Contents

Project Definition & Introduction

Google Store (Gstore) is an online hardware retailer operated by Google. Lots of people visit Google Store every day, which generates h11uge volumes of data. Using Google analytics, each transaction in GStore has been tracked and reported, including website traffic, transaction data, customer buying behavior and customer demographics. This data can be valuable when Google analyzes their customer base and design marketing strategies.

In this project, we use the Google Store dataset to predict revenue per customer. This outcome is useful not only for Google but also for companies who want to make use of Google Analytics to assist data analysis, informing them with actionable promotional efforts and marketing investments.

Solution Overview

Firstly, we want to know whether a customer will return to the store in the next 2 months. So we build a classification model to predict whether a customer will come back or not. If the customer returns in the next two months, we mark it as the return group. Otherwise, we mark it as the non-return group. In this step, we build three models, including random forest, XGBoost and LightGBM. We use accuracy to evaluate random forest and XGBoost model and binary cross-entropy to evaluate LightGBM model. After identifying the return group customers, we are interested in how much they will spend. Then we build another regression model to predict how much one customer will spend in the Google Store. In this process, we build three models, including random forest, XGBoost and LightGBM. We use RMSE to evaluate our models.

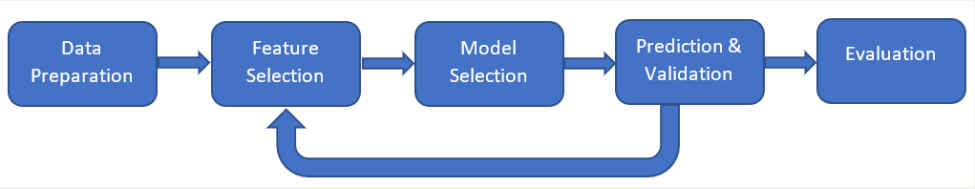
Technical Specifications

JUPYTER NOTEBOOK & PYTHON API
Scikit Learn version 0.21.3
Numpy version 1.17.4
Pandas version 0.24.2
LightGBM version 2.3.0
MatplotLib version 3.0.3

JSON version 2.0.9
Seaborn version 0.9.0
CSV version 1.0
XGBoost version 0.90

Analysis

Methodology



The problem in hand was to predict the future two months revenue of customers using their 5 months historical purchase data. Hence, the first major step in our analysis was to make training data simulate the problem we are trying to solve as it involves predicting customer revenues for the future two months after 42 days of gap between available data. As we didn't have any information about the lagged time of 42 days; we prepared our data to follow the same pattern by adding two columns using available dataset; one whether the customer will return to the site or not after the lagged time and if returned what would be their average spend.

Once we had the strategy set, the next step was to clean the data so that to get much of the information out from the JSON columns, removing variables which are not adding to any information gain and encoding categorical variables to be used in models. Next step was to select algorithms to test our predictions on. Also, since we have to predict both the return label and revenue, we built two models; one for classification and other for revenue prediction. We choose tree-based models as they use bagging and boosting techniques to improve predictions and are more efficient as they work on a subset of data for building each tree. We tried three models viz. Random Forest, XG Boost and Light GBM on the train set with 30 variables and checked for the feature important variables to further subset our features. This helped us improve both the prediction accuracy/RMSE and computation time.

Next for model evaluation, we just multiplied our classification output (which is whether a customer will return) with the regression output (which predicts the expected revenue value). After checking for the RMSE value for the three models we have built, Light GBM gave us the best RMSE of **0.88188**.

Data Preparation

Loading training & testing data

Firstly, we need to load the data. The size of the train set and test set are both very large. The size of the training set is 23 GB and the size of the test set is 7 GB, which makes it difficult to directly read and process in local machines. Therefore, we utilized chunksize in Google Colab to solve this problem.

```
file1 = 'test_v2.csv'
JSON_COLUMNS = ['device', 'geoNetwork', 'totals', 'trafficSource']
df_chunk = pd.read_csv(file1, chunksize=1000000,
                       dtype={'fullVisitorId': 'str'})
```

After loading the data in chunksize, we later contacted the chunks back to get the full dataset.

```
chunk_list1 = []

for chunk in df_chunk:
    chunk_list1.append(chunk)

test = pd.concat(chunk_list1)
```

We found 4 columns containing complex values in JSON format: device, geoNetwork, totals and traffic source. For the following analysis, we need to extract and flatten the variables and values in these JSON columns into as individual variables. We used the json_normalize function to flatten these columns. After the preprocessing, we output the train and test data into csv file for the following analysis.

```
df2 = pd.read_csv('test_json.csv',
                  converters={column: json.loads for column in JSON_COLUMNS},
                  dtype={'fullVisitorId': 'str'}, # Important!!
                  )
for column in JSON_COLUMNS:
        column_as_df = json_normalize(df2[column])
        column_as_df.columns = [f"{column}.{subcolumn}" for subcolumn in column_as_df.columns]
        df2 = df2.drop(column, axis=1).merge(column_as_df, right_index=True, left_index=True)
```

Cleaning and Transformation

Currently, the data is at the visit level, where each row represents a visit to the Google store website, but for prediction, we need to predict on customer level. Therefore, we need to transform the data by aggregate the visit data by customer.

Below are the steps we take to transform the data.

Step #1: remove columns with constant values

```
col_with_constant = pd.DataFrame({'uniq_counts': [col for col in train.columns if train[col].nunique() == 1]})
print("Columns with constant value: ", len(col_with_constant), "columns")
print("Name of constant columns: \n", col_with_constant)
```

```
Columns with constant value:  24 columns
Name of constant columns:
                                          uniq_counts
0                                socialEngagementType
1                              device.browserVersion
2                                 device.browserSize
3                          device.operatingSystemVersion
4                            device.mobileDeviceBranding
5                               device.mobileDeviceModel
6                             device.mobileInputSelector
7                                device.mobileDeviceInfo
8                        device.mobileDeviceMarketingName
9                                 device.flashVersion
10                                   device.language
11                                device.screenColors
12                             device.screenResolution
13                                   geoNetwork.cityId
14                                geoNetwork.latitude
15                               geoNetwork.longitude
16                            geoNetwork.networkLocation
17                                     totals.visits
18                                    totals.bounces
19                                   totals.newVisits
20  trafficSource.adwordsClickInfo.criteriaParameters
21                             trafficSource.isTrueDirect
22            trafficSource.adwordsClickInfo.isVideoAd
23                             trafficSource.campaignCode
```

```
# drop columns with constant value from train data
train = train.drop(columns = constant_cols, axis=1)
```

```
# drop columns with constant value from test data
col_with_constant_t = pd.DataFrame({'uniq_counts': [col for col in test.columns if test[col].nunique() == 1]})
print("Columns with constant value: ", len(col_with_constant_t), "columns")
print("Name of constant columns: \n", col_with_constant_t)
```

