Fraud credit card transaction detection

Candidate **ID : C1195415**

Question 1:

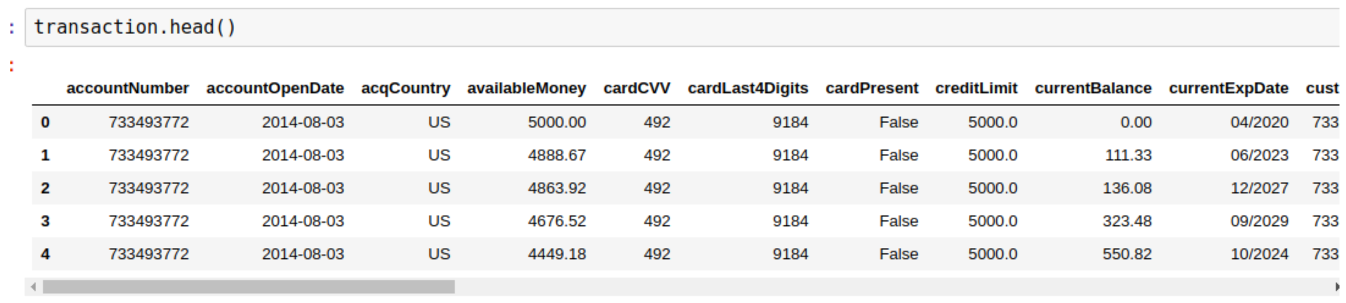
The data in the text file is in the line delimited json format with key value pair. I read the text file and stored every line delimited transaction in the python list. So that every element of the list denoted a transaction.

This is one of the transaction:

```
{'accountNumber': '733493772',
 'accountOpenDate': '2014-08-03',
 'acqCountry': 'US',
 'availableMoney': 5000.0,
 'cardCVV': '492',
 'cardLast4Digits': '9184',
 'cardPresent': False,
 'creditLimit': 5000.0,
 'currentBalance': 0.0,
 'currentExpDate': '04/2020',
 'customerId': '733493772',
 'dateOfLastAddressChange': '2014-08-03',
 'echoBuffer': '',
 'enteredCVV': '492',
 'expirationDateKeyInMatch': False,
 'isFraud': True,
 'merchantCategoryCode': 'rideshare',
 'merchantCity': '',
 'merchantCountryCode': 'US',
 'merchantName': 'Lyft',
 'merchantState': '',
 'merchantZip': '',
 'posConditionCode': '01',
 'posEntryMode': '05',
 'posOnPremises': '',
 'recurringAuthInd': '',
 'transactionAmount': 111.33,
 'transactionDateTime': '2016-01-08T19:04:50',
 'transactionType': 'PURCHASE'}
```

So it is in the form of dictionary where all the key denotes the feature of the transaction and value corresponding to its value for the key. So, the list which contains all the data is list of multiple of dictionaries.

We can use pandas to read list of dictionaries and store it in the dataframe. Dataframe is similar to a table which has indices as the row name and features as the column name.



```
: transaction.head()
:
```

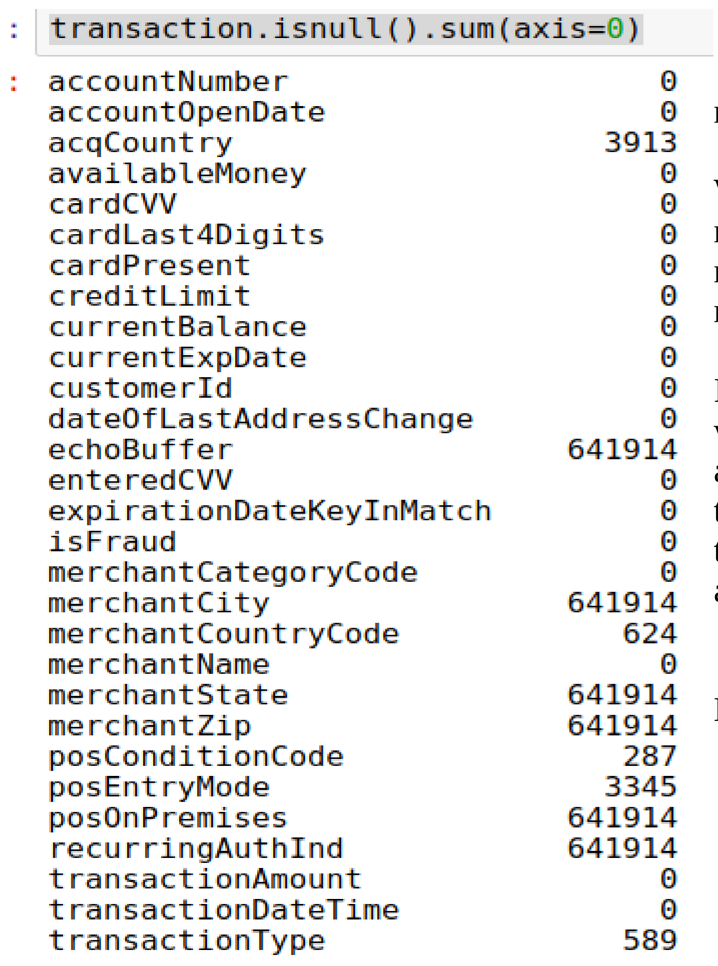
	accountNumber	accountOpenDate	acqCountry	availableMoney	cardCVV	cardLast4Digits	cardPresent	creditLimit	currentBalance	currentExpDate	cust
0	733493772	2014-08-03	US	5000.00	492	9184	False	5000.0	0.00	04/2020	733
1	733493772	2014-08-03	US	4888.67	492	9184	False	5000.0	111.33	06/2023	733
2	733493772	2014-08-03	US	4863.92	492	9184	False	5000.0	136.08	12/2027	733
3	733493772	2014-08-03	US	4676.52	492	9184	False	5000.0	323.48	09/2029	733
4	733493772	2014-08-03	US	4449.18	492	9184	False	5000.0	550.82	10/2024	733

These are all the columns (features) in the dataframe

```
Index(['accountNumber', 'accountOpenDate', 'acqCountry', 'availableMoney',
       'cardCVV', 'cardLast4Digits', 'cardPresent', 'creditLimit',
       'currentBalance', 'currentExpDate', 'customerId',
       'dateOfLastAddressChange', 'echoBuffer', 'enteredCVV',
       'expirationDateKeyInMatch', 'isFraud', 'merchantCategoryCode',
       'merchantCity', 'merchantCountryCode', 'merchantName', 'merchantState',
       'merchantZip', 'posConditionCode', 'posEntryMode', 'posOnPremises',
       'recurringAuthInd', 'transactionAmount', 'transactionDateTime',
       'transactionType'],
      dtype='object')
```

There are 641914 transactions and 29 features in the dataframe.

After having look at first 5 rows in the transaction dataframe, I found that some of the columns like merchantZip, dont have any information. We need to impute the missing values.



```
: transaction.isnull().sum(axis=0)

: accountNumber                     0
  accountOpenDate                   0
  acqCountry                     3913
  availableMoney                    0
  cardCVV                           0
  cardLast4Digits                   0
  cardPresent                       0
  creditLimit                       0
  currentBalance                    0
  currentExpDate                    0
  customerId                        0
  dateOfLastAddressChange           0
  echoBuffer                   641914
  enteredCVV                        0
  expirationDateKeyInMatch          0
  isFraud                           0
  merchantCategoryCode              0
  merchantCity                 641914
  merchantCountryCode             624
  merchantName                      0
  merchantState                641914
  merchantZip                  641914
  posConditionCode                287
  posEntryMode                   3345
  posOnPremises                641914
  recurringAuthInd             641914
  transactionAmount                 0
  transactionDateTime               0
  transactionType                 589
```

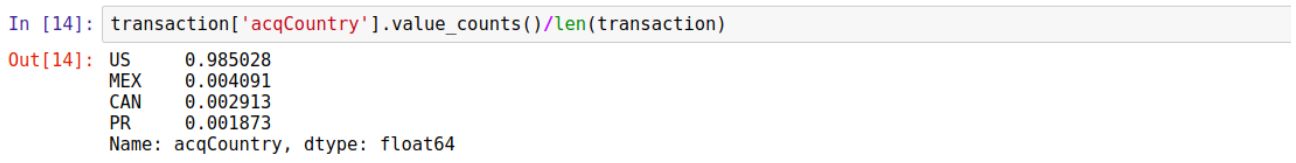
After some processing, we found out the number of missing values in the dataframe.

We can observe that for some of the columns number of missing values are same as the number of rows. For example, echobuffer, merchantCity, etc.

In such cases, imputing wont be perfect and we dont even have any base for the imputing as we dont know what data should be there in those columns. So we will directly remove these columns from the dataframe as these are insignificant with missing data.

Let deal with other columns one by one.

