
0.hdf5')
```

In [ ]:

```
ypred = model.predict(test_df[train_cols])
```

In [ ]:

```
sub_df = pd.DataFrame({"card_id":test_df['card_id'].values})
sub_df["target"] = ypred
sub_df.to_csv("cnn_lstm_F.csv", index=False)
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/case study/upload 15mis/cnn_f.PNG")
```

```
Score in Kaggle
```

Out[ ]:

cnn_lstm_F.csv 36 minutes ago by Niranjan B Subramanian add submission details	3.68342	3.77028

without maxpooling

In [ ]:

```
from keras.layers import Embedding
from keras.models import Sequential
from keras.layers import Dense, Conv1D, Flatten, Dropout, MaxPooling1D
from keras.layers import BatchNormalization, LSTM
```

```
from sklearn.metrics import mean_squared_error
from keras import backend as K
from keras.callbacks import EarlyStopping, ModelCheckpoint

def root_mean_squared_error(y_true, y_pred):
        return K.sqrt(K.mean(K.square(y_pred - y_true)))

filepath = '/content/drive/My Drive/case study/upload 15mis/weight/weights-{epoch:02d}-{l
oss:.2f}.hdf5'
early_stop = EarlyStopping(monitor='loss',patience=6, verbose=1, mode='auto')
model_ckpt = ModelCheckpoint(monitor='loss', save_best_only=True, verbose=1, mode='auto'
,filepath=filepath)

model = Sequential()
model.add(Embedding(324, 128, input_length=324))
model.add(Dropout(0.25))
model.add(Conv1D(64,3,padding='valid',activation='relu',strides=1))

model.add(LSTM(50))
model.add(Dense(1))
model.compile(optimizer='adam', loss=root_mean_squared_error)
```

In [ ]:

```
model.fit(train_df[train_cols], target, batch_size=512, epochs=50, callbacks=[model_ckpt
, early_stop])
```

```
Epoch 1/50
201917/201917 [==============================] - 206s 1ms/step - loss: 3.7428

Epoch 00001: loss improved from 3.81497 to 3.74282, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-01-3.74.hdf5
Epoch 2/50
201917/201917 [==============================] - 203s 1ms/step - loss: 3.7292

Epoch 00002: loss improved from 3.74282 to 3.72921, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-02-3.73.hdf5
Epoch 3/50
201917/201917 [==============================] - 205s 1ms/step - loss: 3.7236

Epoch 00003: loss improved from 3.72921 to 3.72363, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-03-3.72.hdf5
Epoch 4/50
201917/201917 [==============================] - 207s 1ms/step - loss: 3.7250

Epoch 00004: loss did not improve from 3.72363
Epoch 5/50
201917/201917 [==============================] - 206s 1ms/step - loss: 3.7145

Epoch 00005: loss improved from 3.72363 to 3.71454, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-05-3.71.hdf5
Epoch 6/50
201917/201917 [==============================] - 203s 1ms/step - loss: 3.7112

Epoch 00006: loss improved from 3.71454 to 3.71117, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-06-3.71.hdf5
Epoch 7/50
201917/201917 [==============================] - 203s 1ms/step - loss: 3.7110

Epoch 00007: loss improved from 3.71117 to 3.71102, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-07-3.71.hdf5
Epoch 8/50
201917/201917 [==============================] - 202s 1ms/step - loss: 3.7033

Epoch 00008: loss improved from 3.71102 to 3.70332, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-08-3.70.hdf5
Epoch 9/50
201917/201917 [==============================] - 212s 1ms/step - loss: 3.7164

Epoch 00009: loss did not improve from 3.70332
Epoch 10/50
201917/201917 [==============================] - 212s 1ms/step - loss: 3.6988
```

```
Epoch 00010: loss improved from 3.70332 to 3.69884, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-10-3.70.hdf5
Epoch 11/50
201917/201917 [==============================] - 217s 1ms/step - loss: 3.6891