Except for numerical variables ('totals.visits', 'totals.bounces', 'totals.newVisits'), we dropped these columns for both train and test datasets.

```
drop = ['totals.visits', 'totals.bounces', 'totals.newVisits']
for i in drop:
    constant_cols.remove(i)
```

```
num_cols = ['totals.visits', 'totals.hits', 'totals.pageviews', 'totals.bounces', 'totals.newVisits', 'totals.sessionQ
ualityDim', 'totals.timeOnSite', 'totals.transactions', 'totals.transactionRevenue', 'totals.totalTransactionRevenue']

for col in num_cols:
    data[col] = data[col].fillna(0)
```

Step #2: convert the date value into datetime format, and combine the train and test set together for following aggregation and transformation

```
# Transfer datetime data
def date_converter(df):
    df['date'] = df['date'].astype(str)
    df["date"] = df["date"].apply(lambda x : x[:4] + "-" + x[4:6] + "-" + x[6:])
    df["date"] = pd.to_datetime(df["date"])

    return df
```

Step #3: replace missing values in numerical columns with 0 as we are taking sum of the rows in the following aggregation process. We also converted the binary column into 0 and 1.

```
def advance_tranformation(data, k):
    date_min = data['date'].min()
    date_max = data['date'].max()
    df = data[(data['date'] >= date_min + timedelta(168 * (k-1))) & (data['date'] < date_min + timedelta(days=168 * k))]
    df_future_id = pd.unique(data[(data['date'] >= date_min + timedelta(days = 168 * k + 46)) & (data['date'] < date_min + timedelta(days=168 * k + 46 + 62))]['fu
llVisitorId'])
    df_return = df[df['fullVisitorId'].isin(df_future_id)]
    return_unique = pd.unique(df_return['fullVisitorId'])
    df_test = data[(data['fullVisitorId'].isin(return_unique))]
    df_test = df_test[(df_test['date'] >= date_min + timedelta(days = 168 * k + 46)) & (df_test['date'] < date_min + timedelta(days=168 * k + 46 + 62))]
    df_target = df_test.groupby('fullVisitorId').apply(lambda x: np.log1p(sum(x['totals.transactionRevenue']))).reset_index()
    df_target['ret'] = 1
    df_target = df_target.rename(columns= {0: 'target'})
    df_not_return = df[df['fullVisitorId'].isin(df_future_id) == False]
    not_return_unique = pd.unique(df_not_return['fullVisitorId'])
    df_not_ret_df = pd.DataFrame({'fullVisitorId': not_return_unique, 'target': 0, 'ret': 0})
    target_df = pd.concat([df_target, df_not_ret_df])

    df_group = pd.DataFrame()

    df_group['channelGrouping'] = df.groupby('fullVisitorId').agg(lambda x: x['channelGrouping'].mode()[0])['channelGrouping']
    df_group['first_ses_from_the_period_start'] = df['date'].min() - date_min
    df_group['last_ses_from_the_period_end'] = date_max - df['date'].max()
    df_group['interval_dates'] = df['date'].max() - df['date'].min()
    df_group['unique_date_num'] = len(pd.unique(df['date']))
    df_group['maxVisitNum'] = df.groupby('fullVisitorId')['totals.visits'].agg("max")
    df_group['browser'] = df.groupby('fullVisitorId').agg(lambda x: x['device.browser'].mode()[0])['device.browser']
    df_group['operatingSystem'] = df.groupby('fullVisitorId').agg(lambda x: x['device.operatingSystem'].mode()[0])['device.operatingSystem']
    df_group['deviceCategory'] = df.groupby('fullVisitorId').agg(lambda x: x['device.deviceCategory'].mode()[0])['device.deviceCategory']
    df_group['continent'] = df.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.continent'].mode()[0])['geoNetwork.continent']
    df_group['subContinent'] = df.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.subContinent'].mode()[0])['geoNetwork.subContinent']
    df_group['country'] = df.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.country'].mode()[0])['geoNetwork.country']
    df_group['region'] = df.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.region'].mode()[0])['geoNetwork.region']
    df_group['metro'] = df.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.metro'].mode()[0])['geoNetwork.metro']
    df_group['city'] = df.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.city'].mode()[0])['geoNetwork.city']
    df_group['networkDomain'] = df.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.networkDomain'].mode()[0])['geoNetwork.networkDomain']
    df_group['source'] = df.groupby('fullVisitorId').agg(lambda x: x['trafficSource.source'].mode()[0])['trafficSource.source']
    df_group['medium'] = df.groupby('fullVisitorId').agg(lambda x: x['trafficSource.medium'].mode()[0])['trafficSource.medium']
    df_group['isMobile']= df.groupby('fullVisitorId')['device.isMobile'].agg("sum")
    df_group['bounce_sessions'] = df.groupby('fullVisitorId')['totals.bounces'].agg("sum")
    df_group['hits_sum'] = df.groupby('fullVisitorId')['totals.hits'].agg("sum")
    df_group['hits_mean'] = df.groupby('fullVisitorId')['totals.hits'].agg("mean")
    df_group['pageviews_sum'] = df.groupby('fullVisitorId')['totals.pageviews'].agg("sum")
    df_group['pageviews_mean'] = df.groupby('fullVisitorId')['totals.pageviews'].agg("mean")
    df_group['session_cnt'] = len(df['visitStartTime'])
    df_group['transactionRevenue'] = df.groupby('fullVisitorId')['totals.transactionRevenue'].agg("sum")
    df_group['transactions'] = df.groupby('fullVisitorId')['totals.transactions'].agg("sum")

    df_group = pd.merge(df_group, target_df, on='fullVisitorId', sort=False)


    return df_group
```

Step #4: divide the data by 5.5 window, aggregate the data into customer level and create return and target value label.

The test set contains data for more than 2 years, the test set is composed of data in 5.5 months (168 days), and the prediction period is 2 months (62 days). The different length of time period will affect the prediction result. Therefore, we divided the train data into 4 chunks to match the length of the test data.

Since our method is to first predict if the customer will return in the predicing 2 months, we need to create the return label of it. To simulate the time difference between train and test dataset (46 days), and the training period (62 days), we create the return label as 1 if the customer did appear in the period of time + 46 + 62 days, and 0 if he/she didn't.