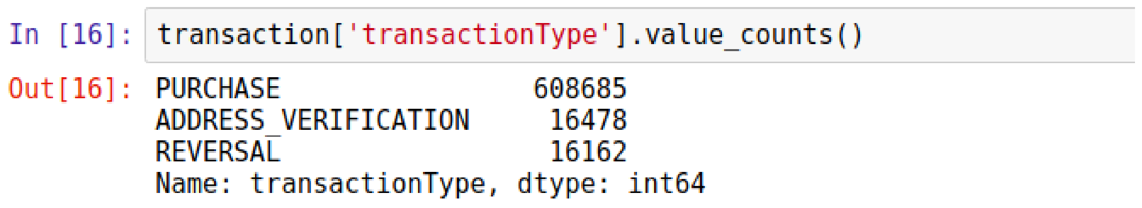
1) acqCountry:



```
In [14]: transaction['acqCountry'].value_counts()/len(transaction)

Out[14]: US     0.985028
         MEX    0.004091
         CAN    0.002913
         PR     0.001873
         Name: acqCountry, dtype: float64
```

Almost 99% of the values in the column has value 'US'. The number of missing values are 3193 which is greater than other values. In this case we cant even replace the values by mean as it is a categorical variable (the column which has non-numeric or text data is called as categorical variable).
Now, lets fill the missing values with the element having maximum frequency which is 'US'. We can also use some methods like using linear regression to predict the values in this case but the category having maximum frequency is dominating the column by large margin and so it would have become unbalanced dataset which would have resulted in 'US' only, in most of the cases. So we replaced it with the maximum frequency element i.e. ‘US’

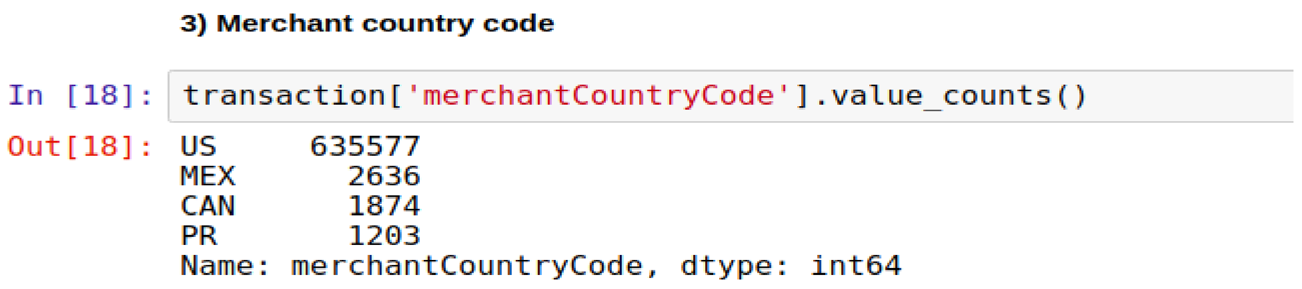
2) Transaction type:



```
In [16]: transaction['transactionType'].value_counts()

Out[16]: PURCHASE                608685
         ADDRESS_VERIFICATION     16478
         REVERSAL                 16162
         Name: transactionType, dtype: int64
```

Transaction type has 589 null values. It also has similar structure as that of acqCountry. Almost 95% of the rows has value 'PURCHASE' and less than 0.1% of the data is missing. So we will go with the same approach that we adopted for the acqCountry column. We will impute the missing values by 'PURCHASE'

3) Merchant Country code:

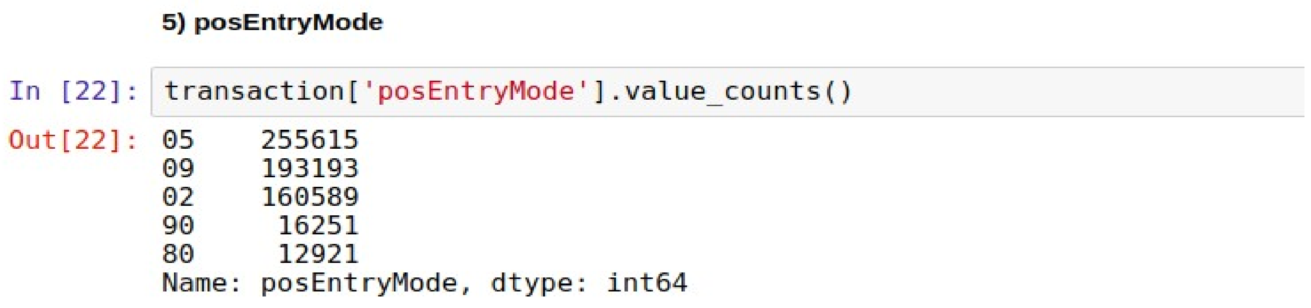


3) Merchant country code

```
In [18]: transaction['merchantCountryCode'].value_counts()

Out[18]: US     635577
         MEX      2636
         CAN      1874
         PR       1203
         Name: merchantCountryCode, dtype: int64
```

In this case, number of missing values are even more than the second most frequent element and we dont even have choice and need to fill the missing rows with the most frequent value which is 'US'

4) Pos entry mode

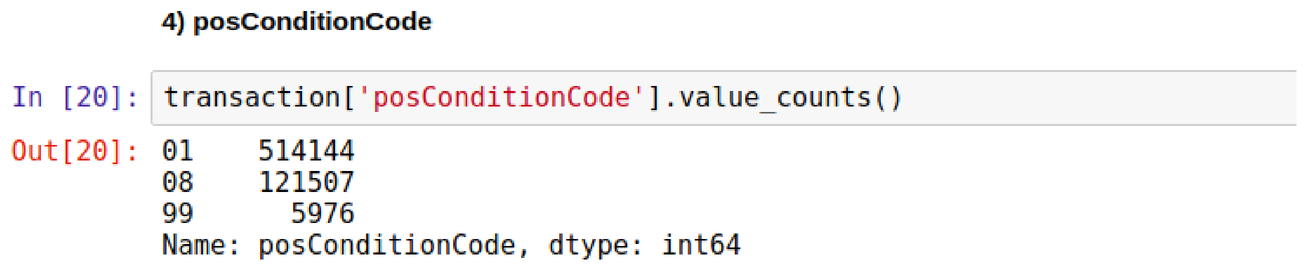


5) posEntryMode

```
In [22]: transaction['posEntryMode'].value_counts()

Out[22]: 05    255615
         09    193193
         02    160589
         90     16251
         80     12921
         Name: posEntryMode, dtype: int64
```

In this case, all the categories have almost equal or comparable distribution. So, we cant directly replace it with maximum element. However we can apply multi-class classification to predict the category which will be computationally heavy task for 0.5% of rows for a single column. Also this method doesnt guarantee the correct imputation. So, the simplest thing we can do is we can create another category for the missing values. ('00')

5) pos condition code:



4) posConditionCode

```
In [20]: transaction['posConditionCode'].value_counts()

Out[20]: 01    514144
         08    121507
         99      5976
         Name: posConditionCode, dtype: int64
```

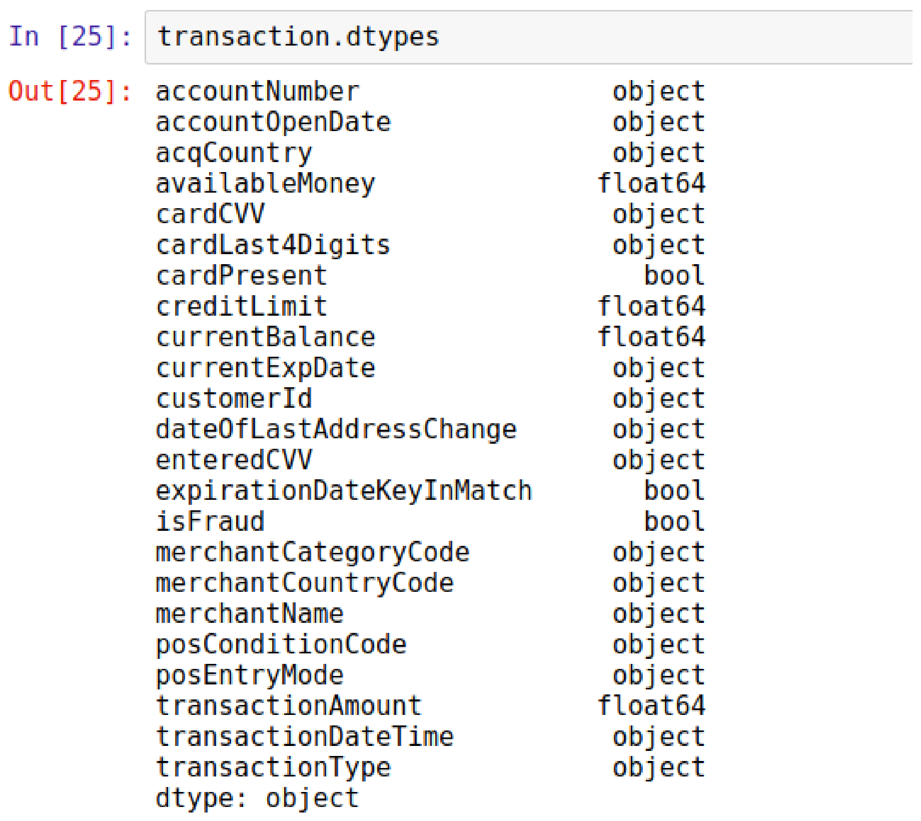
For this column, similar structure is observed as that of the other columns which we discussed before. Missing values are less than 0.1% and we will replace them with the most dominant category.

No method guarantees the perfect imputation as it is the complete unsupervised method. We are not aware of the results and can be completely random. Also its not even worth to invest large amount of time and resources to impute very small fraction of values. So, the approaches which we apply are completely based on the intuations or the previous patterns.

Now, we have removed all the missing values in our dataframe.

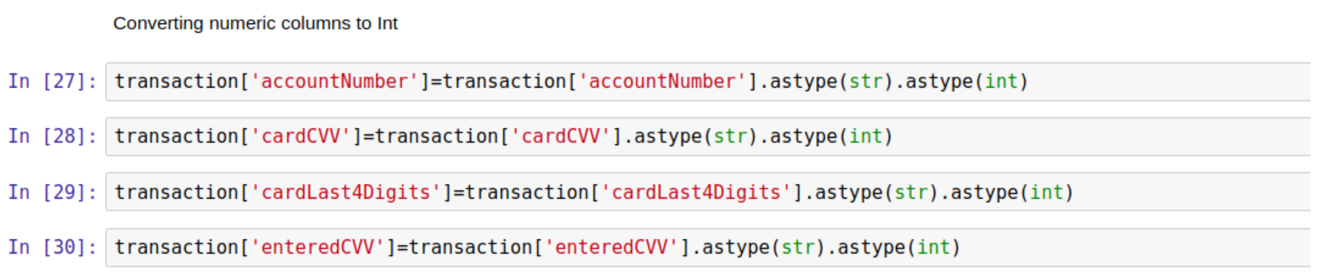
No method guarantees the perfect imputation as it is the complete unsupervised method. We are not aware of the results and can be completely random. Also its not even worth to invest large amount of time and resources to impute very small fraction of values.