Epoch 00011: loss improved from 3.69884 to 3.68910, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-11-3.69.hdf5
Epoch 12/50
201917/201917 [==============================] - 224s 1ms/step - loss: 3.6802

Epoch 00012: loss improved from 3.68910 to 3.68021, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-12-3.68.hdf5
Epoch 13/50
201917/201917 [==============================] - 220s 1ms/step - loss: 3.6613

Epoch 00013: loss improved from 3.68021 to 3.66129, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-13-3.66.hdf5
Epoch 14/50
201917/201917 [==============================] - 221s 1ms/step - loss: 3.6498

Epoch 00014: loss improved from 3.66129 to 3.64975, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-14-3.65.hdf5
Epoch 15/50
201917/201917 [==============================] - 222s 1ms/step - loss: 3.6485

Epoch 00015: loss improved from 3.64975 to 3.64847, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-15-3.65.hdf5
Epoch 16/50
201917/201917 [==============================] - 219s 1ms/step - loss: 3.6363

Epoch 00016: loss improved from 3.64847 to 3.63626, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-16-3.64.hdf5
Epoch 17/50
201917/201917 [==============================] - 222s 1ms/step - loss: 3.6091

Epoch 00017: loss improved from 3.63626 to 3.60912, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-17-3.61.hdf5
Epoch 18/50
201917/201917 [==============================] - 222s 1ms/step - loss: 3.6013

Epoch 00018: loss improved from 3.60912 to 3.60128, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-18-3.60.hdf5
Epoch 19/50
201917/201917 [==============================] - 221s 1ms/step - loss: 3.5752

Epoch 00019: loss improved from 3.60128 to 3.57520, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-19-3.58.hdf5
Epoch 20/50
201917/201917 [==============================] - 223s 1ms/step - loss: 3.5817

Epoch 00020: loss did not improve from 3.57520
Epoch 21/50
201917/201917 [==============================] - 227s 1ms/step - loss: 3.5661

Epoch 00021: loss improved from 3.57520 to 3.56613, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-21-3.57.hdf5
Epoch 22/50
201917/201917 [==============================] - 219s 1ms/step - loss: 3.5482

Epoch 00022: loss improved from 3.56613 to 3.54818, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-22-3.55.hdf5
Epoch 23/50
201917/201917 [==============================] - 225s 1ms/step - loss: 3.5344

Epoch 00023: loss improved from 3.54818 to 3.53436, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-23-3.53.hdf5
Epoch 24/50
201917/201917 [==============================] - 223s 1ms/step - loss: 3.5280

Epoch 00024: loss improved from 3.53436 to 3.52797, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-24-3.53.hdf5
```

```
Epoch 25/50
201917/201917 [==============================] - 221s 1ms/step - loss: 3.5130

Epoch 00025: loss improved from 3.52797 to 3.51302, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-25-3.51.hdf5
Epoch 26/50
201917/201917 [==============================] - 224s 1ms/step - loss: 3.5092

Epoch 00026: loss improved from 3.51302 to 3.50918, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-26-3.51.hdf5
Epoch 27/50
201917/201917 [==============================] - 225s 1ms/step - loss: 3.5017

Epoch 00027: loss improved from 3.50918 to 3.50166, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-27-3.50.hdf5
Epoch 28/50
201917/201917 [==============================] - 225s 1ms/step - loss: 3.4921

Epoch 00028: loss improved from 3.50166 to 3.49211, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-28-3.49.hdf5
Epoch 29/50
201917/201917 [==============================] - 217s 1ms/step - loss: 3.4817

Epoch 00029: loss improved from 3.49211 to 3.48167, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-29-3.48.hdf5
Epoch 30/50
201917/201917 [==============================] - 220s 1ms/step - loss: 3.4997

Epoch 00030: loss did not improve from 3.48167
Epoch 31/50
201917/201917 [==============================] - 223s 1ms/step - loss: 3.5017