If the customer did return, the next step is to predict how much he/she will spend, so we created another new variable call target, which is the log transaction revenue of that customer in the time+46_62 days period (the future transaction revenue).

```
tr_1 = advance_tranformation(data, 1)
tr_2 = advance_tranformation(data, 2)
tr_3 = advance_tranformation(data, 3)
tr_4 = advance_tranformation(data, 4)
```

Through the transformation, we will have 4 chunks of data call tr_1 to tr_4 by segmenting the train data.

```
# Build testing dataset
tr_5 = data[data['date'] >= '2018-05-01']
tr_5_maxdate = tr_5['date'].max()
tr_5_mindate = tr_5['date'].min()

df_group_5 = pd.DataFrame()

df_group_5['channelGrouping'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['channelGrouping'].mode()[0])['channelGrouping']
df_group_5['first_ses_from_the_period_start'] = tr_5['date'].min() - tr_5_mindate
df_group_5['last_ses_from_the_period_end'] = tr_5_maxdate - tr_5['date'].max()
df_group_5['interval_dates'] = tr_5['date'].max() - tr_5['date'].min()
df_group_5['unique_date_num'] = len(pd.unique(tr_5['date']))
df_group_5['maxVisitNum'] = tr_5.groupby('fullVisitorId')['totals.visits'].agg("max")
df_group_5['browser'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['device.browser'].mode()[0])['device.browser']
df_group_5['operatingSystem'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['device.operatingSystem'].mode()[0])['device.operatingSystem']
df_group_5['deviceCategory'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['device.deviceCategory'].mode()[0])['device.deviceCategory']
df_group_5['continent'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.continent'].mode()[0])['geoNetwork.continent']
df_group_5['subContinent'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.subContinent'].mode()[0])['geoNetwork.subContinent']
df_group_5['country'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.country'].mode()[0])['geoNetwork.country']
df_group_5['region'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.region'].mode()[0])['geoNetwork.region']
df_group_5['metro'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.metro'].mode()[0])['geoNetwork.metro']
df_group_5['city'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.city'].mode()[0])['geoNetwork.city']
df_group_5['networkDomain'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['geoNetwork.networkDomain'].mode()[0])['geoNetwork.networkDomain']
df_group_5['source'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['trafficSource.source'].mode()[0])['trafficSource.source']
df_group_5['medium'] = tr_5.groupby('fullVisitorId').agg(lambda x: x['trafficSource.medium'].mode()[0])['trafficSource.medium']
df_group_5['isMobile']= tr_5.groupby('fullVisitorId')['device.isMobile'].agg("sum")
df_group_5['bounce_sessions'] = tr_5.groupby('fullVisitorId')['totals.bounces'].agg("sum")
df_group_5['hits_sum'] = tr_5.groupby('fullVisitorId')['totals.hits'].agg("sum")
df_group_5['hits_mean'] = tr_5.groupby('fullVisitorId')['totals.hits'].agg("mean")
df_group_5['pageviews_sum'] = tr_5.groupby('fullVisitorId')['totals.pageviews'].agg("sum")
df_group_5['pageviews_mean'] = tr_5.groupby('fullVisitorId')['totals.pageviews'].agg("mean")
df_group_5['session_cnt'] = len(tr_5['visitStartTime'])
df_group_5['transactionRevenue'] = tr_5.groupby('fullVisitorId')['totals.transactionRevenue'].agg("sum")
df_group_5['transactions'] = tr_5.groupby('fullVisitorId')['totals.transactions'].agg("sum")
df_group_5['target'] = np.nan
df_group_5['return'] = np.nan
```

Then, we performed the same transformation on the testing period. However, there's one difference: Since we don't know whether the customer will return and how much he/she will spend with this time period (It's the value we aim to predict), we put NaN in the target and return label.

```
train_combined = pd.concat([tr_1, tr_2, tr_3, tr_4, tr_5], sort=False)
```

```
train_combined.columns
```

```
Index(['fullVisitorId', 'channelGrouping', 'first_ses_from_the_period_start',
       'last_ses_from_the_period_end', 'interval_dates', 'unique_date_num',
       'maxVisitNum', 'browser', 'operatingSystem', 'deviceCategory',
       'continent', 'subContinent', 'country', 'region', 'metro', 'city',
       'networkDomain', 'source', 'medium', 'isMobile', 'bounce_sessions',
       'hits_sum', 'hits_mean', 'pageviews_sum', 'pageviews_mean',
       'session_cnt', 'transactionRevenue', 'transactions', 'ret', 'target',
       'date', 'device.browser', 'device.deviceCategory', 'device.isMobile',
       'device.operatingSystem', 'geoNetwork.city', 'geoNetwork.continent',
       'geoNetwork.country', 'geoNetwork.metro', 'geoNetwork.networkDomain',
       'geoNetwork.region', 'geoNetwork.subContinent', 'totals.bounces',
       'totals.hits', 'totals.newVisits', 'totals.pageviews',
       'totals.sessionQualityDim', 'totals.timeOnSite',
       'totals.totalTransactionRevenue', 'totals.transactionRevenue',
       'totals.transactions', 'totals.visits', 'trafficSource.adContent',
       'trafficSource.adwordsClickInfo.adNetworkType',
       'trafficSource.adwordsClickInfo.gclId',
       'trafficSource.adwordsClickInfo.page',
       'trafficSource.adwordsClickInfo.slot', 'trafficSource.campaign',
       'trafficSource.keyword', 'trafficSource.medium',
       'trafficSource.referralPath', 'trafficSource.source', 'visitId',
       'visitNumber', 'visitStartTime'],
      dtype='object')
```

```
drop_col = ['date', 'device.browser', 'device.deviceCategory', 'device.isMobile',
       'device.operatingSystem', 'geoNetwork.city', 'geoNetwork.continent',
       'geoNetwork.country', 'geoNetwork.metro', 'geoNetwork.networkDomain',
       'geoNetwork.region', 'geoNetwork.subContinent', 'totals.bounces',
       'totals.hits', 'totals.newVisits', 'totals.pageviews',
       'totals.sessionQualityDim', 'totals.timeOnSite',
       'totals.totalTransactionRevenue', 'totals.transactionRevenue',
       'totals.transactions', 'totals.visits', 'trafficSource.adContent',
       'trafficSource.adwordsClickInfo.adNetworkType',
       'trafficSource.adwordsClickInfo.gclId',
       'trafficSource.adwordsClickInfo.page',
       'trafficSource.adwordsClickInfo.slot', 'trafficSource.campaign',
       'trafficSource.keyword', 'trafficSource.medium',
       'trafficSource.referralPath', 'trafficSource.source', 'visitId',
       'visitNumber', 'visitStartTime']
```

```
train_combined_2 = train_combined.drop(columns=drop_col)
```

We then combined the data from tr_1 to tr_5 again and dropped duplicate columns.