If we have a look at the datatypes of all the columns, we will observe that some of the columns which contain numeric values have datatypes object like accountNumber and cardCVV.



```
In [25]: transaction.dtypes

Out[25]: accountNumber                object
         accountOpenDate              object
         acqCountry                   object
         availableMoney              float64
         cardCVV                      object
         cardLast4Digits              object
         cardPresent                    bool
         creditLimit                 float64
         currentBalance              float64
         currentExpDate               object
         customerId                   object
         dateOfLastAddressChange      object
         enteredCVV                   object
         expirationDateKeyInMatch       bool
         isFraud                        bool
         merchantCategoryCode         object
         merchantCountryCode          object
         merchantName                 object
         posConditionCode             object
         posEntryMode                 object
         transactionAmount           float64
         transactionDateTime          object
         transactionType              object
         dtype: object
```

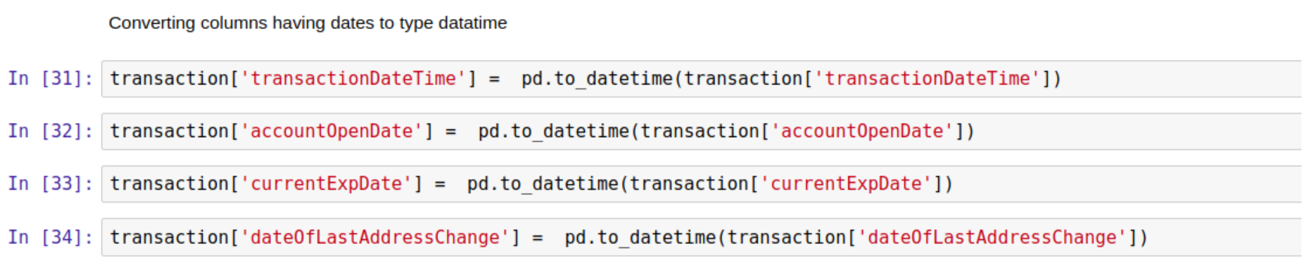
We will change the datatypes of all such columns.



Converting numeric columns to Int

```
In [27]: transaction['accountNumber']=transaction['accountNumber'].astype(str).astype(int)

In [28]: transaction['cardCVV']=transaction['cardCVV'].astype(str).astype(int)

In [29]: transaction['cardLast4Digits']=transaction['cardLast4Digits'].astype(str).astype(int)

In [30]: transaction['enteredCVV']=transaction['enteredCVV'].astype(str).astype(int)
```

Also, for some columns, data contains date and the datatype is object. We need to convert these columns to datetime as well.

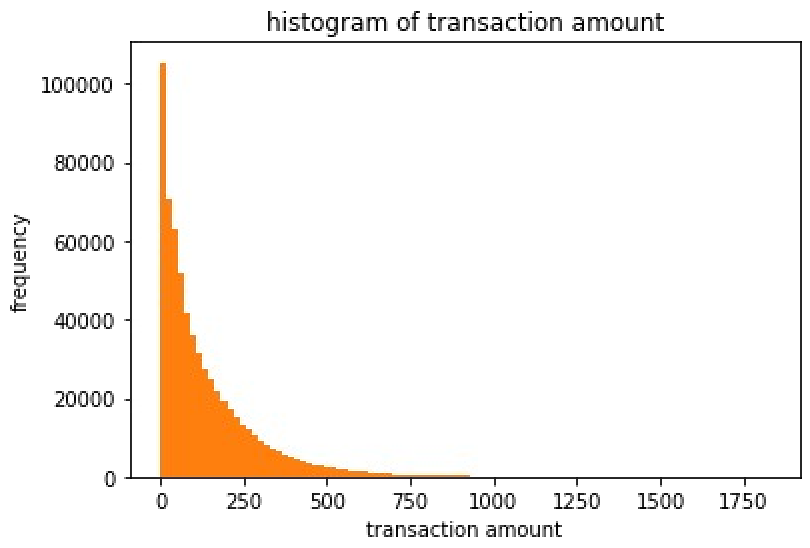


Converting columns having dates to type datatime

```
In [31]: transaction['transactionDateTime'] =  pd.to_datetime(transaction['transactionDateTime'])

In [32]: transaction['accountOpenDate'] =  pd.to_datetime(transaction['accountOpenDate'])

In [33]: transaction['currentExpDate'] =  pd.to_datetime(transaction['currentExpDate'])

In [34]: transaction['dateOfLastAddressChange'] =  pd.to_datetime(transaction['dateOfLastAddressChange'])
```

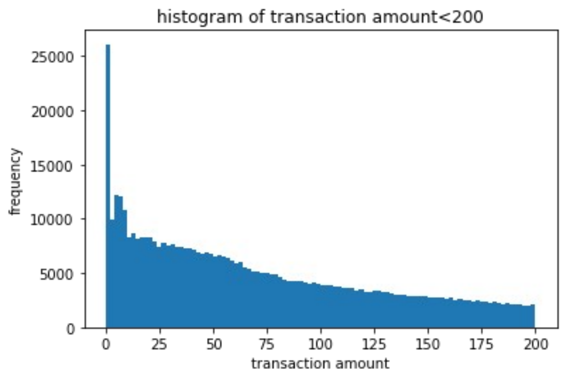
Now the data is properly structured having appropriate datatypes and no missing values. Also we are well familiar with the data. Now lets explore the data and observe relationships between the features.

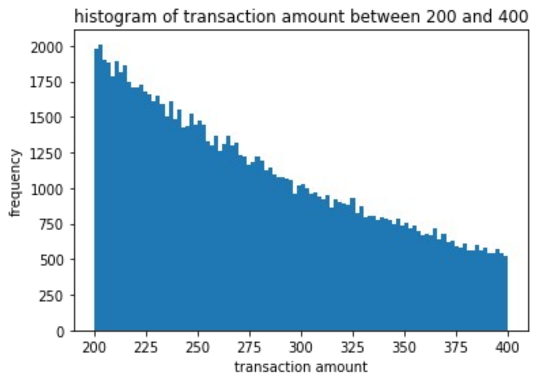
Question 2:

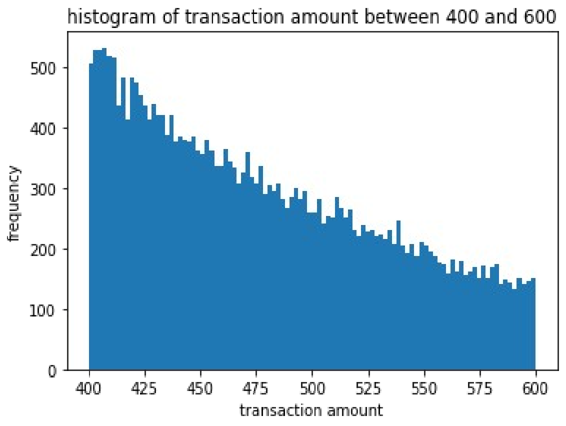


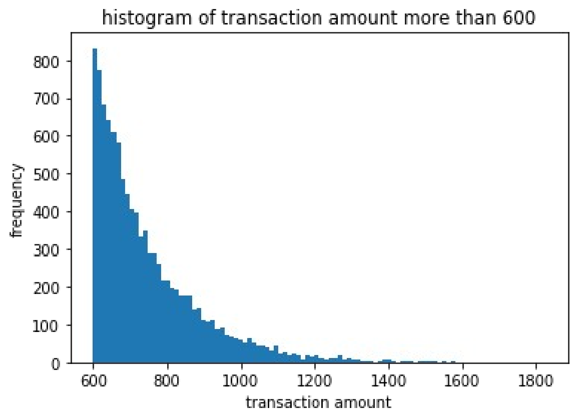
The above diagram shows histogram of all the transactions. Y-axis denotes the frequency of the transaction amount and transaction amount increases along the x-axis. We can observe the trend that as amount increases, the number of transactions are exponentially decreasing. So, users are using credit cards for lower amount of transactions more often than the larger amount. Probably the credit limit is putting restriction on the transactions.

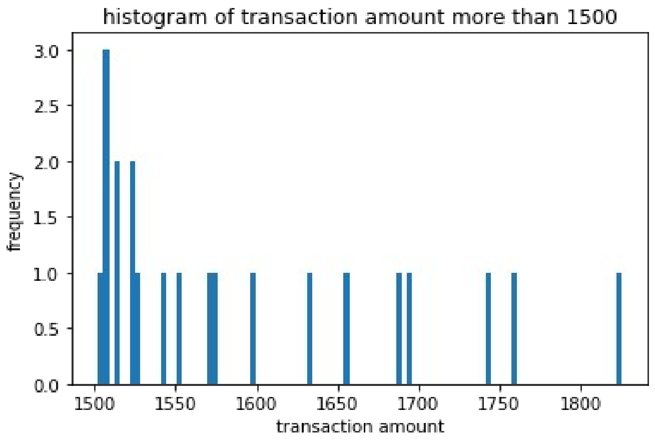
Lets have a look at transaction amount distribution more closely by dividing it in the different ranges.







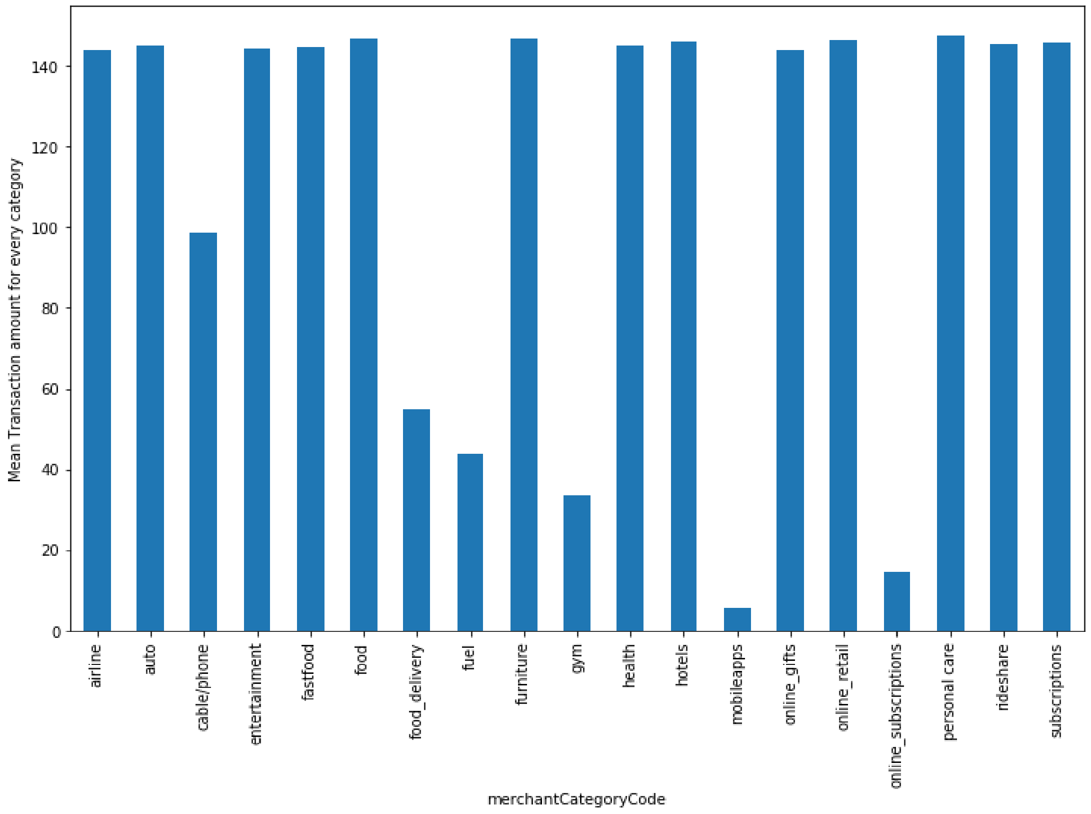




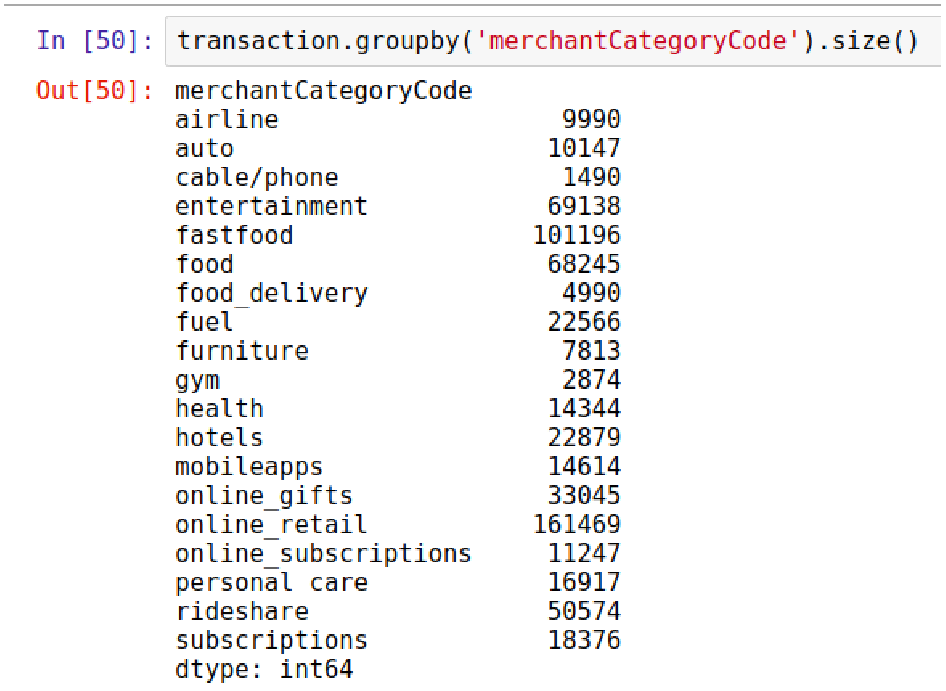
From the above diagrams we can conclude that almost similar trend is observed for all the ranges. As we increase the amount, the frequency also decreases which can be observed from the y-axis range. For the amount more than 1500, there are very rare transactions and clearly indicates that people dont use credit cards for amount more than 1000 very often and most of the transactions are there for amount less than 500.

Now lets segment it out for various categories of the merchants.

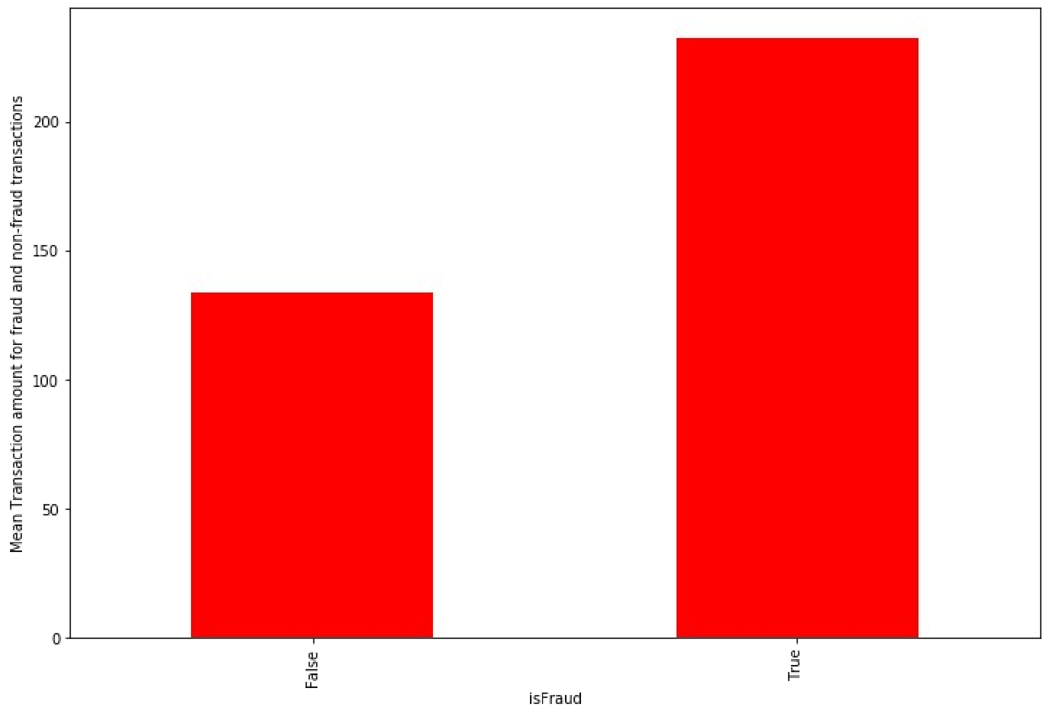
We can observe from the below graph that for most of the merchant categories, average amount is around 140. For fuels, gym, mobile apps and online subscriptions, the average amount lies around 20-40$. The reason is these are monthly one time expanses and generally arent much costlier.



These are number of transaction for every transactions. Maximum amount of transactions on credit cards are done for online retail, fast food and entertainment while least transactions are done for gym and furniture.



```
In [50]: transaction.groupby('merchantCategoryCode').size()

Out[50]: merchantCategoryCode
         airline                    9990
         auto                      10147
         cable/phone                1490
         entertainment             69138
         fastfood                 101196
         food                      68245
         food_delivery              4990
         fuel                      22566
         furniture                  7813
         gym                        2874
         health                    14344
         hotels                    22879
         mobileapps                14614
         online_gifts              33045
         online_retail            161469
         online_subscriptions      11247
         personal care             16917
         rideshare                 50574
         subscriptions             18376
         dtype: int64
```



Average amount for fraud transactions is around 230-240 and for not fraud transaction is much lesser which is 140. So, it is evident that fraud transactions are happening for higher amount while normal transactions have lesser average.

Also, average credit limit for every user is 10226. Here we are assuming that every user has only 1 account as the average credit limit for every account number is calculated.

Question 3:

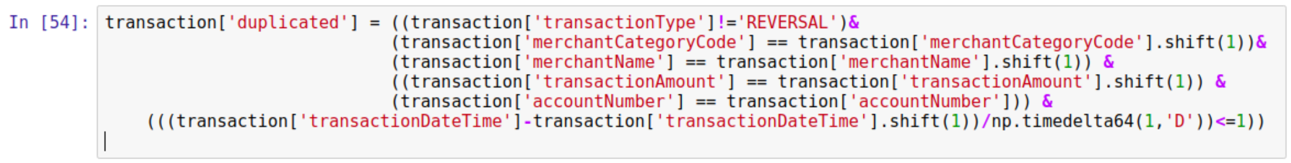
There two types of duplicated transactions:

1) Reversal:
Reversal are easier to detect as it is given by transaction type. In case of reversal the amount is credited back to the user account and so available amount increases. There are 16262 reversed transaction.

```
Here are some observations for reversal transactions:

Number of reversed transactions: 16162
Percentage of reversed transactions:  2.517782755945501 %
Total amount of reversed transactions: 2242915.099999993
Fraction of reversed transactions:  0.025851144829802112
```

2) Multi-swipe duplicated transaction:

These kind of transactions happen due to multiple swipe or multiple payment for the same transaction. In this case, the multiple transactions of same amount happen for the same merchant within short period of time. In the problem statement, the short period of time is not clearly defined and I will assume that multi-swipe transaction reflects on the account within 24 hours of the original amount. (By intuition we know that the transaction happens immediately in most of the cases. But, I am taking longer period as system might take more time to verify some of the transactions and I dont want to rule out such possibilities)



```
In [54]: transaction['duplicated'] = ((transaction['transactionType']!='REVERSAL')&
                                      (transaction['merchantCategoryCode'] == transaction['merchantCategoryCode'].shift(1))&
                                      (transaction['merchantName'] == transaction['merchantName'].shift(1)) &
                                      ((transaction['transactionAmount'] == transaction['transactionAmount'].shift(1)) &
                                      (transaction['accountNumber'] == transaction['accountNumber'])) &
             (((transaction['transactionDateTime']-transaction['transactionDateTime'].shift(1))/np.timedelta64(1,'D'))<=1))
```

This is how the multi-swipe duplicated transactions are detected. The duplicated transaction is the next to its original transaction which happen withing a day. Also, merchant name, amount, are same. Most of the cases are covered by these conditions. It is very well possible that the same item can be purchased from the same merchant on the same day by the same user. But for simplicity I am ignoring this case. If we have to consider this case, we need to reduce the time between these transactions to a minute or an hour. But it can also ignore some system changes.
In case of reversal transactions, same conditions can be satisfied so I am excluding reversal transactions by using first condition.

Also, I am assuming that there is only 1 user for the same account. There is another possibility that at the same time of multi-swipe transaction other user can use the same credit card/account for transaction of other merchant. In this case, there might be another transaction in between multiple swipes. I am completely ignoring this possibility by assuming that these transactions are consecutive.

There are 6175 multi-swipe repeated transactions which are based on the above assumptions in this data.

These are the observations for multi-swipe transactions:

```
Number of duplicate multi-swipe transactions: 6175
Percentage of duplicate multi-swipe transactions:  0.9619668678358784 %
Total amount of duplicate multi-swipe transactions: 884708.1200000027
Fraction of duplicate multi-swipe transactions:  0.010196871804118704
```

Question 4:

In this segment, we need to build a model which will successfully detect the fraud transactions. We cant afford to have false negatives in this case. So I will try to build model with high recall. High recall intuitively denotes that the classification model has ability to detect larger extent of positive examples.

Feature Engineering

The most important step before building the model is data cleaning and feature selection. We have already removed the missing values from the data. So we just need to select the most significant features and build new features which can be important.

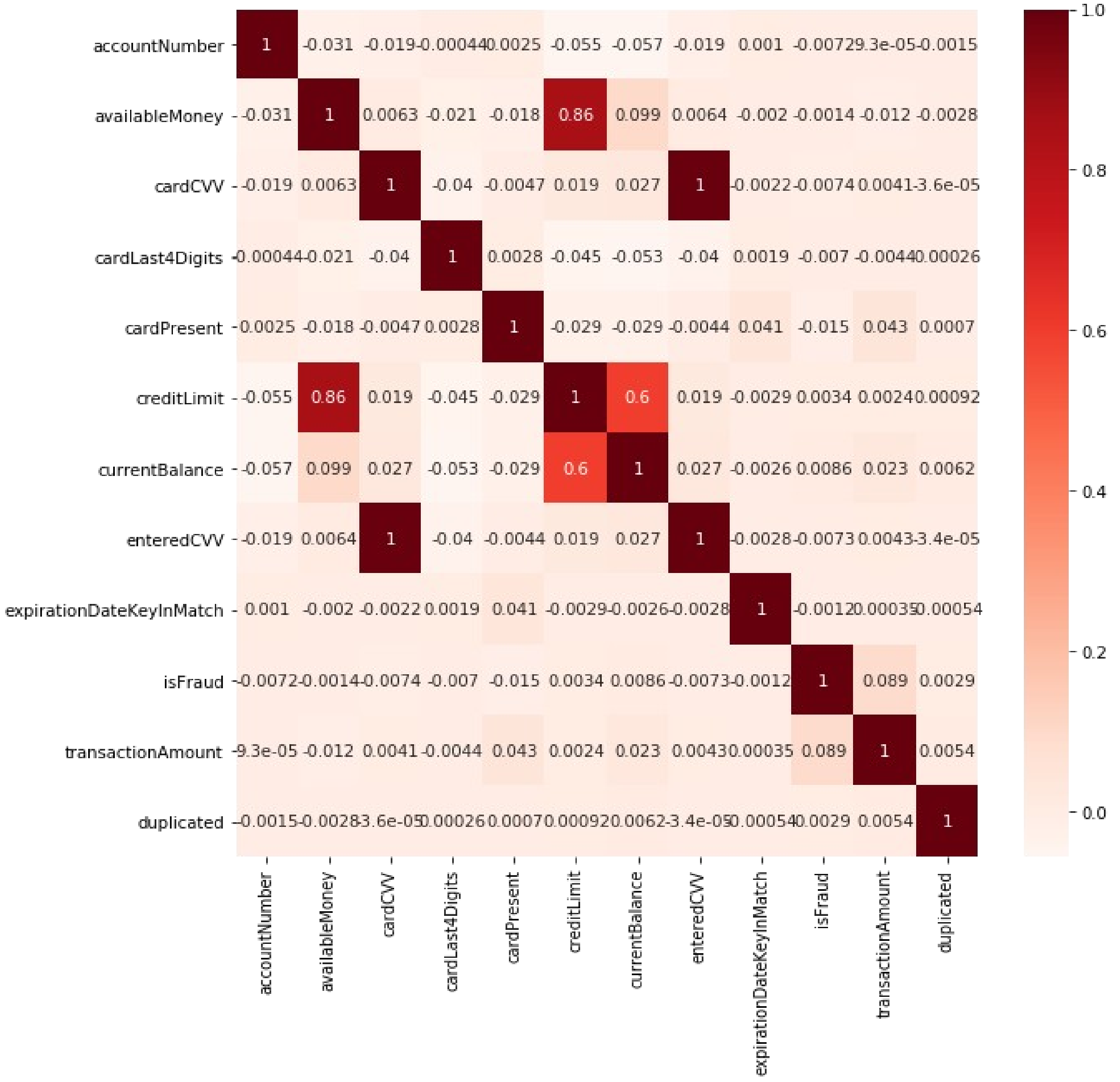
Note that classification model works the best with the integer data. They dont handle datetime and categorical variables very well and can be less accurate with such variables so our first task would be convert such significant features into numeric features.

```
accountNumber                        int64
accountOpenDate             datetime64[ns]
acqCountry                          object
availableMoney                     float64
cardCVV                              int64
cardLast4Digits                      int64
cardPresent                           bool
creditLimit                        float64
currentBalance                     float64
currentExpDate              datetime64[ns]
customerId                          object
dateOfLastAddressChange     datetime64[ns]
enteredCVV                           int64
expirationDateKeyInMatch              bool
isFraud                               bool
merchantCategoryCode                object
merchantCountryCode                 object
merchantName                        object
posConditionCode                    object
posEntryMode                        object
transactionAmount                  float64
transactionDateTime         datetime64[ns]
transactionType                     object
duplicated                            bool
```

1) Integer features:

Lets first filter out the insignificant numeric features (bool, int64 and float64).

I calculated pearson's correlation coefficient among every numeric variables. Its value ranges from -1 to +1 where, -1 denotes negative correlation between the features and +1 denotes very strong positive correlation. 0 denotes no correlation. We need to select the features which has very high positive and negative correlation coefficient with the ‘isfraud’ variable.



Every variable has coefficient +1 with itself.

We can see that apart from transaction amount all other variables are loosely correlated with ‘isFraud’. Intuitively we know that account number is a unique value and doesnt have any relation with the nature of transaction. Also, CVV, cardLast4Digits are insignificant factors. The same thing can be observed from the above heatmap as well.

Now lets decide a threshold and remove all the features which have absolute coefficient lesser than that value. We are considering the absolute values as the negative coefficient also denotes correlation.

```
cardPresent          0.014946
currentBalance       0.008611
isFraud              1.000000
transactionAmount    0.088708
```

These are the only features which we can select as only these features have coefficient larger than 0.008. I decided the threshold by observing the values for all the features. We wanted to ignore the insignificant features like accountNumber and CVV which are unique identifies and doesn’t affect transactions by any means.

This value also filtered some variables like creditlimit and availablebalance. However, pearson’s correlation coefficient denotes that ‘isFraud’ has very weak correlation with these features and in this case we will go by statistical significance rather than the intuition.

2) Categorical features:

Lets deal with categorical (object) variables now and convert them into most interpretable integer variables.

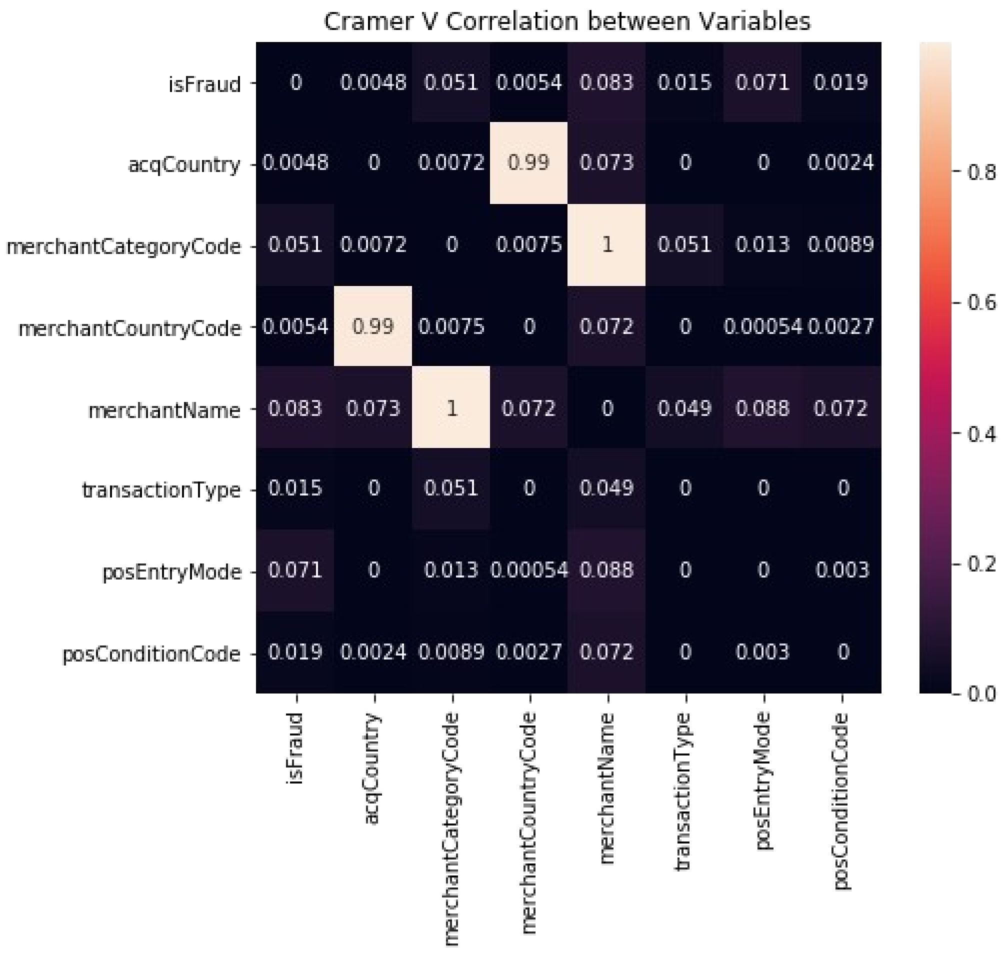
We can’t find the correlation between categorical and nominal variable by the same method. (Boolean variables are nominal as these variables are numeric but not continuous, as these variables have levels.)

Cramer’s V test is the best method to find correlation between categorical or nominal variables. It is based on pearson’s chi squared test. Its value is between 0 and 1 where 0 denotes no correlation and 1 denotes high correlation.

This article explains the concepts of statistical correlation for different types of variables.

The heatmap below displays the correlation between all the categorical variables with each other. As we are only interested in correlation with ‘isFraud’.
In this case, I haven’t calculated correlation of the variable with itself, thats why all the diagonal elements are 0 instead of 1.

We will apply the same threshold which we applied for integer correlations.



```
merchantCategoryCode    0.051460
merchantName            0.082775
transactionType         0.015113
posEntryMode            0.070797
posConditionCode        0.019179
```

These are some of the relevant features we get when I apply the threshold of 0.008.

Now lets deal with the selected categorical features one by one to apply vectorization on them.

1) merchantCategoryCode:

There are 19 total categories (unique values in this column). The number isnt too low, but isnt too high as well.

```
In [76]: print('Number of unique categories',len(transaction['merchantCategoryCode'].unique()))
         transaction['merchantCategoryCode'].value_counts()

         Number of unique categories 19

Out[76]: online_retail           161469
         fastfood                101196
         entertainment            69138
         food                     68245
         rideshare                50574
         online_gifts             33045
         hotels                   22879
         fuel                     22566
         subscriptions            18376
         personal care            16917
         mobileapps               14614
         health                   14344
         online_subscriptions     11247
         auto                     10147
         airline                   9990
         furniture                 7813
         food_delivery             4990
         gym                       2874
         cable/phone               1490
```

There are multiple methods which we can consider for vectorization. We can have a number for every category. (Ordinal encoding). However, the algorithm will assume that these are the levels. For example, if online_retail =1 and fastfood = 2, the algorithm will assume that fastfood has more weight than online_retails and so on. However, these are independent categories and there isnt significance leveling between these categories. So using ordinal encoding isnt a good idea.

We can use 1-hot encoding. In this case, 19 different columns will be created. Each will have value 0 or 1. where, 1 denotes value present.

2) Merchant name:

It is also of the most significant categorical feature. However, there are 2493 unique values in this column. Ordinal encoding wont work here as in this case also, there isnt any statistical significance leveling among the merchants. One-hot encoding wont be a good idea as it will introduce 2493 new features with a lot 2492 0’s for every column. We can use binary encoding or hashing encoding, but there is loss of information in these techniques.

Another approach is we can use k-means clustering to cluster similar merchant and use same category for similar types of merchant. However, merchant category denotes the class and from the heatmap we can observe that these two features are highly significant. So we can use merchant category only and dropping merchant name wont make significant difference as the same information is used in the other column.

3) Transaction type:

Transaction type column has 3 unique values. Each value has both the classes. So we need to consider all the categories. 1-hot encoding will work best in this case as number of categories are less with no statistical levels among the values.

With posEntryMode and posConditionlCode also, same logic can be used for categorical encoding.

3) Datetime features:

There are some of the datetime columns. We cant treat them as categorical variables as those are not simply values and denotes periodicity in the data.

1) TransactionTime:

We can segment this feature to get more information like day of , month, year, day of week, hour in the day. So that we get 5 different features.

2) AccountOpenDate:

We can get current age of account from this feature by subtracting transactionDate from this feature.

3) currentExpDate:

This column denotes the expiration date of the account. We can calculate remaining age of the account at the current time using this feature.

4) dateOfLastAddrChange:

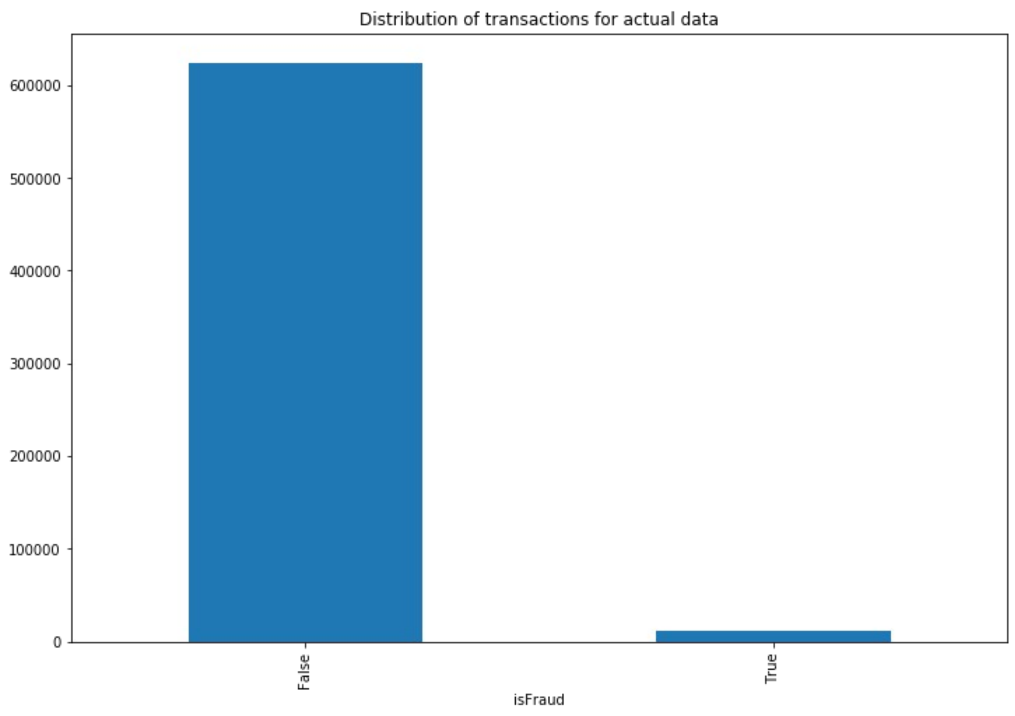
We can again find how long the address was last changed.

In this way, we extracted useful information from the datetime columns in the form of numbers and dropped the original datetime columns.

Model Building:

Now all of our columns are in the form of number of boolean values which can be handled by classification algorithm. After applying all the feature engineering, we now have 49 features. The number is increased than the previous because of the 1-hot encoded columns.

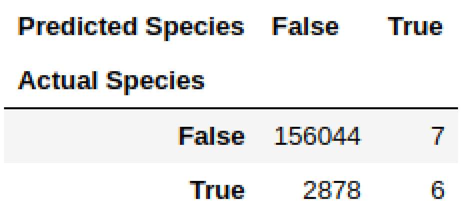
As this is fraud detection algorithm and odds of having fraud transactions are very less, we have highly unbalanced dataset.



When I built classification model (random forest) on this dataset, I got 98.17 accuracy.

The number looks pretty exciting. But, there is a catch. As we have 98.23% of data with a single class. So even if we predict all not_fraud, we will get 98.23% of accuracy. We are not able to detect the actual fraud transactions.

It is evident from the confusion matrix.



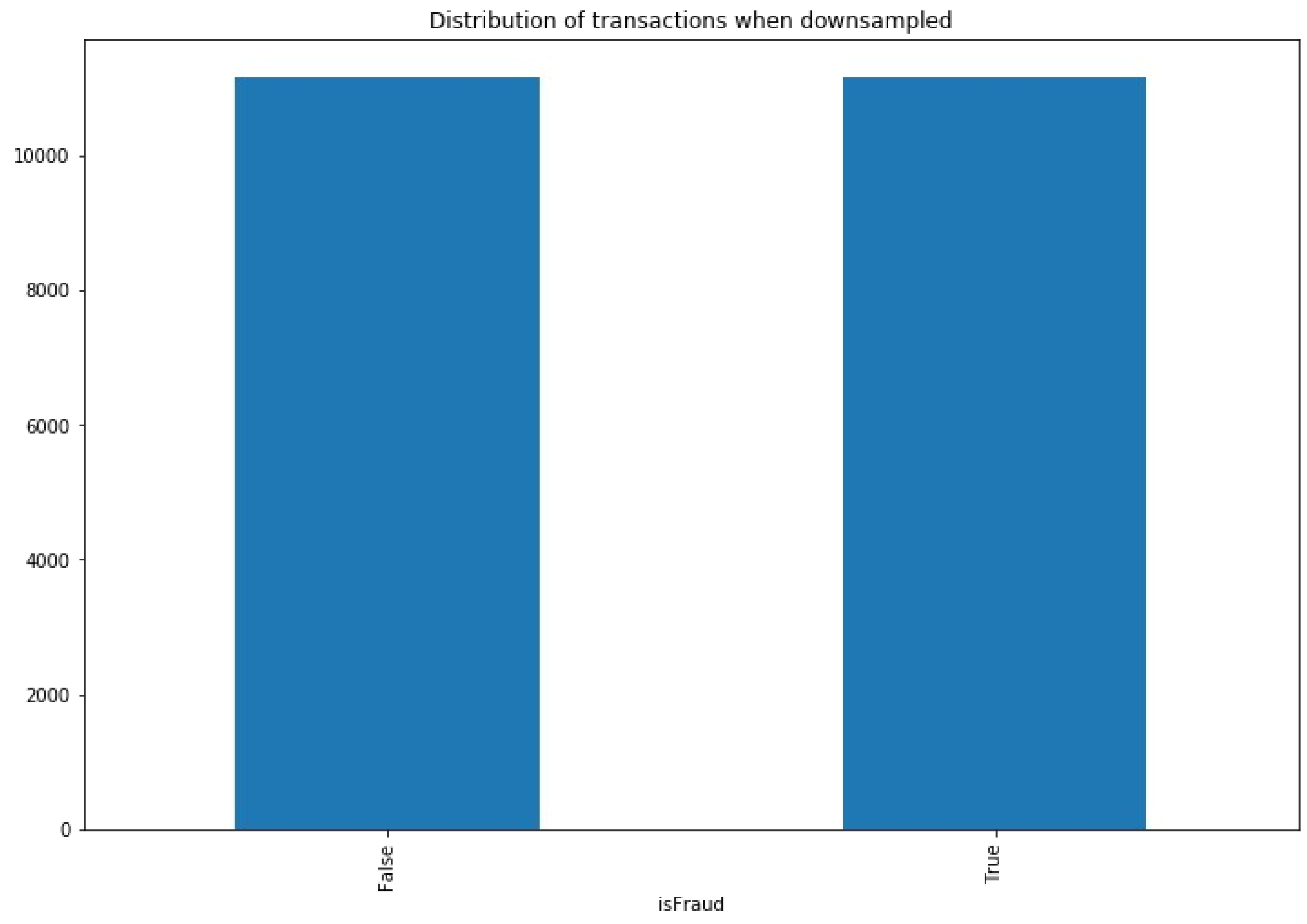
Predicted Species	False	True
Actual Species		
False	156044	7
True	2878	6

Only 13 rows where predicted fraud by the random forest algorithm. There are 2878 true negatives which is not useful in the business sense.
So accuracy score is not necessarily useful for imbalanced dataset. Recall score is 0.002 which is really bad to detect actual fraud transactions.

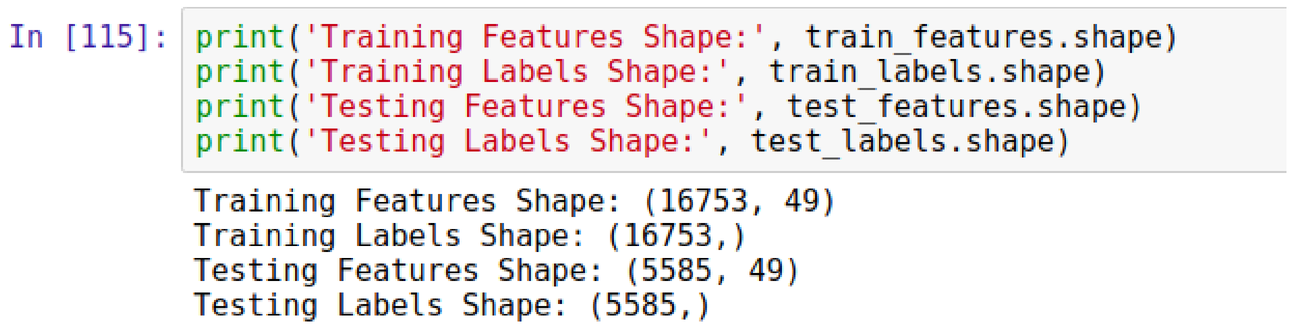
So we will have to modify the dataset. There are multiple methods to handle unbalanced data.

<u>1) Downsampling:</u>

In the down sampling approach, dataset is balanced by selecting the same number of majority class as that of minority class. So, we will select all the Fraud transactions and the same number of non-fraud transactions. We will build the classification model on the selected data.



Splitted the dataframe in 75:25. 75% of the data is used to train the classification model and 25% is used to test the model.



```
In [115]: print('Training Features Shape:', train_features.shape)
          print('Training Labels Shape:', train_labels.shape)
          print('Testing Features Shape:', test_features.shape)
          print('Testing Labels Shape:', test_labels.shape)

          Training Features Shape: (16753, 49)
          Training Labels Shape: (16753,)
          Testing Features Shape: (5585, 49)
          Testing Labels Shape: (5585,)
```

For classification, we have choices like decision tree, random forest, logistic regression, naive bayes, SVM and neural network.

Random Forest:
Random forest is ensambled algorithm derived by combining multiple decision trees. Decision tree builds a tree based on feature significance. Closer the feature to the root, higher its significance. However, decision tree is prune to overfitting (variance between accuracies on different datasets in high) for more features as tree depth is more. So random forest decides the output by building multiple decision trees in parallel and selecting the majority output in case of classification (mean in case of regression).

As our data also has 48 features, it is prune to over-fitting and so, we will start with random forest.

I used gridsearch to select best hyperparameters.

```
param_rf = {"max_depth": [3, None],
            "max_features": [1, 3, 10],
            "min_samples_split": [2, 3, 10],
            "bootstrap": [True, False],
            "criterion": ["gini", "entropy"]}
```

Also applied cross-validation with CV=5 along with gridsearch. It will select the best hyperparameters and crossvalidate each one of them 5 times.

```
Performance of random forest with undersampling and grid search on test set

Precision score:  0.6725082199530464
Recall score:  0.6722143498332905
Accuracy Score:  0.6721575649059982
F-1 Score:  0.6720349258629439
```

```
Performance of random forest with undersampling and grid search on train set

Precision score:  0.6772364417051547
Recall score:  0.6769350581229894
Accuracy Score:  0.6769533814839134
F-1 Score:  0.6768098912662224
```

Precision :
The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The best value is 1 and the worst value is 0.

Recall:
The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.
The best value is 1 and the worst value is 0.

F-1 Score:
The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

F1 = 2 * (precision * recall) / (precision + recall)

Accuracy Score:
It is the fraction of predicted labels that matches with the actual label.

From the above definitions, it is clear that we want our **recall 1 or closer to 1**. As we want to detect the fraud transactions rather than non-fraud transactions. **We need to minimize (ideally 0) the false negatives and maximize the true positives.**

We can observe that random forest with undersampling is performing much better than the previous approach. Recall and precision scores are 67%. However, detecting only 2 out of 3 frauds in not good in the fraud transaction detection and we can say that RF is performing poorly with the undersampled data. It has very low test as well as train error. It has high bias and it is not catching all the patterns in the dataset.

Logistic regression:
Now lets move to another popular algorithm which is logistic regression.
The hyper parameters are selected from these

```
lr_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100]}
grid_lr = GridSearchCV(LogisticRegression(), lr_params)
```

The grid search is applied with CV=5 (as explained for the random forest)

Performance of logistic regression with undersampling and grid search on train set

```
Precision score:  0.6786456735902127
Recall score:  0.6776968762627253
Accuracy Score:  0.6777293619053304
F-1 Score:  0.6772902209936856
```

Performance of Logistic regression with undersampling and grid search on test set

```
Precision score:  0.6769069738127258
Recall score:  0.6760143626570916
Accuracy Score:  0.67591763652641
F-1 Score:  0.6755381607962886
```

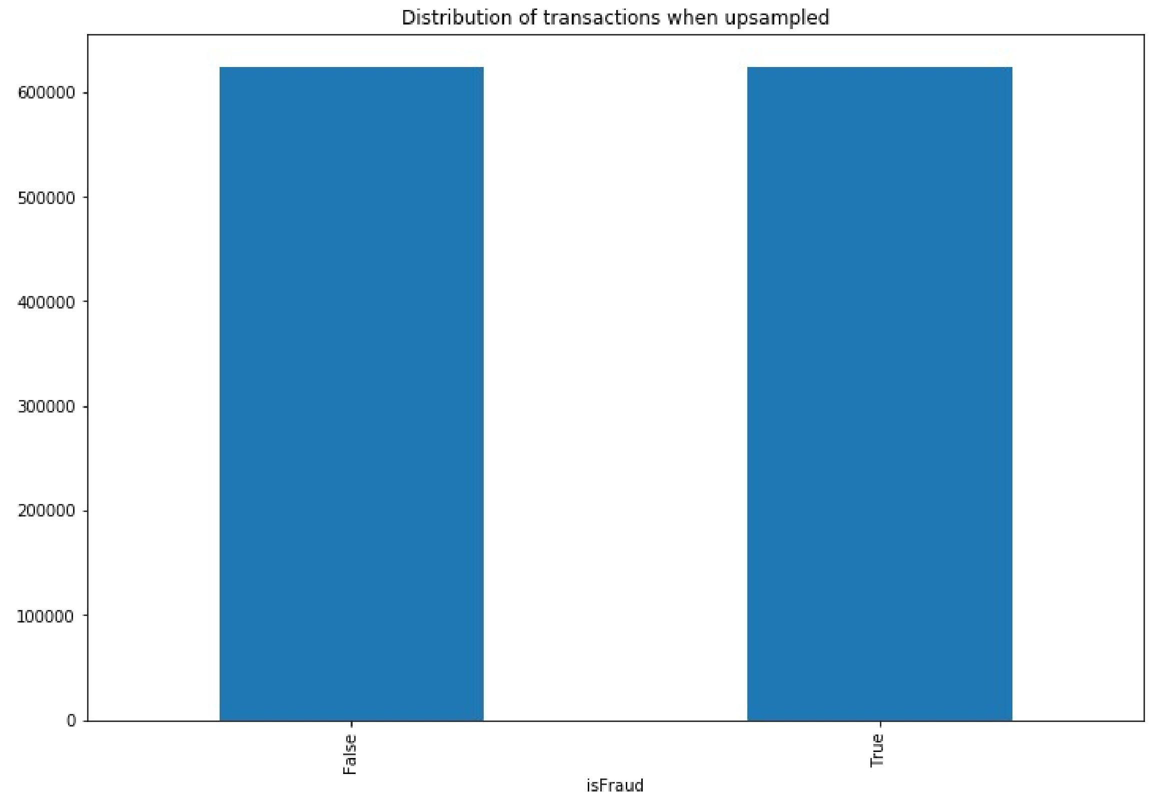
These are the results which we are getting on logistic regression. Even this algorithm is performing poorly with the undersampled data. It also has high bias.

The problem with undersampled data is, it removes more than 95% of data. Training is done on only 3% of available data. So, we might miss important patterns. Also non-fraud data is randomly selected. So the error is subject to change based on the selected data.

There is another approach to handle unbalanced dataset

2) Oversampling (Upsampling):

In this case, we synthetically create the minority class and make it equal to the majority class without removing any of the majority class examples that we did for undersampling. Advantage of this method is we are not lossing any information from the dataset and 100% of data can be used.

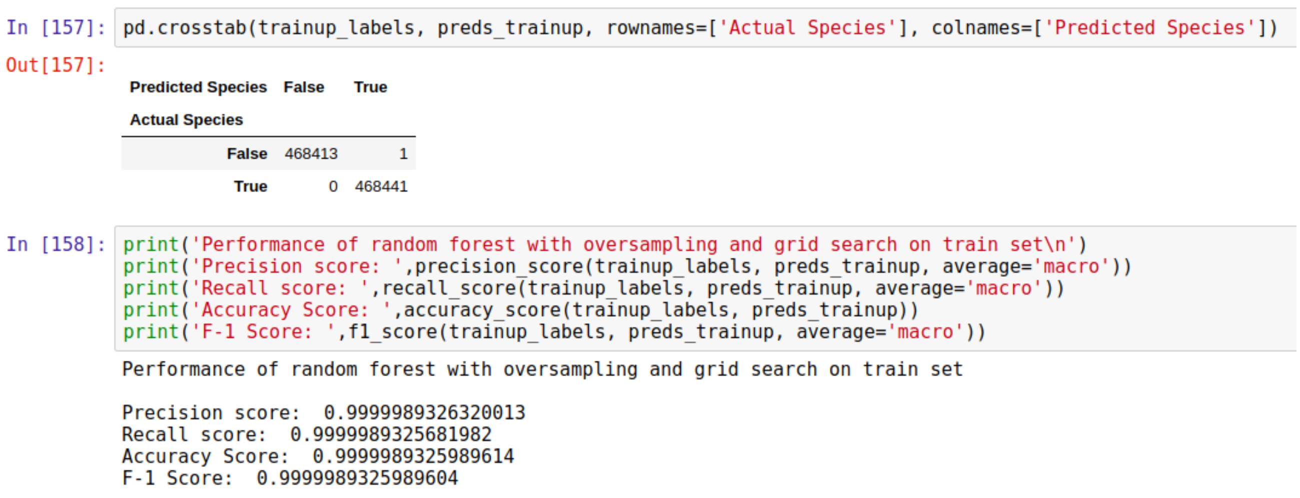


The ratio of the classes is the same but, number of examples are different.

We splited the data by 75:25 as we did for the last approach. Instead of applying gridsearch again, I used the same hyperparameters which were obtained for the last approach.

Random Forest:

Train set performance:

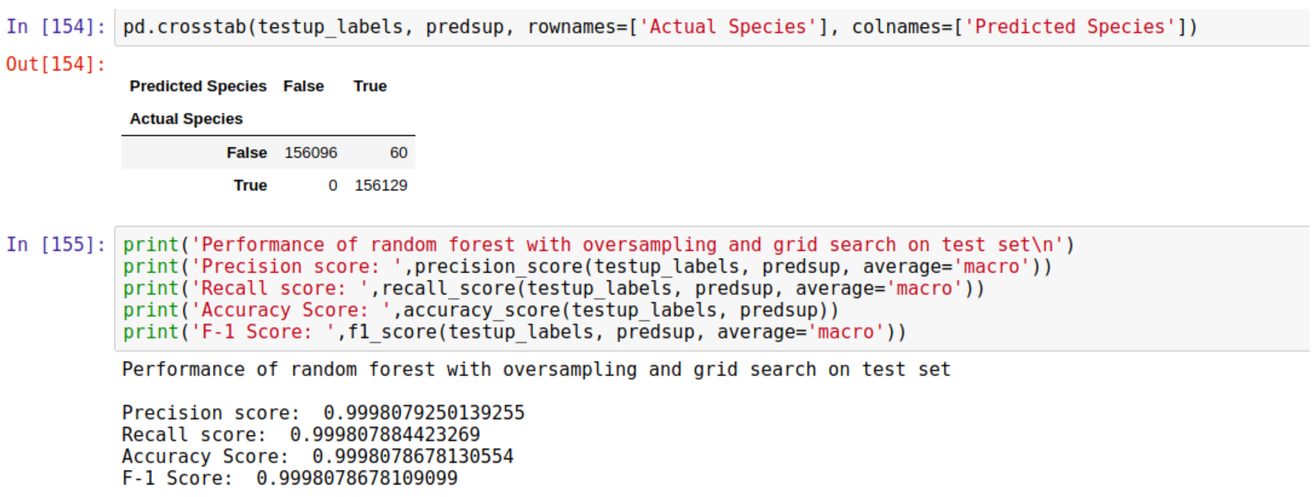


```
In [157]: pd.crosstab(trainup_labels, preds_trainup, rownames=['Actual Species'], colnames=['Predicted Species'])
Out[157]:
```

Predicted Species	False	True
Actual Species		
False	468413	1
True	0	468441

```
In [158]: print('Performance of random forest with oversampling and grid search on train set\n')
          print('Precision score: ',precision_score(trainup_labels, preds_trainup, average='macro'))
          print('Recall score: ',recall_score(trainup_labels, preds_trainup, average='macro'))
          print('Accuracy Score: ',accuracy_score(trainup_labels, preds_trainup))
          print('F-1 Score: ',f1_score(trainup_labels, preds_trainup, average='macro'))

Performance of random forest with oversampling and grid search on train set

Precision score:  0.9999989326320013
Recall score:  0.9999989325681982
Accuracy Score:  0.9999989325989614
F-1 Score:  0.9999989325989604
```

Test set performance:



```
In [154]: pd.crosstab(testup_labels, predsup, rownames=['Actual Species'], colnames=['Predicted Species'])
Out[154]:
```

Predicted Species	False	True
Actual Species		
False	156096	60
True	0	156129

```
In [155]: print('Performance of random forest with oversampling and grid search on test set\n')
          print('Precision score: ',precision_score(testup_labels, predsup, average='macro'))
          print('Recall score: ',recall_score(testup_labels, predsup, average='macro'))
          print('Accuracy Score: ',accuracy_score(testup_labels, predsup))
          print('F-1 Score: ',f1_score(testup_labels, predsup, average='macro'))

Performance of random forest with oversampling and grid search on test set

Precision score:  0.9998079250139255
Recall score:  0.999807884423269
Accuracy Score:  0.9998078678130554
F-1 Score:  0.9998078678109099
```

We can observe here that random forest is performing very excellent on the oversampled data. For the train set out of 937K examples, only 1 is classified in the wrong class, giving more than 99.99% accuracy. Most importantly, it is not giving a single true negative. It is correctly detecting all the frauds.

With the test set also, no true negatives are predicted. Accuracy, precision and recall score percentage is more than 99.99.

So, oversampled data works the best with random forest with hyperparameter tuning. It is detecting every fraud correctly and very negligible false positives which can be tollerated in the fraud transaction detection where detecting fraud is more important than not detecting a fraud transaction.

Alternative approaches I could use /though of using:

1) I got very good accuracy with oversampling the data with random forest model. I could have explored other algorithms like SVM, logistic regression, neural network as well for this data.

2) With undersampling, I tried to use SVM. But, gridsearch was taking a lot of time to build the model with the best hyperparameters. To reduce the computation cost I could use big data tool like PySpark for SVM. PySpark also provides the functionality of gridsearch.

3) I could explore more algorithms like neural network using tensorflow or keras. It could be used using pyspark as well to reduce computation time.

4) I though of using naive bayes for undersampled data. However after observing performances of logistic regression and random forest I directly jumped to oversampling approach. Naive bayes assumes that all the features have equal significance and they are independent of each other which wasnt the case in our data. So, I dropped the plan to use naive bayes.

5) There is a version of Random forest which is called isolation forest. It is used to detect the anomalies in the dataset. I could use this method to find the anomalous transactions.

How to run the code:

I am attaching jupyter notebook file (fraud_detection.ipynb)

You can run the cell by cell and observe results for every step. Every operation is written in individual cells. You will need Ipython and anaconda with python3 kernel to run the notebook file.

It can be downloaded from here.

All the instruction are given in the above link. Once you are done with the installation, type ‘jupyter notebook’ on the terminal/command prompt.

Search the downloaded fraud_detection.ipynb file and open it. Please keep the data in the same folder as that of notebook file.

You can also hide the code and observe only outputs. There is a button at the bottom of first cell.

References:

1) scikit learn documentation
2) Pandas documentation
3) Matplotlib documentation