Epoch 00031: loss did not improve from 3.48167
Epoch 32/50
201917/201917 [==============================] - 218s 1ms/step - loss: 3.4867

Epoch 00032: loss did not improve from 3.48167
Epoch 33/50
201917/201917 [==============================] - 214s 1ms/step - loss: 3.4606

Epoch 00033: loss improved from 3.48167 to 3.46061, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-33-3.46.hdf5
Epoch 34/50
201917/201917 [==============================] - 212s 1ms/step - loss: 3.4631

Epoch 00034: loss did not improve from 3.46061
Epoch 35/50
201917/201917 [==============================] - 213s 1ms/step - loss: 3.4573

Epoch 00035: loss improved from 3.46061 to 3.45727, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-35-3.46.hdf5
Epoch 36/50
201917/201917 [==============================] - 216s 1ms/step - loss: 3.4515

Epoch 00036: loss improved from 3.45727 to 3.45150, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-36-3.45.hdf5
Epoch 37/50
201917/201917 [==============================] - 213s 1ms/step - loss: 3.4592

Epoch 00037: loss did not improve from 3.45150
Epoch 38/50
201917/201917 [==============================] - 213s 1ms/step - loss: 3.4400

Epoch 00038: loss improved from 3.45150 to 3.43999, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-38-3.44.hdf5
Epoch 39/50
201917/201917 [==============================] - 208s 1ms/step - loss: 3.4478

Epoch 00039: loss did not improve from 3.43999
Epoch 40/50
201917/201917 [==============================] - 208s 1ms/step - loss: 3.4621
```

```
Epoch 00040: loss did not improve from 3.43999
Epoch 41/50
201917/201917 [==============================] - 216s 1ms/step - loss: 3.4812

Epoch 00041: loss did not improve from 3.43999
Epoch 42/50
201917/201917 [==============================] - 210s 1ms/step - loss: 3.4315

Epoch 00042: loss improved from 3.43999 to 3.43147, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-42-3.43.hdf5
Epoch 43/50
201917/201917 [==============================] - 213s 1ms/step - loss: 3.4312

Epoch 00043: loss improved from 3.43147 to 3.43124, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-43-3.43.hdf5
Epoch 44/50
201917/201917 [==============================] - 213s 1ms/step - loss: 3.4309

Epoch 00044: loss improved from 3.43124 to 3.43092, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-44-3.43.hdf5
Epoch 45/50
201917/201917 [==============================] - 213s 1ms/step - loss: 3.4173

Epoch 00045: loss improved from 3.43092 to 3.41725, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-45-3.42.hdf5
Epoch 46/50
201917/201917 [==============================] - 217s 1ms/step - loss: 3.4300

Epoch 00046: loss did not improve from 3.41725
Epoch 47/50
201917/201917 [==============================] - 214s 1ms/step - loss: 3.4163

Epoch 00047: loss improved from 3.41725 to 3.41628, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-47-3.42.hdf5
Epoch 48/50
201917/201917 [==============================] - 212s 1ms/step - loss: 3.4039

Epoch 00048: loss improved from 3.41628 to 3.40387, saving model to /content/drive/My Dri
ve/case study/upload 15mis/weight/weights-48-3.40.hdf5
Epoch 49/50
201917/201917 [==============================] - 211s 1ms/step - loss: 3.4220

Epoch 00049: loss did not improve from 3.40387
Epoch 50/50
201917/201917 [==============================] - 212s 1ms/step - loss: 3.4198

Epoch 00050: loss did not improve from 3.40387
```

Out[ ]:

```
<keras.callbacks.callbacks.History at 0x7f30075be390>
```

In [ ]:

```
model1 = Sequential()
model1.add(Embedding(324, 128, input_length=324))
model1.add(Dropout(0.25))
model1.add(Conv1D(64,3,padding='valid',activation='relu',strides=1))

model1.add(LSTM(50))
model1.add(Dense(1))
model1.compile(optimizer='adam', loss=root_mean_squared_error)

model1.load_weights('/content/drive/My Drive/case study/upload 15mis/weight/weights-48-3.
40.hdf5')
```

In [ ]:

```
ypred = model1.predict(test_df[train_cols])
```

In [ ]:

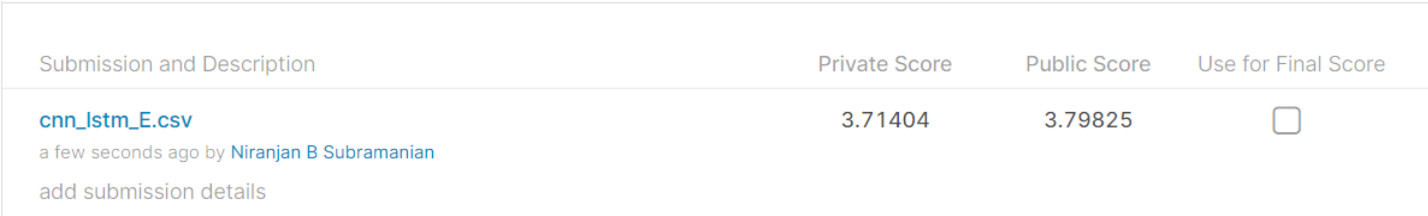
```
sub_df = pd.DataFrame({"card_id":test_df['card_id'].values})
sub_df["target"] = ypred
sub_df.to_csv("cnn_lstm_E.csv", index=False)
```

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/case study/upload 15mis/cnn_e.PNG")
```

Score in Kaggle

Out[ ]:



Submission and Description	Private Score	Public Score	Use for Final Score
cnn_lstm_E.csv a few seconds ago by Niranjan B Subramanian add submission details	3.71404	3.79825	

The performance of all the deep learning based models that we have tried are worse compared to the other models. However adding the cnn + lstm slightly improves the performance when compared to simple conv1d model.

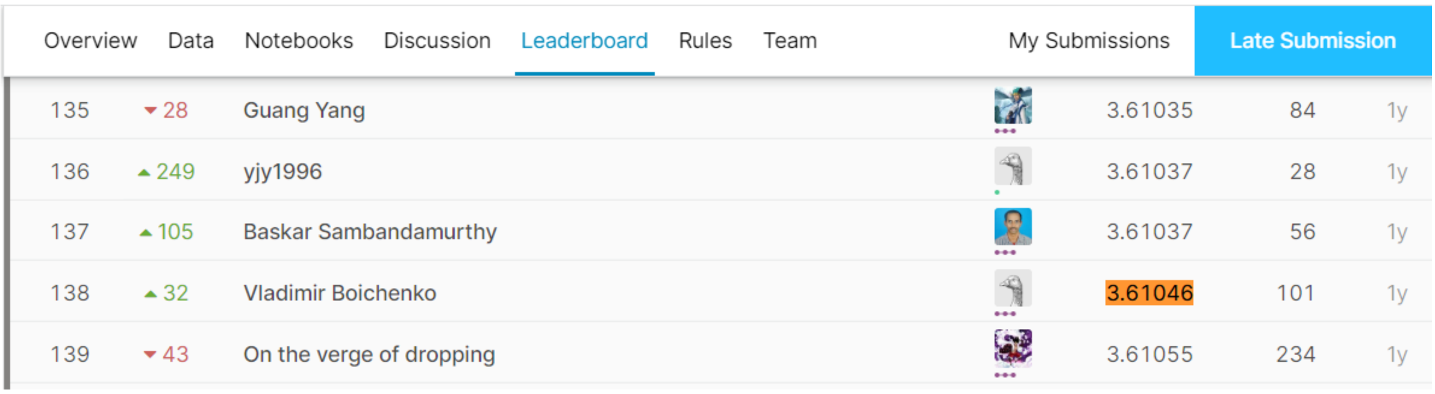
The best score I get is by using the stacked model by using all the features. It gives me a score of 3.61046, this puts me in top 4% in the leaderboard.