Step #5: deal with categorical variables.

We observed that there are several features with categorical values, for instance, city, channelGrouping and other text data. In order to achieve better model performance, choosing an appropriate encoding of categorical data is necessary. The Python scikit-learn provides a wide range of encoding options in Category Encoder packages. However, we find that most of the nominal data are equipped with more than 4 categories; for example, the feature “city” includes 981 distinct cities. In order to deal with this problem, we consider 3 different encoders: OneHotEncoding, TargetEncoding and LabelEncoding.

```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
c = ['city','browser','channelGrouping',"operatingSystem","continent",'subContinent','country',
     'region','metro','networkDomain','source','medium', 'deviceCategory']
for col in c:
    train_combined_2[col] = train_combined_2[col].fillna('na')
    train_combined_2[col] = le.fit_transform(train_combined_2[col].values)
```

There are different encoding methods we can use: target encoding, one hot encoding and label encoding. First of all, when a feature with several nominal data, OneHot encoding will create a large amount of columns with 0, 1. Since we have 27 categorical variables, most of which have hundreds of different levels. The one hot encoding gives us over 5000 columns and very sparse data.

Secondly, we are considering target-based encoding which is suitable to be utilized when the number of categories is higher than 4. Nevertheless, it is not suitable to be utilized because the test dataset has no target variable. In other words, the non-numeric predictors in test dataset could not rely on target to be represented as numeric one. Also, it might generate over-fitting issues. As a result, we are using LabelEncoder to convert these discrete nominal data to numeric one.

```
train_combined_2['interval_dates'] = pd.to_numeric(train_combined_2['interval_dates'], errors= 'coerce')
train_combined_2['first_ses_from_the_period_start'] = pd.to_numeric(train_combined_2['first_ses_from_the_period_start'], errors= 'coerce')
train_combined_2['last_ses_from_the_period_end'] = pd.to_numeric(train_combined_2['last_ses_from_the_period_end'], errors= 'coerce')
```

```
train_combined_2.head()
```

	fullVisitorId	channelGrouping	first_ses_from_the_period_start	last_ses_from_the_period_end	interval_dates	unique_date_num	maxVisitNum
0	0000018966949534117	4	0	0	14428800000000000	168.0	1.0
1	0000039738481224681	2	0	0	14428800000000000	168.0	1.0
2	0000073585230191399	4	0	0	14428800000000000	168.0	1.0
3	0000087588448856385	4	0	0	14428800000000000	168.0	1.0
4	0000149787903119437	4	0	0	14428800000000000	168.0	1.0

To record the time difference issues in different data segments, we created date interval columns (interval_dates, first_ses_from_the_period_start, andlast_ses_from_the_period_end). The date interval columns are still in str format, so we converted it into numerical formats for the modeling process later.

```
# Divide train/ test
train = train_combined_2[train_combined_2['target'].isna()]
test = train_combined_2[train_combined_2['target'].isna() == False]
```

```
train.sort_values(by='first_ses_from_the_period_start')
```

	fullVisitorId	channelGrouping	first_ses_from_the_period_start	last_ses_from_the_period_end	interval_dates	unique_date_num	maxVi
0	7460955084541987166	4	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
133857	9900874527485563945	4	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
133858	6037578503108830215	2	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
133859	8630812730608319988	4	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
133860	7827975973225241449	4	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
...	...	...	...	...	...	...	...
267729	7476629900831012268	6	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
267730	1184221904466941690	5	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
267731	7875020615350219520	2	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
267733	8192330271104386623	4	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	
401588	3875690118293601911	2	-9223372036854775808	-9223372036854775808	-9223372036854775808	NaN	

803178 rows × 30 columns

```
test.head()
```

	fullVisitorId	channelGrouping	first_ses_from_the_period_start	last_ses_from_the_period_end	interval_dates	unique_date_num	maxVisitNum
0	0000018966949534117	4	0	0	14428800000000000	168.0	1.0
1	0000039738481224681	2	0	0	14428800000000000	168.0	1.0
2	0000073585230191399	4	0	0	14428800000000000	168.0	1.0
3	0000087588448856385	4	0	0	14428800000000000	168.0	1.0
4	0000149787903119437	4	0	0	14428800000000000	168.0	1.0

```
train.shape
```

(803178, 30)

```
test.shape
```

(296530, 30)

```
train.to_csv("train_final.csv")
```

```
test.to_csv("test_final.csv")
```

After all the cleaning and transformation, we divided the data back into the original time periods, and output them into csv file.

Train & Test Split

Following our approach, we are going to run two separate models: one is classification, predicting whether a customer will return in the future two months or not; another is regression, predicting how much a returned customer will spend in the future two months.

For our classification model, we use train_test_split from Scikit-learn module to randomly split the Kaggle train data into training and validation datasets. For our regression model, we firstly filtered returned customers from the original train data, then we use the same technique to randomly split it into training and validation datasets.

```
# Split train and validation data

# for target variable = return
x_train_return, x_valid_return, y_train_return, y_valid_return = train_test_split(x_train_return, y_train_return, test
_size=0.33)

# for target variable = target
x_train_target, x_valid_target, y_train_target, y_valid_target = train_test_split(x_train_target, y_train_target, test
_size=0.33)
```

Model Selection & Validation

After label encoding and separating training and test dataset, some features like networkDomain, the data range is from 1 to 42,722 so it should be normalized first to avoid bad model performance. Therefore, the next step is to normalize the data so that all of the features were on the same scale. We use the MinMaxScaler in scikit-learn to scale the training, validation and test datasets. In addition, the nan value in columns is transferred to 0.

```
# Normalized training dataset

# Normalize train_target
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(x_train_target)
x_train_target_scaled = scaler.transform(x_train_target)
x_valid_target_scaled = scaler.transform(x_valid_target)

# Normalize train_return
scaler = MinMaxScaler()
scaler.fit(x_train_return)
x_train_return_scaled = scaler.transform(x_train_return)
x_valid_return_scaled = scaler.transform(x_valid_return)

# Normalize test dataset
scaler = MinMaxScaler()
scaler.fit(x_test)
x_test_scaled = scaler.transform(x_test)
```

```
# Deal with nan values
np.argwhere(np.isnan(x_train_return_scaled))
x_train_return_scaled = np.nan_to_num(x_train_return_scaled)

np.argwhere(np.isnan(x_valid_return_scaled))
x_valid_return_scaled = np.nan_to_num(x_valid_return_scaled)

np.argwhere(np.isnan(x_test_scaled))
x_test_scaled = np.nan_to_num(x_test_scaled)
```

Random Forest

The Random Forest model can solve both classification and regression problems. As a tree-based model, it works well to create many trees on the sub-samples of dataset and integrate all of them through averaging to improve accuracy.

Moreover, it provides the feature importance so that the accuracy rate and error rate could be improved after choosing high-correlated variables. Before tuning the parameters, for the classification model, the outcome is as following.

```
# Classification Report
y_pred_valid_return = clf.predict(x_valid_return_scaled)
print(classification_report(y_valid_return, y_pred_valid_return))
# Calculating R-square
print("Its R-square of random forest regression model is", reg.score(x_train_target_scaled, y_train_target))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      1.00    464923
           1       0.00      0.00      0.00      2877

    accuracy                           0.99    467800
   macro avg       0.50      0.50      0.50    467800
weighted avg       0.99      0.99      0.99    467800

Its R-square of random forest regression model is 0.4786546091953813
```

Concerning regression model before adjusting hyperparameters, its RMSE of validation data is 4.0032 and the result of RMSE in Kaggle is 1.1043. After tuning hyperparameters, the RMSE of validation data is **3.9018**.

Next we tune hyperparameters

```
from sklearn.model_selection import RandomizedSearchCV
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 500, num = 10)]

# Number of features to consider at every split
max_features = ['auto', 'sqrt']

# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 100, num = 11)]
max_depth.append(None)

# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]

# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]

# Method of selecting samples for training each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
print(random_grid)
```

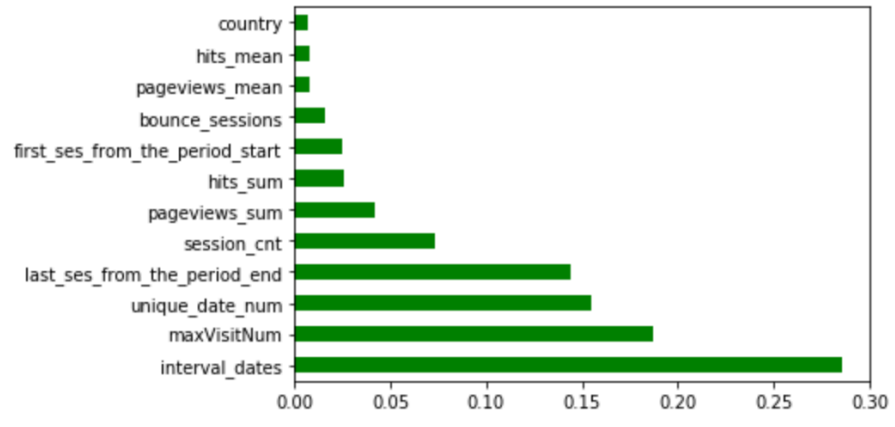
```
{'n_estimators': [200, 233, 266, 300, 333, 366, 400, 433, 466, 500], 'max_features': ['auto', 'sqrt'], 'max_depth':
[10, 19, 28, 37, 46, 55, 64, 73, 82, 91, 100, None], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1, 2, 4],
'bootstrap': [True, False]}
```

After tuning the hyperparameters, the below is tuned parameters in the optimal model.

```
reg2_random.best_params_
```

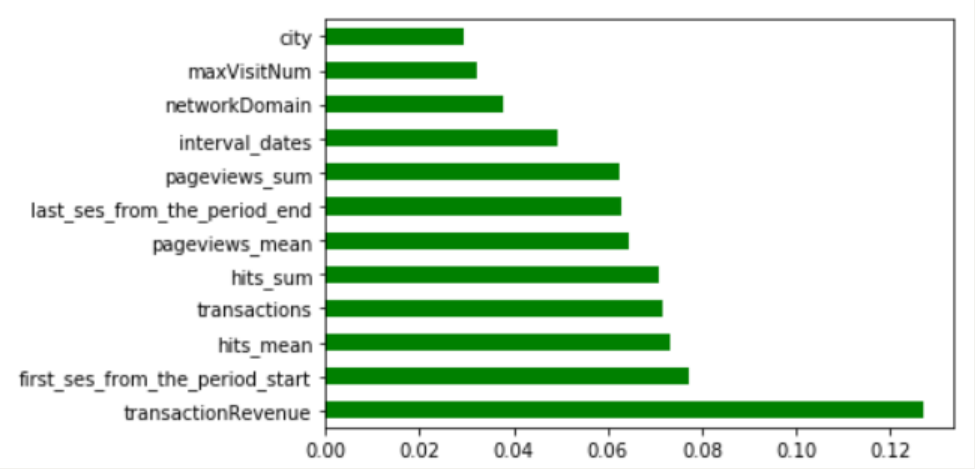
```
{'n_estimators': 200,
 'min_samples_split': 5,
 'min_samples_leaf': 2,
 'max_features': 'sqrt',
 'max_depth': 10,
 'bootstrap': True}
```

Next, we use the model with tuned hyperparameters to find the important features for both classification and regression models. For classification model, we select the features with importance higher than 0.05 to run the classification model again.



	precision	recall	f1-score	support
0	0.99	1.00	1.00	93699
1	0.60	0.01	0.02	568
accuracy			0.99	94267
macro avg	0.80	0.51	0.51	94267
weighted avg	0.99	0.99	0.99	94267

In the regression model, we also choose the features with importance higher than 0.05 to execute the regression model.



Finally, we use the model trained with tuned hyperparameters and obtain the result of RMSE for validation data is 3.4740. The outcome of the test dataset is 1.0744 submitted in Kaggle.

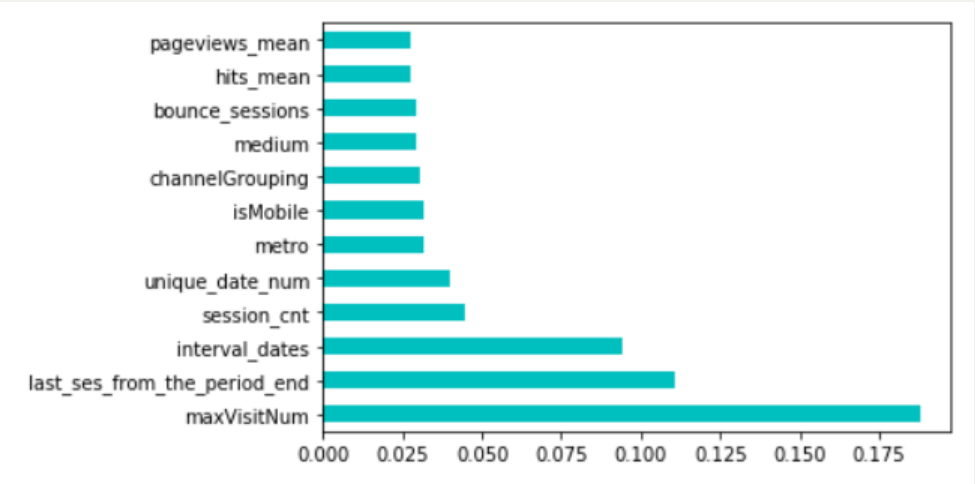
XGBoost

On the other hand, we perform XGBoost with the similar process as random forest. First, we train both classification and regression models with normalized training data with no nan value. The classification report is below.

```
# CLassification Report
print(classification_report(y_valid_return, y_pred_valid_return))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      1.00    464957
           1       0.43      0.02      0.04      2843

    accuracy                           0.99    467800
   macro avg       0.71      0.51      0.52    467800
weighted avg       0.99      0.99      0.99    467800
```

Additionally. the RMSE of regression model for validation data is 4.4074. Subsequently, we use feature importance given by two models. For classification model, we set the importance higher than 0.035 and decide our final classification model with features including maxVisitNum, last_ses_from_the_period_end, interval_dates, session_cnt, unique_date_num.

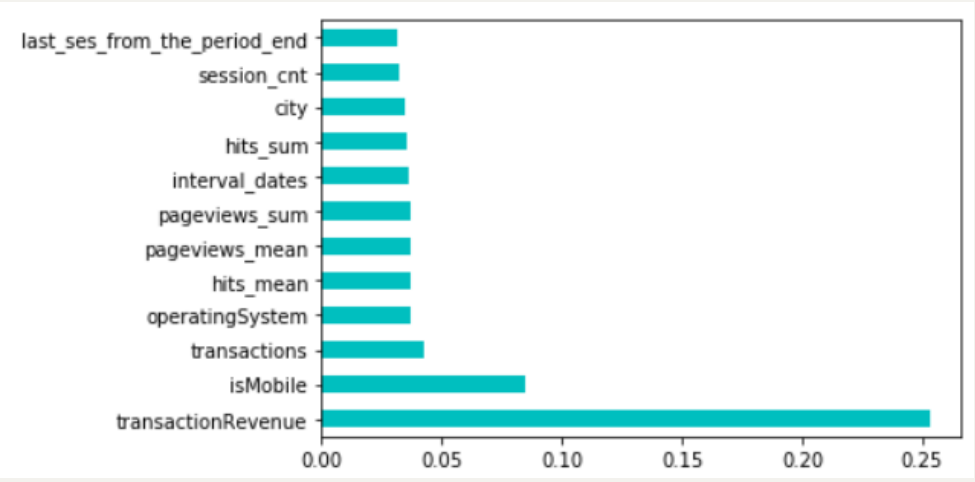


And the result of the updated classification model is below.

```
# Classification Report
print(classification_report(y_valid_return, y_pred_valid_return))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      1.00    311474
           1       0.36      0.02      0.04      1952

    accuracy                           0.99    313426
   macro avg       0.68      0.51      0.52    313426
weighted avg       0.99      0.99      0.99    313426
```

For the regression model, the threshold of feature importance is decided as 0.035 and then take the columns consisting of transactionRevenue, isMobile, transactions, pageviews_mean, pageviews_sum, interval_dates, hits_sum to execute regression model again..



Last but not least, the RMSE of regression model with important features for validation data is 4.1716, and for test dataset submitted in Kaggle is 1.03434.To sum up, for random forest and XGBoost, the performance of classification and regression model after tuned hyperparameters and feature selection are higher than the earlier one.

LightGBM

The LightGBM model allows us to directly use categorical values in regression without having to encode them into numerical ones. Since the label encoder we currently use might have negative impacts like incorrect weight allocation, the LightGBM might bring us a nice prediction outcome!

We firstly use RandomSearch to do parameter tuning. According to the latest LightGBM document, there are several hyperparameters we should choose for different goals. For example, for better accuracy, we should tune max_bin, learning_rate and num_leaves. If we have to run a model with many child leaves, we should do regularization to deal with overfitting, which means we should set bagging_fraction, bagging_frequency, and feature_fraction.

At last, we choose 5 parameters to tune for both classification and regression models.

```
param_grid = {'num_leaves': [40,50,70,90], 'bagging_fraction ' : [0.5,0.7,0.9], 'bagging_freq ': [9, 10, 11],
    'feature_fraction': [0.4, 0.5, 0.6],
    'learning_rate': [0.001, 0.01, 0.1]
}
```

After applying RandomSearch, we got a set of optimal hyper parameters.

```
# For Regression Model
param_grid = {'num_leaves': [40,50,70,90],
              'bagging_fraction ' : [0.5,0.7,0.9],
              'bagging_freq ': [9, 10, 11],
              'feature_fraction': [0.4, 0.5, 0.6],
              'learning_rate': [0.001, 0.01, 0.1]}

estimator = lgb.LGBMRegressor()
lbm_re = RandomizedSearchCV(estimator, param_distributions = param_grid, cv=5, scoring = 'neg_mean_squared_error')
lbm_re.fit(X_train_re, y_train_re)
print(lbm_re.best_params_)
```

```
{'num_leaves': 70, 'learning_rate': 0.01, 'feature_fraction': 0.4, 'bagging_freq ': 10, 'bagging_fraction ': 0.9}
```

```
# For Classification Model
estimator = lgb.LGBMClassifier()

lbm_clf = RandomizedSearchCV(estimator, param_distributions = param_grid, cv=5, scoring = 'neg_log_loss')
lbm_clf.fit(X_train_clf, y_train_clf)
print(lbm_clf.best_params_)
```

```
{'num_leaves': 50, 'learning_rate': 0.1, 'feature_fraction': 0.4, 'bagging_freq ': 11, 'bagging_fraction ': 0.5}
```

Next, we apply the hyperparameters to model training. We firstly train the classification model on training data and then use the validation data to calculate the best iteration number. We do the same thing for the regression model.

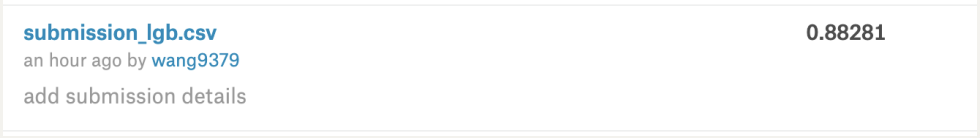
```
# The best iteration number is 61
clf_model = lgb.train(param_lgb2, dtrain_clf, 1200, valid_sets = [dtest_clf],
                      verbose_eval=20, early_stopping_rounds=700, categorical_feature = cat)

# The best iteration number is 202
re_model = lgb.train(param_lgb3, dtrain_re, 1200, valid_sets = [dtest_re],
                      verbose_eval=20, early_stopping_rounds=700, categorical_feature = cat)
```

Finally, with all tuned hyperparameters and best iteration numbers, we train a lightGBM model on the total training data and do prediction. We first train a classification model, the output of which is a binary variable indicating whether a customer is going to return or not. Then we train a regression

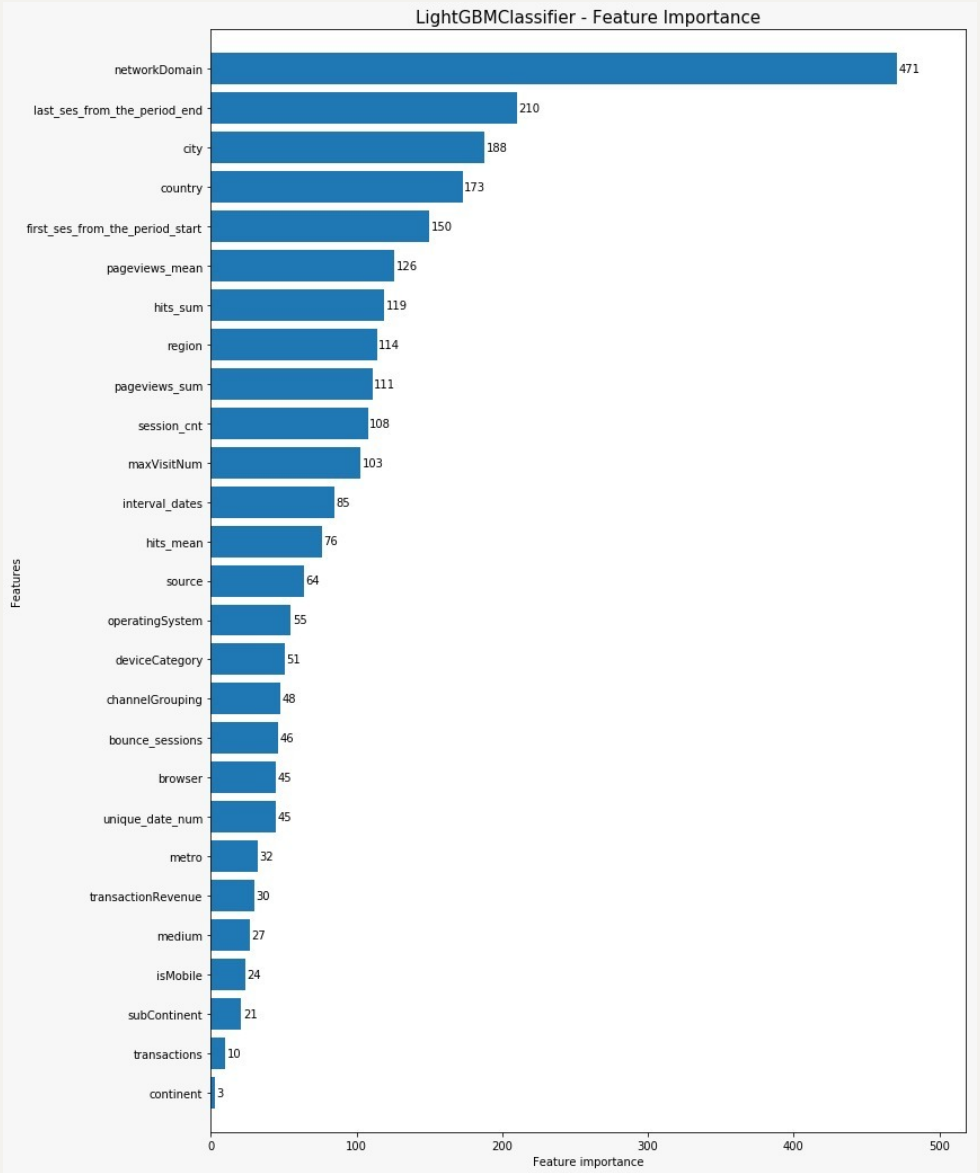
model, the output of which is a numerical variable indicating how much the customer will spend in the future two months. Lastly, we multiply the two output to get the final revenue for each customer in the future two months.

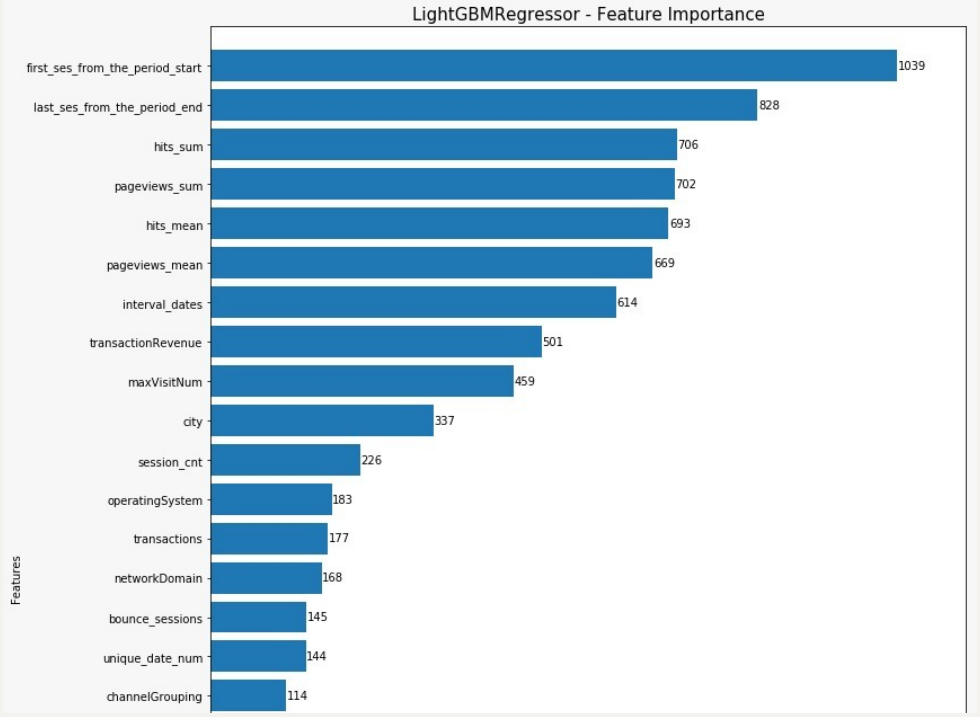
```
pr_lgb_sum = 0
for i in range(10):
    lgb_model1 = lgb.train(param_lgb2, X_train_clf, 65)
    pr_lgb = lgb_model1.predict(X_test)
    lgb_model2 = lgb.train(param_lgb3, X_train_re,  168)
    pr_lgb_ret = lgb_model2.predict(X_test)
    pr_lgb_sum = pr_lgb_sum + pr_lgb * pr_lgb_ret

pr_final2 = (pr_lgb_sum / 10).round(20)
```

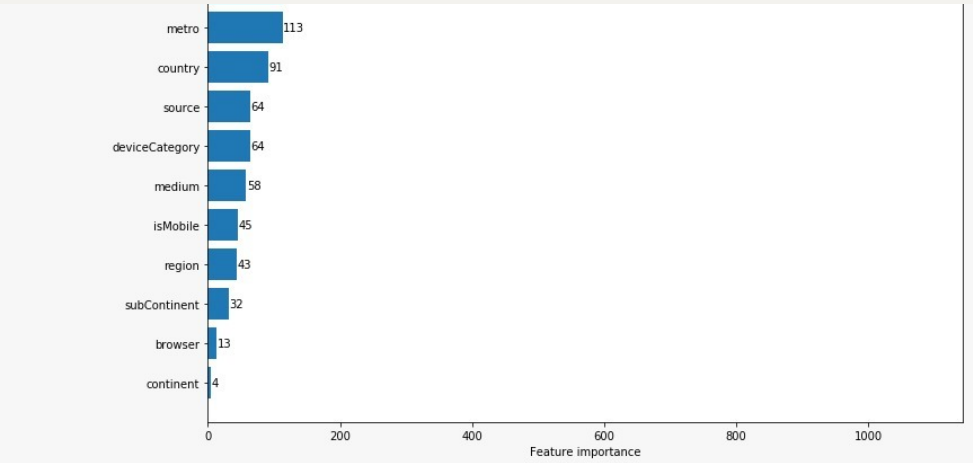
We submitted the output file to Kaggle and got RMSE of **0.88281**.



Next, we do feature selection, selecting 10 important predictors out of 27 variables we currently have, based on descending feature importance.


LightGBMClassifier - Feature Importance
networkDomain 471
last_ses_from_the_period_end 210
city 188
country 173
first_ses_from_the_period_start 150
pageviews_mean 126
hits_sum 119
region 114
pageviews_sum 111
session_cnt 108
maxVisitNum 103
interval_dates 85
hits_mean 76
source 64
operatingSystem 55
deviceCategory 51
channelGrouping 48
bounce_sessions 46
browser 45
unique_date_num 45
metro 32
transactionRevenue 30
medium 27
isMobile 24
subContinent 21
transactions 10
continent 3
Features
0 100 200 300 400 500
Feature importance


LightGBMRegressor - Feature Importance
first_ses_from_the_period_start 1039
last_ses_from_the_period_end 828
hits_sum 706
pageviews_sum 702
hits_mean 693
pageviews_mean 669
interval_dates 614
transactionRevenue 501
maxVisitNum 459
city 337
session_cnt 226
operatingSystem 183
transactions 177
networkDomain 168
bounce_sessions 145
unique_date_num 144
channelGrouping 114
Features



After choosing the top 10 features for each model, we apply the same procedure - tuning hyperparameters, getting the best iteration numbers and training the LightGBM model. The result we got from Kaggle evaluation is RMSE of **0.88283**, which is almost the same as the previous one without any feature selection.

submission_lgb_fs.csv 2 minutes ago by wang9379	0.88283

This is because each variable, even the barely important one, contributes to the final prediction. For further research, we can apply PCA to combine various features instead of simply subsetting them. But now for interpretation convenience, we will use the model with feature selection.

Results & Future Steps

Results

Modes	RMSE on Validation Data with all features	RMSE on Validation Data with important features	RMSE on test Data with all features	RMSE on test Data with important features
Random Forest	3.90184	3.47407	1.10428	1.07444
XGBoost	4.40743	4.13472	1.25359	1.03434
LightGBM	3.86255	3.91213	**0.88188**	**0.88281**

Overall, LightGBM performs better on both test data with all features and with important features as compared to other models. From the above table it can be observed that RMSE on test data with all features for LightGBM is slightly lower than the RMSE on test data with important features i.e. after feature

selection. Although there is a little tradeoff of RMSE when selecting the model with feature selection, it should be better to use feature selection in the final model as it simplifies the model and reduces the overall running cost of the analysis i.e. (data collection and data processing).

Business Value

Our best model LightGBM uncovers hidden patterns and trends in the data. It can be a guiding tool to Google Store for their marketing strategies and future decision making. Our first classification model can help to predict whether one customer will purchase in the next two months or not. Based on this, Google Store can classify their customers into two groups, return group or non-return group. Then, the marketing team can develop separate marketing strategies for different groups. For example, they can provide customized service to their valued customers which can enhance their shopping experience.

For the non-return group, they can send them incentivized surveys to understand the reason why they don't return or send them offers of their loss leaders so that they can attract them to visit their website. Our second regression model can help to precisely predict how much one customer will spend, which will help Google Store to predict their future revenue. Besides, it can be used as a budget control tool that the marketing team can use this model to use their budget wisely.

Implementation Method

We used categorical variables without encoding in the LightGBM model as encoding techniques like one-hot encoding made data sparse and other techniques as mentioned under data preparation had their own shortcomings.

Future steps

The potential issue with the current scenario is the lack of data available to consider the time-series factor. We have only one year of data which have months same as the months of prediction data because of which analysis lacks one of the important factors for predicting customer revenue based on time.

To mitigate this issue, longer period of data will be needed which should have the same periods as that of prediction data to further implement time-series analysis on the data.