In [ ]:

```
from IPython.display import Image
print("Score in Kaggle")
Image("/content/drive/My Drive/Colab Notebooks/ELO/MODEL/leader.PNG")
```

Score in Kaggle

Out[ ]:



Overview Data Notebooks Discussion Leaderboard Rules Team My Submissions Late Submission

135	▼ 28	Guang Yang	3.61035	84	1y
136	▲ 249	yjy1996	3.61037	28	1y
137	▲ 105	Baskar Sambandamurthy	3.61037	56	1y
138	▲ 32	Vladimir Boichenko	3.61046	101	1y
139	▼ 43	On the verge of dropping	3.61055	234	1y

SUMMARY OF THE MODELS

In [1]:

```
from prettytable import PrettyTable

x = PrettyTable()

x.field_names = ["Models", "Private Score", "Public Score"]

x.add_row(["LGBM with RFE features\n", 3.61287, 3.69493])
x.add_row(["LGBM with all the features\n", 3.61084, 3.69359])
x.add_row(["Simple weight Blending of predictions\n0.8*LGB prediction + 0.2*XGB predicti
ons\n", 3.61184, 3.69340])
x.add_row(["XGBOOST with all the features\n", 3.62927, 3.70532])
x.add_row(["Simple Bagging model with\nDT base learner and stacked XGB as metalearner\n",
3.68270, 3.76989])
```

```
x.add_row(["Stacked LGB and XGB predictions\nwith Ridge as metalearner\n", 3.61046, 3.69
093])
x.add_row(["ANN with 5 layers(reduced dropout)\n", 3.81882, 3.93563])
x.add_row(["Conv1D Model\n", 3.77332, 3.89143])
x.add_row(["Conv1D + LSTM Model with maxpool\n", 3.75455, 3.86007])
x.add_row(["Conv1D + LSTM Model without maxpool\n", 3.71404, 3.79825])

print(x)
```

Models	Private Score	Public Score
LGBM with RFE features	3.61287	3.69493
LGBM with all the features	3.61084	3.69359
Simple weight Blending of predictions 0.8*LGB prediction + 0.2*XGB predictions	3.61184	3.6934
XGBOOST with all the features	3.62927	3.70532
Simple Bagging model with DT base learner and stacked XGB as metalearner	3.6827	3.76989
Stacked LGB and XGB predictions with Ridge as metalearner	3.61046	3.69093
ANN with 5 layers(reduced dropout)	3.81882	3.93563
Conv1D Model	3.77332	3.89143
Conv1D + LSTM Model with maxpool	3.75455	3.86007
Conv1D + LSTM Model without maxpool	3.75455	3.86007

1) At first I tried LGBM with the features which is selected using the Recursive Feature Elimination. There are a total of 254 features. The hyperparameters are tuned using Optuna. We got a score of 3.61284.

2) Then I tried LGBM with all the features which is around 330. Using all the features improves the performance of the model dramatically. We got a score of around 3.61084 Since this is a huge improvement from the last model we'll use all the data to train the XGB.

3) For XGBOOST the hyperparameters are tuned using the RandomSearchCV. Using the XGB model we attain a score of 3.62927 which is not that much impressive as we have already attained better score by using LGBM.

4) Next I tried simple weight based blending of the predictions of both LGBM and XGB. We got a score of 3.61184 which is better than the XGB model alone.

5) The custom bagging model with DT as base learner and their predictions stacked with XGBOOST as a metalearner is the worst performant of all. It gives a score of 3.6827

6) Also tried neural network based architectures like 3, 5 layer MLP, a Conv1D and CNN + LSTM models but their performance are not comparable to others.

7) Finally a stacked model with Ridge as the metamodel gives the best score. It gives a score of 3.61046

In [ ]: