IST 718 Project Code

December 14, 2021

1 Load All The Required Packages

[1]:
```
# create spark and sparkcontext objects
from pyspark.sql import SparkSession
import numpy as np

#spark = SparkSession.builder.config('','4g').getOrCreate()
spark = SparkSession.builder.config('spark.driver.memory','4g').getOrCreate()

sc = spark.sparkContext

import pyspark
from pyspark.ml import feature, regression, Pipeline
from pyspark.sql import functions as func, Row
from pyspark import sql

from pyspark.sql.functions import *
from pyspark.sql.types import IntegerType, DoubleType, FloatType

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

2 Load the data

[2]:
```
train_df = spark.read.csv('fraudTrain.csv', header=True, inferSchema=True)
test_df = spark.read.csv('fraudTest.csv', header=True, inferSchema=True)
```

[3]:
```
train_df.printSchema()
```

```
root
 |-- _c0: integer (nullable = true)
 |-- trans_date_trans_time: string (nullable = true)
 |-- cc_num: long (nullable = true)
 |-- merchant: string (nullable = true)
```

```
 |-- category: string (nullable = true)
 |-- amt: double (nullable = true)
 |-- first: string (nullable = true)
 |-- last: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- street: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- zip: integer (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- city_pop: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- dob: string (nullable = true)
 |-- trans_num: string (nullable = true)
 |-- unix_time: integer (nullable = true)
 |-- merch_lat: double (nullable = true)
 |-- merch_long: double (nullable = true)
 |-- is_fraud: integer (nullable = true)
```

[4]:
```
test_df.printSchema()
```

```
root
 |-- _c0: integer (nullable = true)
 |-- trans_date_trans_time: string (nullable = true)
 |-- cc_num: long (nullable = true)
 |-- merchant: string (nullable = true)
 |-- category: string (nullable = true)
 |-- amt: double (nullable = true)
 |-- first: string (nullable = true)
 |-- last: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- street: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- zip: integer (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- city_pop: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- dob: string (nullable = true)
 |-- trans_num: string (nullable = true)
 |-- unix_time: integer (nullable = true)
 |-- merch_lat: double (nullable = true)
 |-- merch_long: double (nullable = true)
 |-- is_fraud: integer (nullable = true)
```

As we can see above, the schema results for both datasets shows us the data type of available features. Since both datasets have same features we can combine them to create a new dataset.

[5]:
```
# Combine train and test data and use cross validation later
combined_df = train_df.union(test_df)
row = combined_df.count()
col = len(combined_df.columns)
print(f'Dimension of the combined Dataframe is: {(row,col)}')
```

```
Dimension of the combined Dataframe is: (1852394, 23)
```

Let's create some new features to unveil the historical behavior transactions done by customers.

- haversine udf function helps to find out the distance between two different places based on the latitude and longitude data. In our case we are finding the distance between merchant's place to the customer's place.

Using create_column udf for following feature generation: - - day_of_week - a particular day of a week of a transaction extracted from trans_date_trans_time feature. - hour_of_transaction - 24-hour details of a transaction extracted from trans_date_trans_time feature. - year - Year of a transaction extracted from trans_date_trans_time feature. - month - Month of a transaction extracted from trans_date_trans_time feature. - month_year - Month & year of a transaction extracted from trans_date_trans_time feature. - trans_date - Transaction date of a transaction extracted from trans_date_trans_time feature. - age - Age of a customer extracted using transaction date and date of birth of a customer.

[6]:
```
# generating column age, day_of_week, hour_of_transaction

from pyspark.sql.functions import *
from pyspark.sql.types import IntegerType, DoubleType


# Function to calculate the distance between two adress
def haversine(lon1, lat1, lon2, lat2):
    lon1, lat1, lon2, lat2 = map(np.radians, [lon1, lat1, lon2, lat2])
    newlon = lon2 - lon1
    newlat = lat2 - lat1
    haver_formula = (
        np.sin(newlat / 2.0) ** 2
        + np.cos(lat1) * np.cos(lat2) * np.sin(newlon / 2.0) ** 2
    )
    dist = 2 * np.arcsin(np.sqrt(haver_formula))
    miles = 3958 * dist
    return float(miles)

# create a udf for implementing python function in pyspark
udf_haversine = udf(haversine, DoubleType())

def create_column(data):
```

```
    # day of week and the transaction hour
    data = data.withColumn('day_of_week', date_format('trans_date_trans_time',␣
 ↪'EEEE')) #1st col added
    data = data.withColumn('hour_of_transaction',␣
 ↪hour('trans_date_trans_time')) #2nd col added

    #month_year
    data = data.withColumn('year', year('trans_date_trans_time'))
    data = data.withColumn('month', month('trans_date_trans_time'))
    data = data.withColumn('month_year', concat_ws('-', data.year ,data.month)).
 ↪drop(*['year', 'month']) #3rd col added

    # trans_date
    data = data.withColumn("trans_date", func.to_date(func.
 ↪col("trans_date_trans_time")))

    #age
    #data = data.
 ↪withColumn("age",round(months_between(current_date(),col("dob"))/lit(12),2))
    data = data.
 ↪withColumn("age",round(months_between(col('trans_date'),col("dob"))/
 ↪lit(12),2))
    data = data.withColumn("age", data["age"].cast(IntegerType()))

    # distance between merchant and client
    # data = data.withColumn("distance", udf_haversine("long", "lat",␣
 ↪"merch_long","merch_lat"))

    return data

combined_df = create_column(combined_df)
train_df = create_column(train_df)
test_df = create_column(test_df)
```

[7]:
```
# finding distance between merchant and customer

udf_haversine = udf(haversine, DoubleType())
combined_df = combined_df.withColumn("distance", udf_haversine("long", "lat",␣
 ↪"merch_long","merch_lat"))
```

[8]:
```
combined_df.printSchema()
```

```
root
 |-- _c0: integer (nullable = true)
 |-- trans_date_trans_time: string (nullable = true)
 |-- cc_num: long (nullable = true)
 |-- merchant: string (nullable = true)
```

```
 |-- category: string (nullable = true)
 |-- amt: double (nullable = true)
 |-- first: string (nullable = true)
 |-- last: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- street: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- zip: integer (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- city_pop: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- dob: string (nullable = true)
 |-- trans_num: string (nullable = true)
 |-- unix_time: integer (nullable = true)
 |-- merch_lat: double (nullable = true)
 |-- merch_long: double (nullable = true)
 |-- is_fraud: integer (nullable = true)
 |-- day_of_week: string (nullable = true)
 |-- hour_of_transaction: integer (nullable = true)
 |-- month_year: string (nullable = false)
 |-- trans_date: date (nullable = true)
 |-- age: integer (nullable = true)
 |-- distance: double (nullable = true)
```

As we can see above, the schema results shows new feature generated.

[9]:
```
# count unique credit cards
from pyspark.sql.functions import countDistinct
combined_df.select(countDistinct('cc_num').alias('CreditCard_Count')).show()
```

```
+----------------+
|CreditCard_Count|
+----------------+
|             999|
+----------------+
```

We have data of 999 credit cards. Credit card fraud detection is based on analysis of a card's spending behavior. It is important to know the past transaction history of a credit card. Thus, we need to use feature engineering to create columns that can study a card's frequency of transaction in past 1 day, 1 week, 1 month and 3 months.

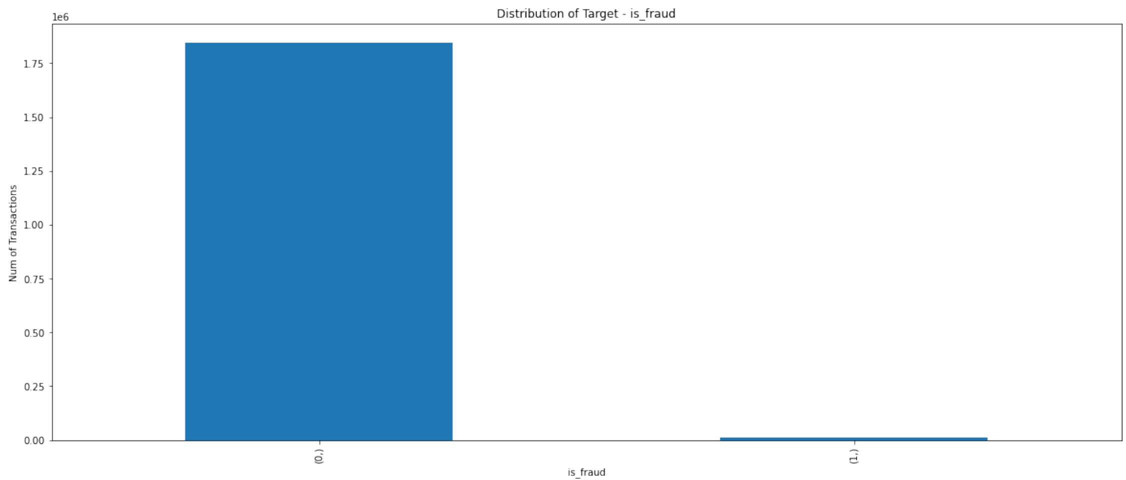
[10]:
```
# Adding dervided columns to understand the credit card usage behaviour Daily,␣
↪Monthly and weekly.
# Creating new_df dataframe to store the original dataframe alongwith new␣
↪features.
```

```
combined_df.createOrReplaceTempView("combined_df")  # creating temp view/table␣
↪to use it for SQL purpose

new_df = \
    spark.sql(
    """SELECT *, mean(amt) OVER (
        PARTITION BY cc_num
        ORDER BY CAST(trans_date AS timestamp)
        RANGE BETWEEN INTERVAL 0 DAYS PRECEDING AND CURRENT ROW
     ) AS rolling_24h_avg_amt,

     mean(amt) OVER (
        PARTITION BY cc_num
        ORDER BY CAST(trans_date AS timestamp)
        RANGE BETWEEN INTERVAL 6 DAYS PRECEDING AND CURRENT ROW
     ) AS rolling_1_week_avg_amt,

     mean(amt) OVER (
        PARTITION BY cc_num
        ORDER BY CAST(trans_date AS timestamp)
        RANGE BETWEEN INTERVAL 29 DAYS PRECEDING AND CURRENT ROW
     ) AS rolling_1month_avg_amt,

     count(_c0) OVER (
    PARTITION BY cc_num, trans_date
    ) AS number_trans_24h,

    count(_c0) OVER (
    PARTITION BY cc_num, day_of_week
    ) AS number_trans_specific_day,

    count(_c0) OVER (
    PARTITION BY cc_num, month_year
    ) AS number_trans_month,

    sum(amt) OVER (
        PARTITION BY cc_num
        ORDER BY CAST(trans_date AS timestamp)
        RANGE BETWEEN INTERVAL 89 DAYS PRECEDING AND CURRENT ROW
     ) AS total_3month_amt

     FROM combined_df""")

new_df = new_df.
↪withColumn('weekly_avg_amt_over_3_months',(col('total_3month_amt')/ (1.
↪0*12))) # deriving weekly_avg_amt_over_3_months from total_3month_amt
new_df = new_df.drop(*['total_3month_amt'])  # Removing unnecessary column
```

2.1 Exploratory Data Analysis

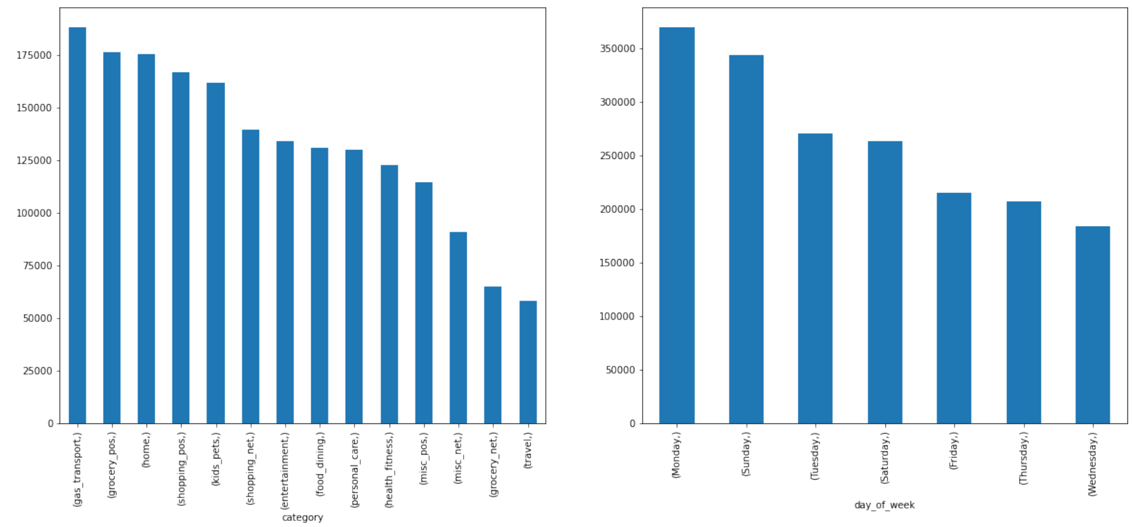
Let's understand the distribution of target variable: -

[55]:
```
# countplot of target variable is_fraud
plt.figure(figsize=(20,8))
new_df.select('is_fraud').toPandas().value_counts().plot.bar()
plt.ylabel('Num of Transactions')
plt.title('Distribution of Target - is_fraud')
plt.show()
```



As we can see from above graph, the target variable is highly imbalanced. We need to tackle this imbalance by choosing the appropriate performance metrics.

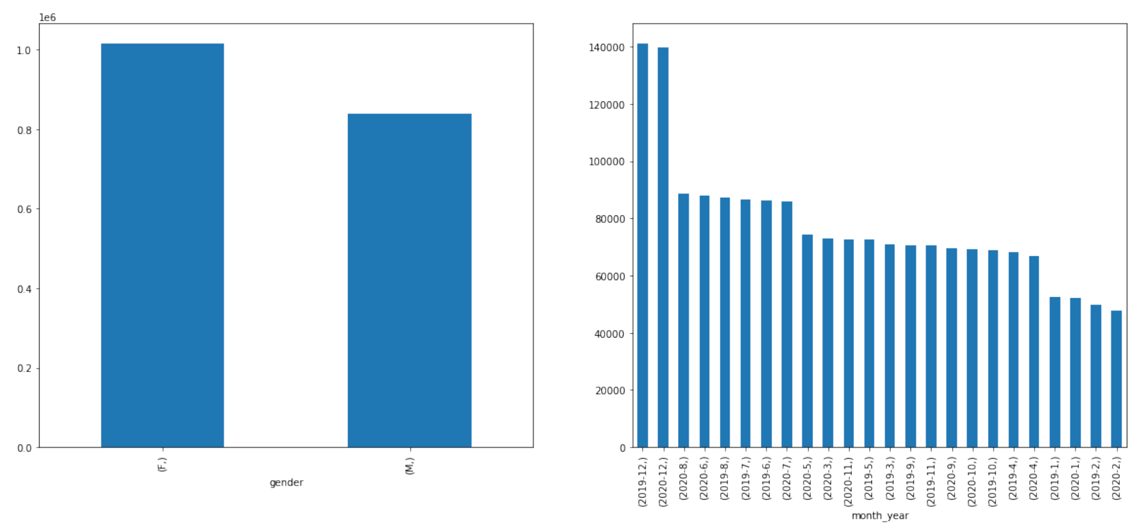
[12]:
```
# countplots for Category and day_of_week features
plt.figure(figsize=(20,8))
plt.subplot(1,2,1)
new_df.select('category').toPandas().value_counts().plot.bar();
plt.subplot(1,2,2)
new_df.select('day_of_week').toPandas().value_counts().plot.bar();
```



As we can see in the above visualizations the no. of transactions happening at gas_transport category are more and for travel category are less.

Also, from the second graph we can visualize that more number of transactions happen on Monday and less on Wednesday.

[13]:
```
# countsplots for gender and month_year features
plt.figure(figsize=(20,8))
plt.subplot(1,2,1)
new_df.select('gender').toPandas().value_counts().plot.bar();
plt.subplot(1,2,2)
new_df.select('month_year').toPandas().value_counts().plot.bar();
```



As we can see from the above countplots, no of transactions done by female are more as compared with males. Also, there are large no. of transactions happening in the month of December for both 2019 and 2020 years.

[14]:
```python
# changing the transdate_trans_time and dob to timestamp

new_df = new_df.withColumn('trans_date_trans_time_new',␣
 ↪to_timestamp('trans_date_trans_time'))
new_df = new_df.drop('trans_date_trans_time')
new_df = new_df.
 ↪withColumnRenamed('trans_date_trans_time_new','trans_date_trans_time')

new_df = new_df.withColumn('dob_new', to_date('dob'))
new_df = new_df.drop('dob')
new_df = new_df.withColumnRenamed('dob_new','dob')
```

[15]:
```python
# verifying the data type and new feature changes in the new_df dataframe
new_df.printSchema()
```

```
root
 |-- _c0: integer (nullable = true)
 |-- cc_num: long (nullable = true)
 |-- merchant: string (nullable = true)
 |-- category: string (nullable = true)
 |-- amt: double (nullable = true)
 |-- first: string (nullable = true)
 |-- last: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- street: string (nullable = true)
 |-- city: string (nullable = true)
 |-- state: string (nullable = true)
 |-- zip: integer (nullable = true)
 |-- lat: double (nullable = true)
 |-- long: double (nullable = true)
 |-- city_pop: integer (nullable = true)
 |-- job: string (nullable = true)
 |-- trans_num: string (nullable = true)
 |-- unix_time: integer (nullable = true)
 |-- merch_lat: double (nullable = true)
 |-- merch_long: double (nullable = true)
 |-- is_fraud: integer (nullable = true)
 |-- day_of_week: string (nullable = true)
 |-- hour_of_transaction: integer (nullable = true)
 |-- month_year: string (nullable = false)
 |-- trans_date: date (nullable = true)
 |-- age: integer (nullable = true)
 |-- distance: double (nullable = true)
 |-- rolling_24h_avg_amt: double (nullable = true)
```

```
 |-- rolling_1_week_avg_amt: double (nullable = true)
 |-- rolling_1month_avg_amt: double (nullable = true)
 |-- number_trans_24h: long (nullable = false)
 |-- number_trans_specific_day: long (nullable = false)
 |-- number_trans_month: long (nullable = false)
 |-- weekly_avg_amt_over_3_months: double (nullable = true)
 |-- trans_date_trans_time: timestamp (nullable = true)
 |-- dob: date (nullable = true)
```

[16]:
```
# verifying the distribution for age feature using pandas dataframe
np.round(new_df.select('age').toPandas().describe())
```

[16]:

	age
count	1852394.0
mean	46.0
std	17.0
min	13.0
25%	32.0
50%	44.0
75%	57.0
max	96.0

[17]:
```
# verifying the distribution for age feature using spark dataframe
new_df.select('age').describe().show()
```

```
+-------+------------------+
|summary|               age|
+-------+------------------+
|  count|           1852394|
|   mean|45.767610994205334|
| stddev| 17.41244464440168|
|    min|                13|
|    max|                96|
+-------+------------------+
```

33-57 age people are 50% of our customers

Minimum age of customer is 13

Maximum age of customer is 96

Modifying the age variable with Categorical distinctions as follows:

[18]:
```
def udf_age_category(age):
    if (age < 25):
        return 'Age Under 25'
    elif (age >= 25 and age < 40):
        return 'Age Between 25 & 40'
```

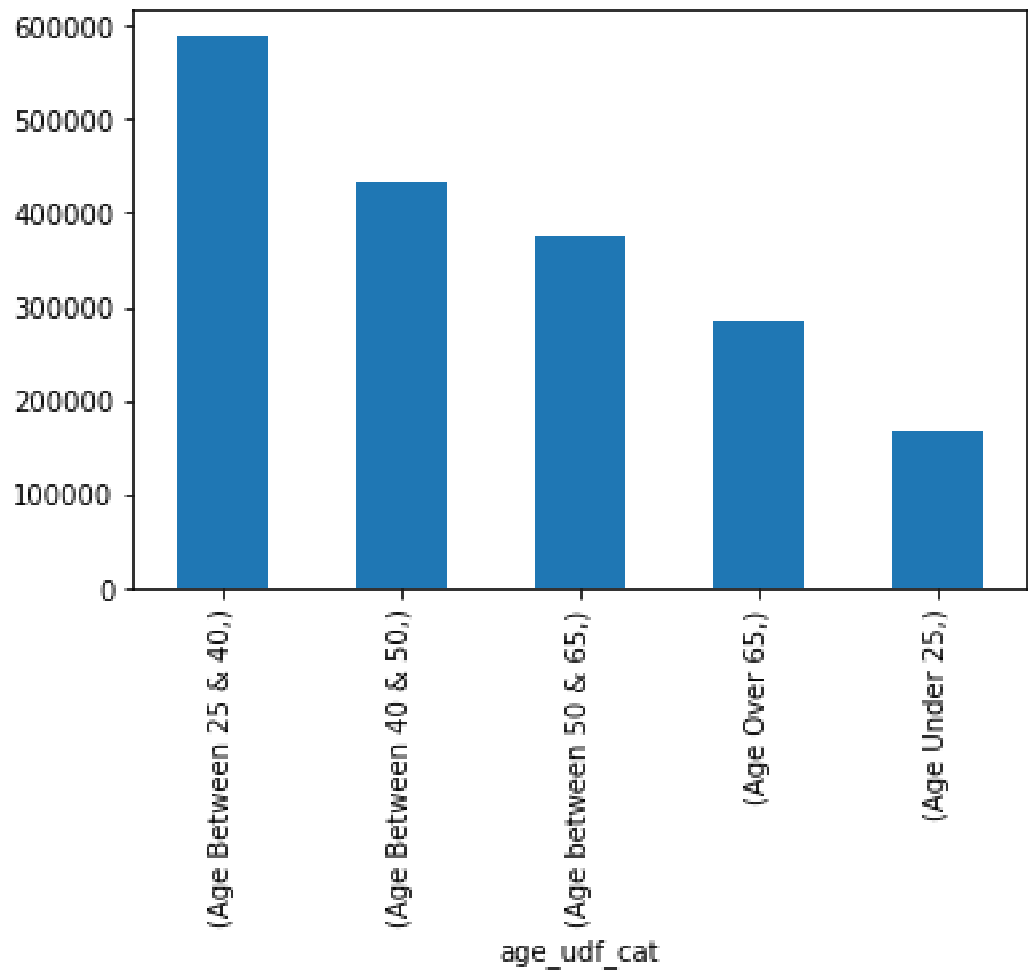
```
    elif (age >= 40 and age < 50):
        return 'Age Between 40 & 50'
    elif (age >=50 and age < 65):
        return 'Age between 50 & 65'
    elif (age >=65):
        return 'Age Over 65'
    else: return 'N/A'

age_udf = udf(udf_age_category)

new_df = new_df.withColumn('age_udf_cat',age_udf('age'))
```

[19]:
```
# Understanding the distinction of transactions done by various age categories.
age_distribution = new_df.select('age_udf_cat').groupBy('age_udf_cat').
↪agg(count(col('age_udf_cat')).alias('Age_Count')).sort('Age_Count',␣
↪ascending = False).show(truncate = False)
age_distribution
```

```
+-------------------+---------+
|age_udf_cat        |Age_Count|
+-------------------+---------+
|Age Between 25 & 40|588955   |
|Age Between 40 & 50|433516   |
|Age between 50 & 65|377176   |
|Age Over 65        |284802   |
|Age Under 25       |167945   |
+-------------------+---------+
```

[20]:
```
# Visualizing the distinction of transactions done by various age categories.
new_df.select('age_udf_cat').toPandas().value_counts().plot.bar()
```

[20]: <AxesSubplot:xlabel='age_udf_cat'>



As we can see from above visualizations, large no. of transactions are done by customers in the age group of 25 to 40.

[21]:
```
# Descriptive statistics for the overall amount of transactions
np.round(((new_df.select('amt')).toPandas().describe(percentiles = [0.25,0.5,0.
↪75,0.95,0.999])),3)
```

[21]:

	amt
count	1852394.000
mean	70.064
std	159.254
min	1.000
25%	9.640
50%	47.450
75%	83.100
95%	195.340
99.9%	1517.241

```
max      28948.900
```

[22]:
```python
# Descriptive statistics for the legitimate amount of transactions
np.round(((new_df.select('amt').filter(col('is_fraud') == 0)).toPandas().
 ↪describe(percentiles = [0.25,0.5,0.75,0.95,0.999])),3)
```

[22]:

	amt
count	1842743.000
mean	67.651
std	153.548
min	1.000
25%	9.610
50%	47.240
75%	82.560
95%	189.590
99.9%	1519.623
max	28948.900

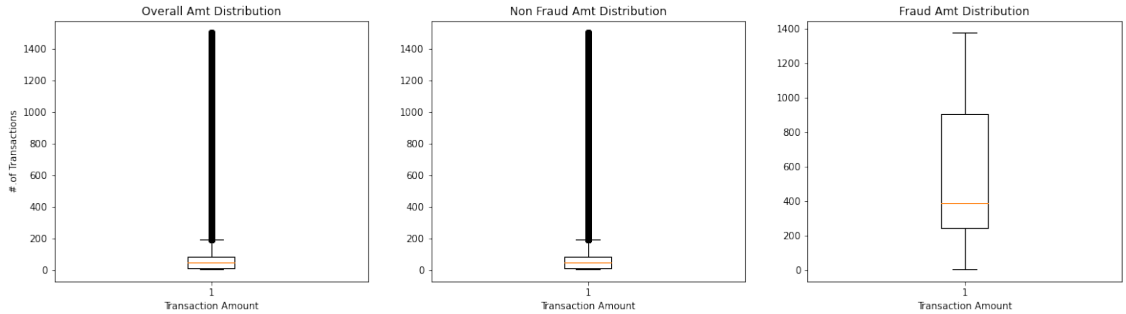
[23]:
```python
# Descriptive statistics for the fraudulent amount of transactions
np.round(((new_df.select('amt').filter(col('is_fraud') == 1)).toPandas().
 ↪describe(percentiles = [0.25,0.5,0.75,0.95,0.999])),3)
```

[23]:

	amt
count	9651.000
mean	530.661
std	391.029
min	1.060
25%	240.075
50%	390.000
75%	902.365
95%	1084.090
99.9%	1293.127
max	1376.040

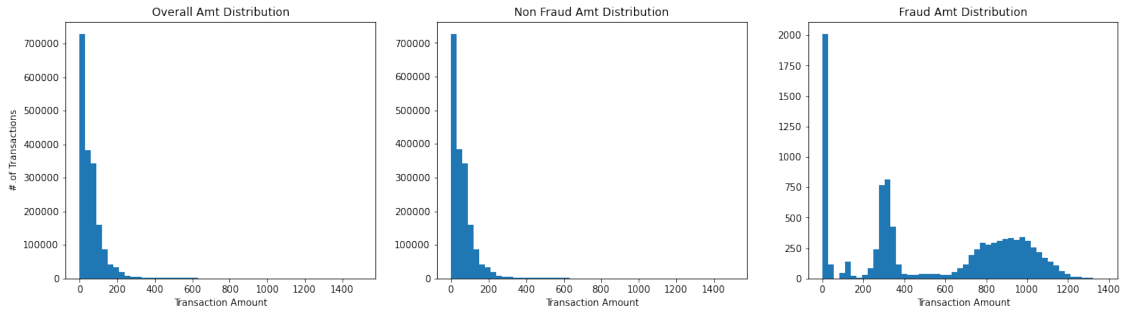
[24]:
```python
# boxplot distribution for overall amount, legitimate amount, and fraudulent␣
 ↪amount
fig, ax = plt.subplots(1,3,figsize=(20,5))
ax[0].boxplot((new_df.select('amt').filter(col('amt') <= 1500.0)).toPandas())
ax[1].boxplot((new_df.select('amt').filter((col('amt') <= 1500.0) &␣
 ↪(col('is_fraud') == 0))).toPandas())
ax[2].boxplot((new_df.select('amt').filter((col('amt') <= 1500.0) &␣
 ↪(col('is_fraud') == 1))).toPandas())

ax[0].set_title('Overall Amt Distribution')
ax[1].set_title('Non Fraud Amt Distribution')
ax[2].set_title('Fraud Amt Distribution')
```

```
ax[0].set_xlabel('Transaction Amount')
ax[0].set_ylabel('#.of Transactions')

ax[1].set_xlabel('Transaction Amount')
ax[2].set_xlabel('Transaction Amount')
plt.show()
```



[25]:

```
# histogram distribution for overall amount, legitimate amount, and fraudulent␣
↪amount
fig, ax = plt.subplots(1,3,figsize=(20,5))
ax[0].hist((new_df.select('amt').filter(col('amt') <= 1500.0)).toPandas(),bins␣
↪= 50)
ax[1].hist((new_df.select('amt').filter((col('amt') <= 1500.0) &␣
↪(col('is_fraud') == 0))).toPandas(),bins = 50)
ax[2].hist((new_df.select('amt').filter((col('amt') <= 1500.0) &␣
↪(col('is_fraud') == 1))).toPandas(),bins = 50)

ax[0].set_title('Overall Amt Distribution')
ax[1].set_title('Non Fraud Amt Distribution')
ax[2].set_title('Fraud Amt Distribution')

ax[0].set_xlabel('Transaction Amount')
ax[0].set_ylabel('#.of Transactions')

ax[1].set_xlabel('Transaction Amount')
ax[2].set_xlabel('Transaction Amount')
plt.show()
```



As we can see in the above distribution and visualizations around 99% of total records are approximatly below 1500 amt. Also the mean observed for legitimate and fraudulent transactions is diferent. In other words, the mean amount for fraudulent transactions is approx. USD 530 and mean amount for legitimate transactions is approx. USD 67. This tells us that, for a general customer the credit card transaction could be fraudulent if there is sudden spike, such as USD 530, in his transaction amount because usually the transaction amount should be around USD 67.

[26]:
```
(new_df.select('hour_of_transaction').filter(col('is_fraud') == 1).
↪groupBy(col('hour_of_transaction'))\
.count()).orderBy('count', ascending = False).show()
```

```
+-------------------+-----+
|hour_of_transaction|count|
+-------------------+-----+
|                 22| 2481|
|                 23| 2442|
|                  1|  827|
|                  0|  823|
|                  3|  804|
|                  2|  792|
|                 18|  111|
|                 19|  105|
|                 21|  101|
|                 14|  100|
|                 15|  100|
|                 20|   98|
|                 16|   97|
|                 13|   94|
|                 17|   94|
|                 12|   84|
|                  5|   80|
|                  7|   72|
|                  9|   61|
|                  4|   61|
+-------------------+-----+
only showing top 20 rows
```

As we can see in the above analysis, most frauds are done at odd hours, by which we can speculate that scammers can find more customers searching or buying something using credit cards.

[27]:
```
(new_df.select('day_of_week').filter(col('is_fraud') == 1).
 ↪groupBy(col('day_of_week')).count().orderBy('count', ascending = False).
 ↪show()
```

```
+-----------+-----+
|day_of_week|count|
+-----------+-----+
|     Sunday| 1590|
|   Saturday| 1493|
|     Monday| 1484|
|     Friday| 1376|
|   Thursday| 1317|
|    Tuesday| 1266|
|  Wednesday| 1125|
+-----------+-----+
```

In a similar fashion, we can see in the above analysis, most frauds are done at weekends or at start of the week, by which we can speculate that scammers can find more customers searching or buying something using credit cards.

2.1.1 Time Series plots to understand trends

[28]:
```
# used pandas to get visualizations and tables based on monthly transactions␣
 ↪and fraud transactions.
df = new_df.select(to_date('month_year').alias('month_year'),col('trans_num').
 ↪alias('number_of_transactions'),
                                col('cc_num').
 ↪alias('number_of_customers'),col('is_fraud'),col('gender'),
                                   col('category')).toPandas()
```

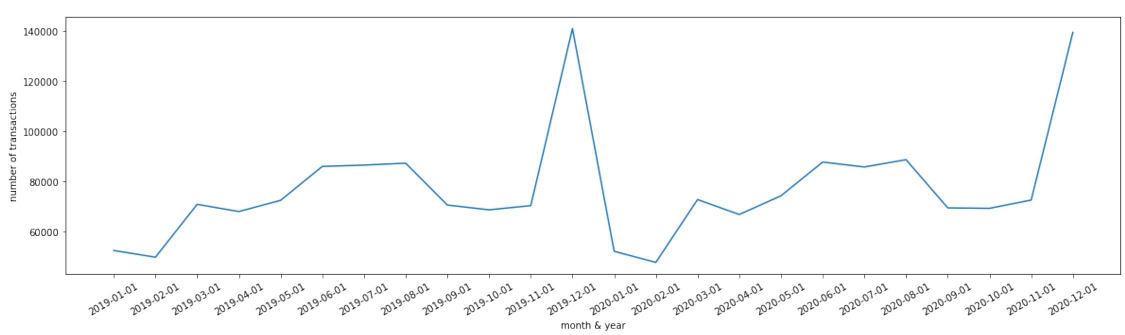
[29]:
```
df_ts_month_trans = df.
 ↪groupby(df['month_year'])[['number_of_transactions','number_of_customers']].
 ↪nunique().reset_index()
#df_ts_month_trans = df_ts_month_trans.sort_values(by = ['month_year'])
df_ts_month_trans
```

[29]:

	month_year	number_of_transactions	number_of_customers
0	2019-01-01	52525	913
1	2019-02-01	49866	918
2	2019-03-01	70939	916
3	2019-04-01	68078	913

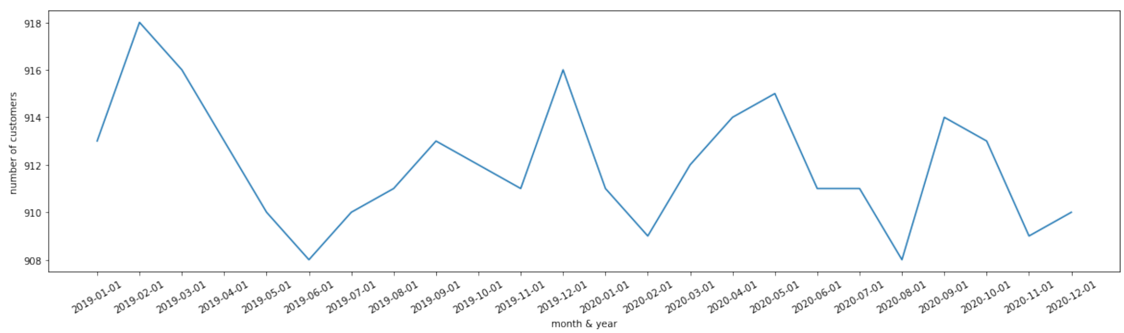
4	2019-05-01	72532	910
5	2019-06-01	86064	908
6	2019-07-01	86596	910
7	2019-08-01	87359	911
8	2019-09-01	70652	913
9	2019-10-01	68758	912
10	2019-11-01	70421	911
11	2019-12-01	141060	916
12	2020-01-01	52202	911
13	2020-02-01	47791	909
14	2020-03-01	72850	912
15	2020-04-01	66892	914
16	2020-05-01	74343	915
17	2020-06-01	87805	911
18	2020-07-01	85848	911
19	2020-08-01	88759	908
20	2020-09-01	69533	914
21	2020-10-01	69348	913
22	2020-11-01	72635	909
23	2020-12-01	139538	910

[58]:
```
# Time series plot to understand number of transactions done per year per month.

x = np.arange(0,len(df_ts_month_trans),1)

fig, ax = plt.subplots(1,1,figsize=(20,5))
ax.plot(x,df_ts_month_trans['number_of_transactions'])
ax.set_xticks(x)
ax.set_xticklabels(df_ts_month_trans['month_year'])

ax.set_xlabel('month & year')
ax.set_ylabel('number of transactions')
plt.xticks(rotation = 30)
plt.show()
```



As mentioned earlier, we can see a spike in the number of transactions for December 2019 and December 2020.

[57]:
```
# Time series plot to understand transactions done by number of customers per␣
↪year per month.

x = np.arange(0,len(df_ts_month_trans),1)

fig, ax = plt.subplots(1,1,figsize=(20,5))
ax.plot(x,df_ts_month_trans['number_of_customers'])
ax.set_xticks(x)
ax.set_xticklabels(df_ts_month_trans['month_year'])

ax.set_xlabel('month & year')
ax.set_ylabel('number of customers')
plt.xticks(rotation = 30)
plt.show()
```



Number of customers are highest for different quarters over the year. Year 2019 has different amount of customers than year 2020.

[32]:
```
# Analyzing the trend of customers and transactions just for fraudulent data..

df_fraud_transactions = df[df['is_fraud']==1]

df_ts_fraud_month_trans = df_fraud_transactions.
↪groupby(df_fraud_transactions['month_year'])[['number_of_transactions','number_of_customers
↪nunique().reset_index()
df_ts_fraud_month_trans.columns =␣
↪['month_year','num_of_fraud_transactions','fraud_customers']
df_ts_fraud_month_trans
```

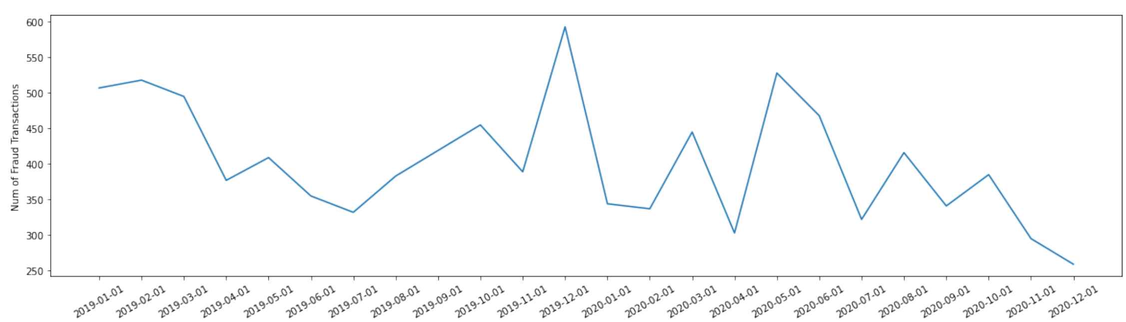
[32]:

	month_year	num_of_fraud_transactions	fraud_customers
0	2019-01-01	506	50
1	2019-02-01	517	53

2	2019-03-01	494	49
3	2019-04-01	376	41
4	2019-05-01	408	42
5	2019-06-01	354	35
6	2019-07-01	331	36
7	2019-08-01	382	39
8	2019-09-01	418	44
9	2019-10-01	454	50
10	2019-11-01	388	41
11	2019-12-01	592	62
12	2020-01-01	343	40
13	2020-02-01	336	35
14	2020-03-01	444	45
15	2020-04-01	302	36
16	2020-05-01	527	54
17	2020-06-01	467	47
18	2020-07-01	321	35
19	2020-08-01	415	41
20	2020-09-01	340	35
21	2020-10-01	384	39
22	2020-11-01	294	31
23	2020-12-01	258	26

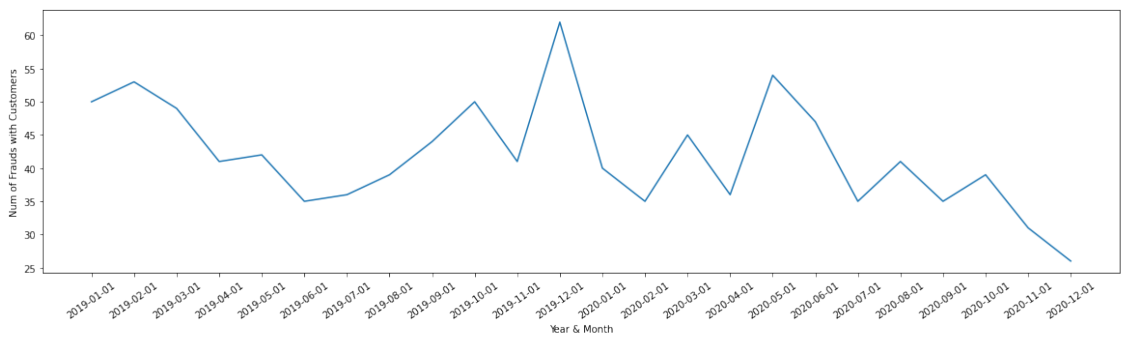
[56]:

```
# Time series plot to understand number of fraudulent transactions done per␣
↪year per month.

x = np.arange(0,len(df_ts_fraud_month_trans),1)

fig, ax = plt.subplots(1,1,figsize=(20,5))
ax.plot(x,df_ts_fraud_month_trans['num_of_fraud_transactions'])
ax.set_xticks(x)
ax.set_xticklabels(df_ts_fraud_month_trans['month_year'])
plt.xticks(rotation = 30)

ax.set_xlabel('Year & Month')
ax.set_ylabel('Num of Fraud Transactions')
plt.show()
```



Number of fraudulent transactions sees some spikes in the month of January 2019, December 2019 and May 2020.

[34]:
```
# Time series plot to understand number of fraudulent transactions done per␣
↪year per month.

x = np.arange(0,len(df_ts_fraud_month_trans),1)

fig, ax = plt.subplots(1,1,figsize=(20,5))
ax.plot(x,df_ts_fraud_month_trans['fraud_customers'])
ax.set_xticks(x)
ax.set_xticklabels(df_ts_fraud_month_trans['month_year'])

ax.set_xlabel('Year & Month')
ax.set_ylabel('Num of Frauds with Customers')
plt.xticks(rotation = 35) # Rotates X-Axis Ticks by 35-degrees
plt.show()
```



The time series trend of no. of frauds happening with customers is following the time series trend of no. of transactions done in two years.

[35]:
```
# Gender wise distribution for Fraudulent and legimiate transactions
df_gender = df[['gender','number_of_transactions']].groupby(['gender']).count().
↪reset_index()

df_gender.columns = ['Gender','gender_count']

df_gender['percent'] = (df_gender['gender_count']/df_gender['gender_count'].
↪sum())*100

df_fraud_gender = df[['gender','is_fraud','number_of_transactions']].
↪groupby(['gender','is_fraud']).count().reset_index()
df_fraud_gender.columns = ['Gender','is_fraud','count']
```

```
df_fraud_gender = df_fraud_gender.
↪merge(df_gender[['Gender','gender_count']],how='inner',\
                                  left_on='Gender',right_on='Gender')


df_fraud_gender['percent_grp'] = (df_fraud_gender['count']/
↪df_fraud_gender['gender_count'])*100


df_fraud_gender
```

[35]:

	Gender	is_fraud	count	gender_count	percent_grp
0	F	0	1009850	1014749	99.517221
1	F	1	4899	1014749	0.482779
2	M	0	832893	837645	99.432695
3	M	1	4752	837645	0.567305

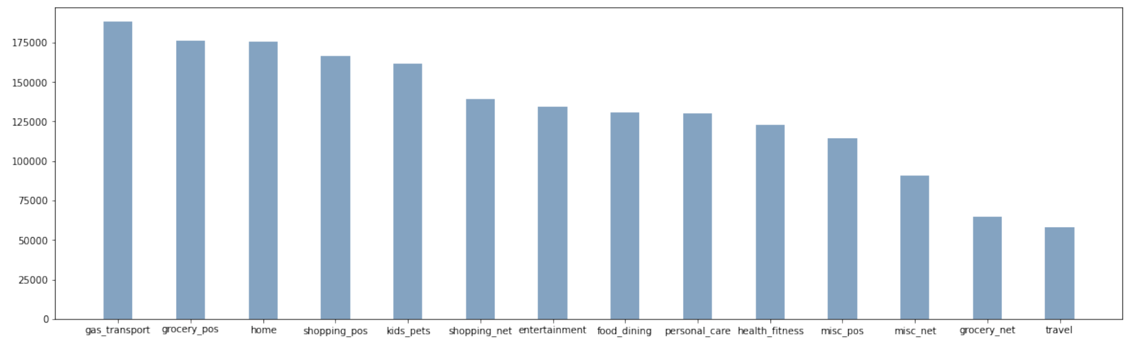
[36]:

```
# Categoriwise distribution for overall transactions
df_category = df[['category','number_of_transactions']].groupby(['category']).
↪count().reset_index()
df_category.columns = ['Category','category_count']

df_category['percent'] = (df_category['category_count']/
↪df_category['category_count'].sum())*100

df_category = (df_category.sort_values(by = ['percent'], ascending=False).
↪reset_index()).drop('index',axis = 1)
df_category
```

[36]:

	Category	category_count	percent
0	gas_transport	188029	10.150594
1	grocery_pos	176191	9.511529
2	home	175460	9.472067
3	shopping_pos	166463	8.986371
4	kids_pets	161727	8.730702
5	shopping_net	139322	7.521186
6	entertainment	134118	7.240252
7	food_dining	130729	7.057300
8	personal_care	130085	7.022534
9	health_fitness	122553	6.615925
10	misc_pos	114229	6.166561
11	misc_net	90654	4.893883
12	grocery_net	64878	3.502387
13	travel	57956	3.128708

[37]:
```
# Visualizing categoriwise distribution for overall transactions
fig = plt.figure(figsize = (20, 6))


plt.bar(df_category['Category'], df_category['category_count'], color=(0.2, 0.
↪4, 0.6, 0.6),
        width = 0.4)

plt.show()
```



As we can see the number of transactions are more for categories such as gas_transport, grocery point of sale, home related transactions, shopping point of sale,… etc.

Let's understand the distribution of categories over fraudulent transactions as follows: -

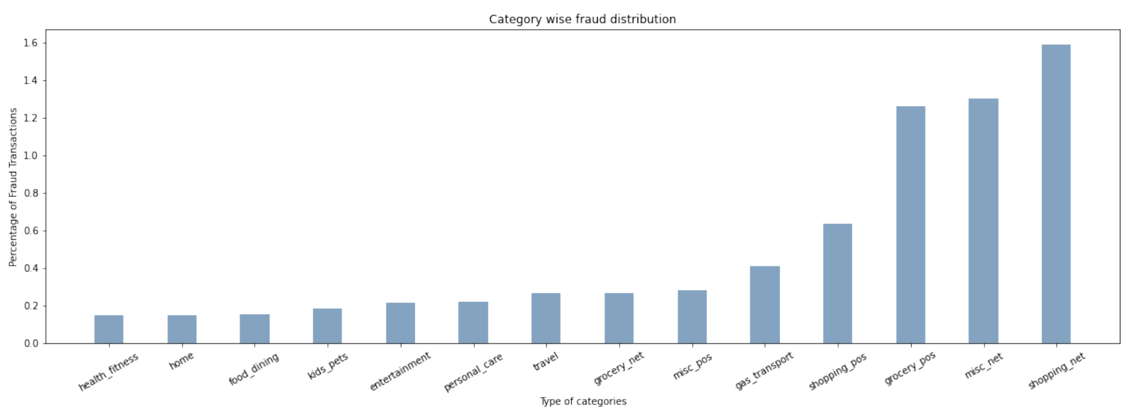
[38]:
```
# Categoriwise distribution for fraudulent transactions
df_fraud_category = df[['category','is_fraud','number_of_transactions']].
↪groupby(['category','is_fraud']).count().reset_index()
df_fraud_category.columns = ['Category','is_fraud','count']

df_fraud_category = df_fraud_category.
↪merge(df_category[['Category','category_count','percent']],how='inner',\
                                    left_on='Category',right_on='Category')


df_fraud_category['percent_grp'] = (df_fraud_category['count']/
↪df_fraud_category['category_count'])*100

df_fraud=df_fraud_category[df_fraud_category['is_fraud'] == 1].sort_values(by =␣
↪['percent_grp'])
df_fraud
```

[38]:

	Category	is_fraud	count	category_count	percent	percent_grp
11	health_fitness	1	185	122553	6.615925	0.150955
13	home	1	265	175460	9.472067	0.151032

3	food_dining	1	205	130729	7.057300	0.156813
15	kids_pets	1	304	161727	8.730702	0.187971
1	entertainment	1	292	134118	7.240252	0.217719
21	personal_care	1	290	130085	7.022534	0.222931
27	travel	1	156	57956	3.128708	0.269170
7	grocery_net	1	175	64878	3.502387	0.269737
19	misc_pos	1	322	114229	6.166561	0.281890
5	gas_transport	1	772	188029	10.150594	0.410575
25	shopping_pos	1	1056	166463	8.986371	0.634375
9	grocery_pos	1	2228	176191	9.511529	1.264537
17	misc_net	1	1182	90654	4.893883	1.303859
23	shopping_net	1	2219	139322	7.521186	1.592713

[39]:
```
# Visualizing categoriwise distribution for fraudulent transactions
fig = plt.figure(figsize = (20, 6))


plt.bar(df_fraud['Category'] , df_fraud['percent_grp'], color=(0.2, 0.4, 0.6, 0.
↪6),
        width = 0.4)
plt.xticks(rotation = 30)
plt.xlabel("Type of categories")
plt.ylabel('Percentage of Fraud Transactions')
plt.title('Category wise fraud distribution')
plt.show()
```



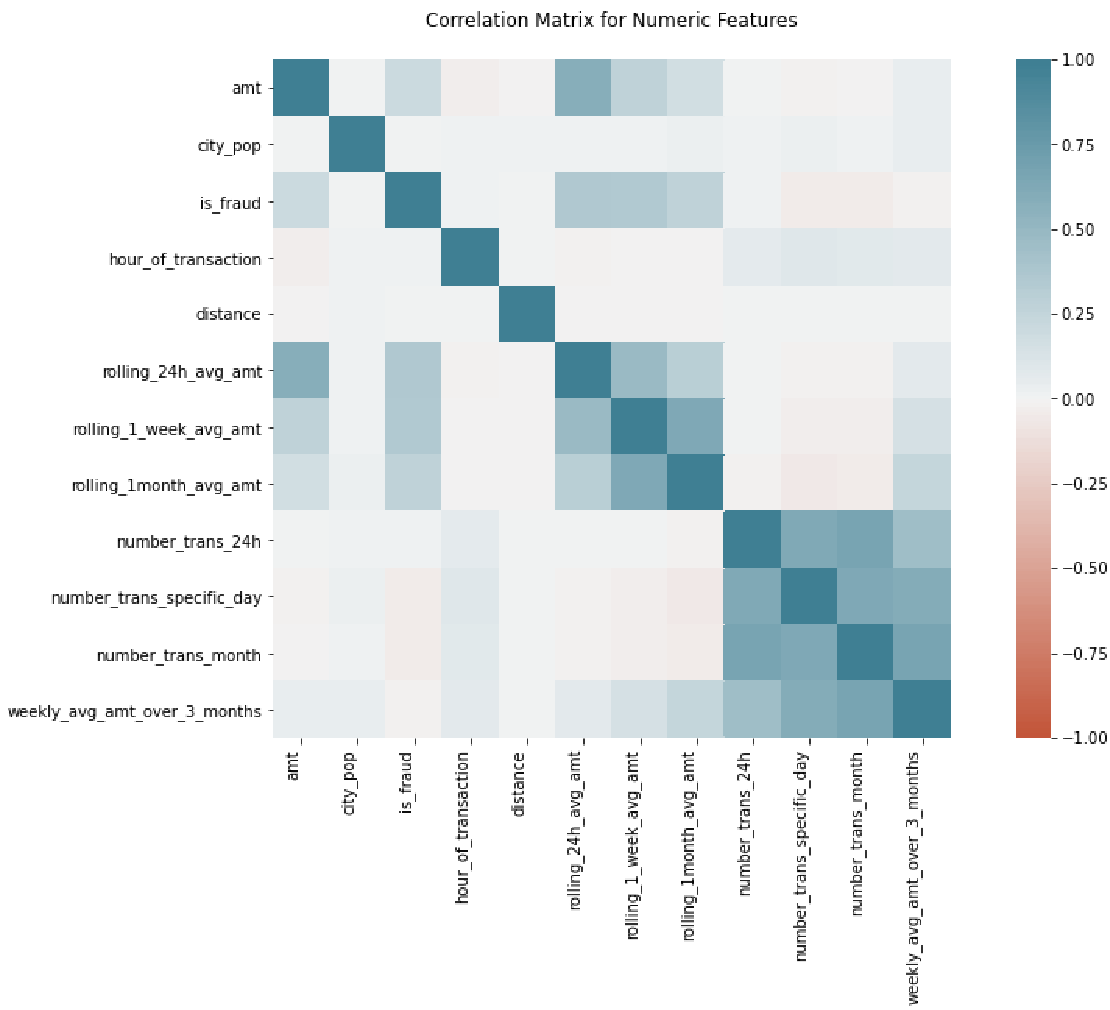
As we can see, from the above visualizations that customers are doing credit card transactions which turned out to be fraud are more when the transactions are done over Internet and for shopping.

3 Correlation Matrix

[40]:
```
cols = ['_c0', 'age', 'trans_date_trans_time', 'cc_num', 'merchant', 'first',␣
↪'last', 'street', 'city', \
       'zip', 'job', 'dob', 'state','unix_time', 'trans_num','lat',␣
↪'long','month_year', 'trans_date', 'merch_lat', 'merch_long']
preprocessed_data = new_df.drop(*cols)
```

[41]:
```
# checking correlation between numerical features

from pyspark.ml.stat import Correlation
from pyspark.ml.feature import VectorAssembler

# creating a dataframe of only numerical features.
numeric_features = [t[0] for t in preprocessed_data.dtypes if t[1] != 'string']
numeric_features_df = preprocessed_data.select(numeric_features)

# convert to vector column first
vector_col = "corr_features"
assembler = VectorAssembler(inputCols=numeric_features_df.columns,␣
↪outputCol=vector_col)
df_vector = assembler.transform(numeric_features_df).select(vector_col)

# Generating Correlation Matrix
matrix = Correlation.corr(df_vector, vector_col).collect()[0][0]
corrmatrix = matrix.toArray().tolist()
```

[42]:
```
import seaborn as sns
plt.figure(figsize = (15,8))
ax = sns.heatmap(
    corrmatrix,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(20, 220, n=200),
    square=True
)
ax.set_title("Correlation Matrix for Numeric Features\n")
ax.set_xticklabels(
    numeric_features_df.columns,
    rotation=90,
    horizontalalignment='right'
);
ax.set_yticklabels(numeric_features_df.columns,
    rotation=0,
    horizontalalignment='right'
);
plt.show()
```



From the above correlation heat-map we can see high positive correlation between rolling average amount spent over 24 hours and rolling average amount spent over a month. Similarly, number of transactions done in month are highly positively correlated with number of transactions done in a day or week.

4 One-hot encoding and VectorAssembler

[43]:
```
# Count of fraud transactions and non-fraud transactions
preprocessed_data.groupBy('is_fraud').count().show()
```

```
+--------+-------+
|is_fraud|  count|
+--------+-------+
|       1|   9651|
|       0|1842743|
```

```
+--------+-------+
```

[44]:
```
# make count plot for target variable
```

Tried different approach for using String Indexer, One Hot encoder and vector assembler as follows:

[45]:
```
from pyspark.ml.feature import OneHotEncoder, StringIndexer, VectorAssembler

def transformColumnsToNumeric(df, inputCol):

    #apply StringIndexer to inputCol
    inputCol_indexer = StringIndexer(inputCol = inputCol, outputCol = inputCol␣
 ↪+ "-index").fit(df)
    df = inputCol_indexer.transform(df)

    onehotencoder_vector = OneHotEncoder(inputCol = inputCol + "-index",␣
 ↪outputCol = inputCol + "-vector")
    df = onehotencoder_vector.fit(df).transform(df)

    return df

    pass
```

[46]:
```
# Indexer and one hot encoding for categorical columns
df = transformColumnsToNumeric(preprocessed_data, "category")
df = transformColumnsToNumeric(df, "gender")
df = transformColumnsToNumeric(df, "age_udf_cat")
df = transformColumnsToNumeric(df, "day_of_week")
```

[47]:
```
# Using Vector Assemblar to accumulate features to be used for machine learning␣
 ↪models
from pyspark.ml.feature import VectorAssembler

cols = [
 'amt',
 'city_pop',
 'hour_of_transaction',
 'distance',
 'rolling_24h_avg_amt',
 'rolling_1_week_avg_amt',
 'rolling_1month_avg_amt',
 'number_trans_24h',
 'number_trans_specific_day',
 'number_trans_month',
 'weekly_avg_amt_over_3_months',
 'category-vector',
```

```
 'gender-vector',
 'age_udf_cat-vector',
 'day_of_week-vector']

vectorAssembler = VectorAssembler().setInputCols(cols).
 ↪setOutputCol('finalfeatures')
df = vectorAssembler.transform(df)
```

[48]:
```
# spliting the data into train, validation and test set
train, validate, test = df.randomSplit([0.6, 0.3, 0.1])
```

[49]:
```
# checking the target variable distribution for fraud and legitimate records
train.groupBy('is_fraud').count().show()
```

```
+--------+-------+
|is_fraud|  count|
+--------+-------+
|       1|   5846|
|       0|1105515|
+--------+-------+
```

[50]:
```
# checking the target variable distribution for fraud and legitimate records
validate.groupBy('is_fraud').count().show()
```

```
+--------+------+
|is_fraud| count|
+--------+------+
|       1|  2884|
|       0|553337|
+--------+------+
```

[51]:
```
# checking the target variable distribution for fraud and legitimate records
test.groupBy('is_fraud').count().show()
```

```
+--------+------+
|is_fraud| count|
+--------+------+
|       1|   921|
|       0|183891|
+--------+------+
```

5 Applying Machine Learning Models

Let's understand some important performance parameters: -

- **False Positive Rate (FPR)**: $\frac{FalsePositives(FP)}{FalsePositives(FP)+TrueNegatives(TN)}$
- **True Positive Rate (TPR) or Recall**: $\frac{TruePositives(TP)}{TruePositives(TP)+FalseNegatives(FN)}$
- **Precision**: $\frac{TruePositives(TP)}{TruePositives(TP)+FalsePositives(FP)}$
- **F1**: $\frac{2*Precision*Recall}{Precision+Recall}$

Let's understand how to tackle the High class imbalance: -

In our case, we have only 0.52% of the transactions as fraud. This implies we will have a large number of true negatives.

The ROC curve is a graph of True Positive rate vs. False positive rate. Whereas, Precision Recall curve is a graph of Precision vs. recall for different threshold of proabability.

Thus, solving this classification problem, due to class imbalance, the area under ROC curve will always closer to one because the FPR will always be very small given the high number of True Negatives. In this case, we should focus on choosing the Area under Precision-Recall Curve as our performance metric and creating the best model to push the area under Precision-Recall curve closer to one. At the end, we will find a threshold probability which maximizes F1 score and above which transactions will be classified as fraud.

Using handyspark package to draw curves

[50]:
```
!pip install handyspark
```

```
WARNING: Value for scheme.headers does not match. Please report this to
<https://github.com/pypa/pip/issues/9617>
distutils: /opt/conda/include/python3.8/UNKNOWN
sysconfig: /opt/conda/include/python3.8
WARNING: Additional context:
user = False
home = None
root = None
prefix = None
Collecting handyspark
  Downloading handyspark-0.2.2a1-py2.py3-none-any.whl (39 kB)
Requirement already satisfied: findspark in /opt/conda/lib/python3.8/site-
packages (from handyspark) (1.4.2)
Requirement already satisfied: numpy in /opt/conda/lib/python3.8/site-packages
```

```
(from handyspark) (1.19.5)
Collecting pyarrow
  Downloading
pyarrow-6.0.1-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (25.6 MB)
     |                 | 25.6 MB 9.0 MB/s eta 0:00:01
Requirement already satisfied: pandas in /opt/conda/lib/python3.8/site-
packages (from handyspark) (1.2.4)
Requirement already satisfied: seaborn in /opt/conda/lib/python3.8/site-packages
(from handyspark) (0.11.1)
Requirement already satisfied: scikit-learn in /opt/conda/lib/python3.8/site-
packages (from handyspark) (0.24.1)
Requirement already satisfied: pyspark in /opt/conda/lib/python3.8/site-packages
(from handyspark) (3.1.2)
Requirement already satisfied: scipy in /opt/conda/lib/python3.8/site-packages
(from handyspark) (1.6.2)
Requirement already satisfied: matplotlib in /opt/conda/lib/python3.8/site-
packages (from handyspark) (3.3.4)
Requirement already satisfied: python-dateutil>=2.1 in
/opt/conda/lib/python3.8/site-packages (from matplotlib->handyspark) (2.8.1)
Requirement already satisfied: kiwisolver>=1.0.1 in
/opt/conda/lib/python3.8/site-packages (from matplotlib->handyspark) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.8/site-
packages (from matplotlib->handyspark) (0.10.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.3 in
/opt/conda/lib/python3.8/site-packages (from matplotlib->handyspark) (2.4.7)
Requirement already satisfied: pillow>=6.2.0 in /opt/conda/lib/python3.8/site-
packages (from matplotlib->handyspark) (8.1.2)
Requirement already satisfied: six in /opt/conda/lib/python3.8/site-packages
(from cycler>=0.10->matplotlib->handyspark) (1.15.0)
Requirement already satisfied: pytz>=2017.3 in /opt/conda/lib/python3.8/site-
packages (from pandas->handyspark) (2021.1)
Requirement already satisfied: py4j==0.10.9 in /opt/conda/lib/python3.8/site-
packages (from pyspark->handyspark) (0.10.9)
Requirement already satisfied: joblib>=0.11 in /opt/conda/lib/python3.8/site-
packages (from scikit-learn->handyspark) (1.0.1)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/opt/conda/lib/python3.8/site-packages (from scikit-learn->handyspark) (2.1.0)
Installing collected packages: pyarrow, handyspark
WARNING: Value for scheme.headers does not match. Please report this to

<https://github.com/pypa/pip/issues/9617>

distutils: /opt/conda/include/python3.8/UNKNOWN

sysconfig: /opt/conda/include/python3.8
```

```
WARNING: Additional context:
user = False
home = None
root = None
prefix = None
Successfully installed handyspark-0.2.2a1 pyarrow-6.0.1
```

5.1 Logistic Regression

5.1.1 Model training with 3 different parameters

[51]:
```python
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import feature, classification

# default parameters, regParam = 0.0, elasticNetParam = 0.0
lr = LogisticRegression(featuresCol='finalfeatures', labelCol='is_fraud')
lr_model = lr.fit(train)
validations_lr = lr_model.transform(validate)

# Lasso (L1) Regularization, regParam = 0.5, elasticNetParam = 0.0
lr_lasso = LogisticRegression(featuresCol='finalfeatures', labelCol='is_fraud',␣
↪regParam=0.5)
lr_lasso_model = lr_lasso.fit(train)
validations_lr_lasso = lr_lasso_model.transform(validate)

# Ridge (L2) Regularization, regParam = 0.0, elasticNetParam = 0.5
lr_ridge = LogisticRegression(featuresCol='finalfeatures', labelCol='is_fraud',␣
↪elasticNetParam=1.0)
lr_ridge_model = lr_ridge.fit(train)
validations_lr_ridge = lr_ridge_model.transform(validate)


# Defining the evaluator to find out the best cross validated model

from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Area under PR curve (main focus since this is imbalanced dataset)
bce_pr = BinaryClassificationEvaluator(labelCol = 'is_fraud',␣
↪metricName='areaUnderPR')

# Area under ROC
bce_roc = BinaryClassificationEvaluator(labelCol = 'is_fraud')
```

```
print('Area under PR curve for Logistic Regression with no regularization: {0}'.
 ↪format(bce_pr.evaluate(validations_lr)))
print('Area under ROC curve for Logistic Regression with no regularization:␣
 ↪{0}'.format(bce_roc.evaluate(validations_lr)))
print('')

print('Area under PR curve for Logistic Regression with L1 regularization: {0}'.
 ↪format(bce_pr.evaluate(validations_lr_lasso)))
print('Area under ROC curve for Logistic Regression with L1 regularization:␣
 ↪{0}'.format(bce_roc.evaluate(validations_lr_lasso)))
print('')

print('Area under PR curve for Logistic Regression with L2 regularization: {0}'.
 ↪format(bce_pr.evaluate(validations_lr_ridge)))
print('Area under ROC curve for Logistic Regression with L2 regularization:␣
 ↪{0}'.format(bce_roc.evaluate(validations_lr_ridge)))
```

```
Area under PR curve for Logistic Regression with no regularization:
0.5470781462412638
Area under ROC curve for Logistic Regression with no regularization:
0.9572195712627165

Area under PR curve for Logistic Regression with L1 regularization:
0.445964641205791
Area under ROC curve for Logistic Regression with L1 regularization:
0.9866110740654371

Area under PR curve for Logistic Regression with L2 regularization:
0.547124292328769
Area under ROC curve for Logistic Regression with L2 regularization:
0.9572243886291897
```

We use regularization to penalize the cost function in an effort to reduce overfitting. We choose our metric as Area Under PR Curve to find the model with best metric.

5.1.2 Estimating generalization performance for no regularization model

[52]:
```
predictions_lr = lr_model.transform(test)

from pyspark.sql.types import FloatType
from pyspark.mllib.evaluation import MulticlassMetrics

#select only prediction and label columns
preds_and_labels_lr = predictions_lr.select(['prediction','is_fraud']).\
                                withColumn('is_fraud', func.col('is_fraud').
 ↪cast(FloatType())).orderBy('prediction')
```

5.1.3 Confusion Matrix

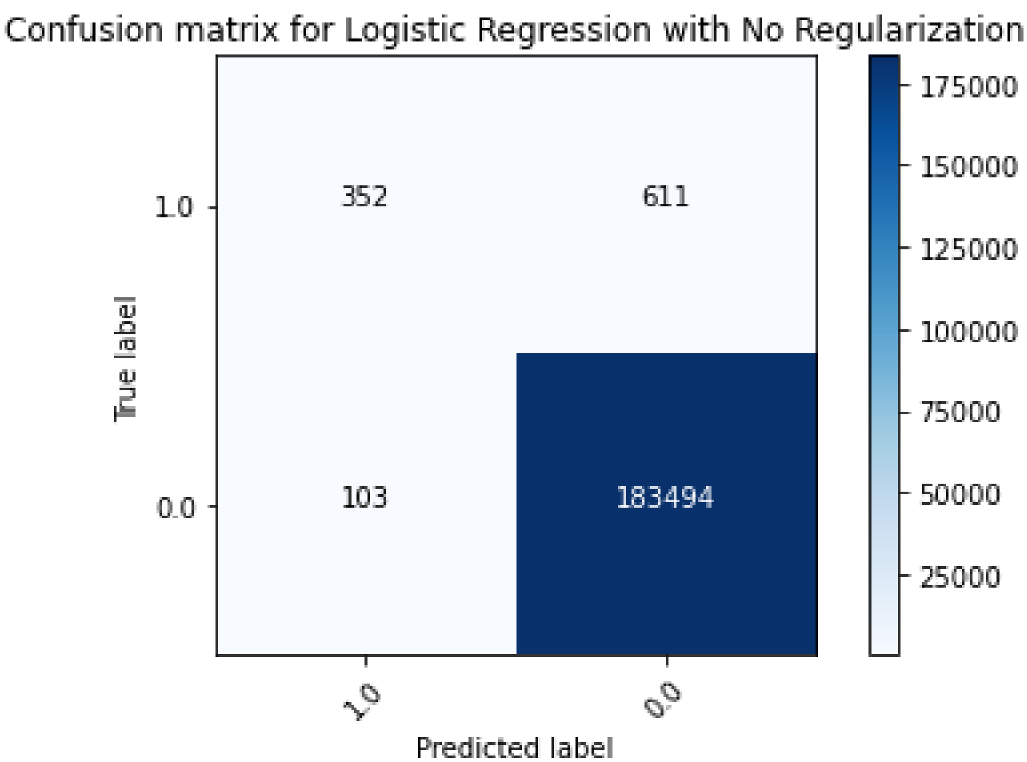
[53]:
```
confusion_matrix_lr = MulticlassMetrics(preds_and_labels_lr.rdd.map(tuple)).
 ↪confusionMatrix().toArray()
confusion_matrix_lr
```

[53]:
```
array([[1.83494e+05, 1.03000e+02],
       [6.11000e+02, 3.52000e+02]])
```

[54]:
```
# function to plot confusion matrix. Necessary to execute next cell

class_names=[1.0,0.0]
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

[55]:
```python
from sklearn.metrics import confusion_matrix

class_names=[1.0,0.0]

y_true_lr = predictions_lr.select("is_fraud")
y_true_lr = y_true_lr.toPandas()

y_pred_lr = predictions_lr.select("prediction")
y_pred_lr = y_pred_lr.toPandas()

cnf_matrix = confusion_matrix(y_true_lr, y_pred_lr,labels=class_names)
#cnf_matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names,
                      title='Confusion matrix for Logistic Regression with No␣
 ↪Regularization')
plt.show()
```

```
Confusion matrix, without normalization
[[    352    611]
 [    103 183494]]
```



5.1.4 Precision, Recall and F1 Score

[56]:
```
precision_lr = confusion_matrix_lr[1,1]/np.add(confusion_matrix_lr[0,1],␣
 ↪confusion_matrix_lr[1,1])
print("Precision: ",precision_lr)
recall_lr = confusion_matrix_lr[1,1]/np.add(confusion_matrix_lr[1,0],␣
 ↪confusion_matrix_lr[1,1])
print("recall: ",recall_lr)
f1_score_lr = 2*(precision_lr*recall_lr)/(precision_lr+recall_lr)
print("f1_score: ",f1_score_lr)
```

```
Precision:  0.7736263736263737
recall:  0.3655244029075805
f1_score:  0.4964739069111425
```

When we set a threshold of 0.5 for classification i.e. given a transaction, if the probability of the transaction being fraud is greater than 0.5, we will classify it as fraud, precision is 0.73 and recall is 0.36

We want to maximize recall since it will be lethal to classify a fraudulent transaction as legitimate. But there is a trade-off between precision and recall. So, we will try to maximize the F-1 score (harmonic mean of precision and recall) by finding out the optimal probability threshold.

5.1.5 Area under ROC Curve and Precision-Recall Curve

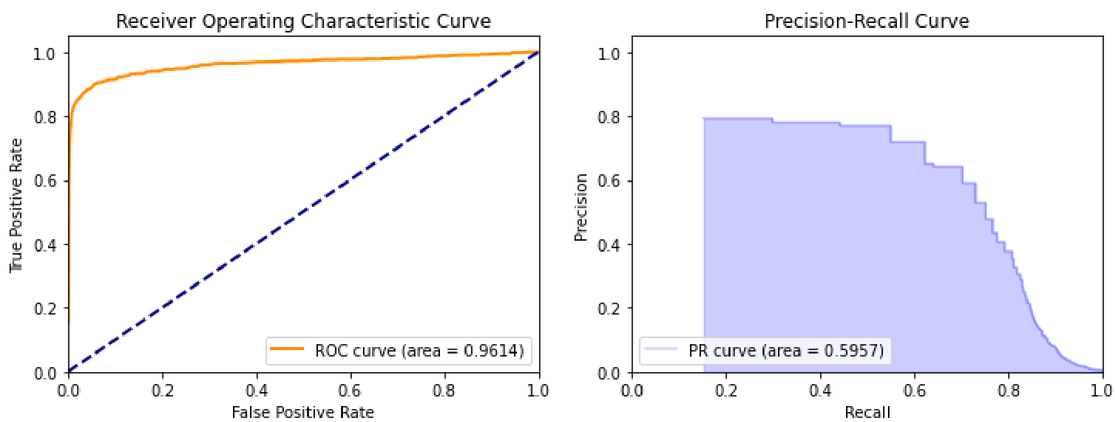
[57]:
```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Let's use the run-of-the-mill evaluator
evaluator = BinaryClassificationEvaluator(labelCol='is_fraud')

# We have only two choices: area under ROC and PR curves :-(
auroc_lr = evaluator.evaluate(predictions_lr, {evaluator.metricName:␣
 ↪"areaUnderROC"})
auprc_lr = evaluator.evaluate(predictions_lr, {evaluator.metricName:␣
 ↪"areaUnderPR"})
print("Area under ROC Curve for test data: {:.4f}".format(auroc_lr))
print("Area under PR Curve for test data: {:.4f}".format(auprc_lr))
```

```
Area under ROC Curve for test data: 0.9614
Area under PR Curve for test data: 0.5955
```

[58]:
```
from handyspark import *
# Creates instance of extended version of BinaryClassificationMetrics
# using a DataFrame and its probability and label columns, as the output
# from the classifier
bcm = BinaryClassificationMetrics(predictions_lr, scoreCol='probability',␣
 ↪labelCol='is_fraud')
```

```
# But now we can PLOT both ROC and PR curves!
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm.plot_roc_curve(ax=axs[0])
bcm.plot_pr_curve(ax=axs[1])

# We can also get all metrics (FPR, Recall and Precision) by threshold
#bcm.getMetricsByThreshold().filter('fpr between 0.19 and 0.21').toPandas()

# And get the confusion matrix for any threshold we want
#bcm.print_confusion_matrix(.415856)
```

[58]:
```
<AxesSubplot:title={'center':'Precision-Recall Curve'}, xlabel='Recall',
ylabel='Precision'>
```



Let's tune the hyperparameters of l1 and l2 penalties to get the best performnace parameters and essentially best PR curve.

5.1.6 Hyperparameter tuning to find the best lr model. Parameters: regParam = [0.0, 0.1, 0.5], elasticNetParam = [0.0, 0.1, 0.5]

[59]:
```
# https://dhiraj-p-rai.medium.com/logistic-regression-in-spark-ml-8a95b5f5434c

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

train_cv, hold_out_test_cv = df.randomSplit([0.9, 0.1])

lr = LogisticRegression(labelCol="is_fraud", featuresCol="finalfeatures",␣
↪maxIter=100)

paramGrid = ParamGridBuilder()\
```

```
    .addGrid(lr.regParam,[0.0, 0.1, 1.0])\
    .addGrid(lr.elasticNetParam,[0.0, 0.1, 1.0])\
    .build()

# Create 5-fold CrossValidator
lr_cvEstimator = CrossValidator(estimator=lr, estimatorParamMaps=paramGrid, \
                                ␣
 ↪evaluator=BinaryClassificationEvaluator(metricName='areaUnderPR',␣
 ↪labelCol='is_fraud'),\
                                numFolds=3, seed = 0)

# Run cross validations
lr_cvModel = lr_cvEstimator.fit(train_cv)
lr_cvModel.getEstimatorParamMaps()[np.argmax(lr_cvModel.avgMetrics)]
```

[59]:

```
{Param(parent='LogisticRegression_582620fe7cb2', name='regParam',
doc='regularization parameter (>= 0).'): 0.0,
 Param(parent='LogisticRegression_582620fe7cb2', name='elasticNetParam',
doc='the ElasticNet mixing parameter, in range [0, 1]. For alpha = 0, the
penalty is an L2 penalty. For alpha = 1, it is an L1 penalty.'): 0.0}
```

After 3-fold Cross Validation and Hyperparameter tuning, we see that the best L1 and L2 regularization parameters are 0. So, the best model is the model above with default parameters and gives exact same area under ROC and area under precision recall curves.

[60]:

```
best_lrModel = lr.setElasticNetParam(0.0).setRegParam(0.0).fit(train_cv)



best_lrPredictions = best_lrModel.transform(hold_out_test_cv)
best_predsAndLabels_lr = best_lrPredictions.select(['prediction','is_fraud']).\
                                withColumn('is_fraud', func.col('is_fraud').
 ↪cast(FloatType())).orderBy('prediction')

# Precision, Recall and F1 Score

best_confusionMatrix_lr = MulticlassMetrics(best_predsAndLabels_lr.rdd.
 ↪map(tuple)).confusionMatrix().toArray()
precision_lrBest = best_confusionMatrix_lr[1,1]/np.
 ↪add(best_confusionMatrix_lr[0,1], best_confusionMatrix_lr[1,1])
print("Precision: ",precision_lrBest)
recall_lrBest = best_confusionMatrix_lr[1,1]/np.
 ↪add(best_confusionMatrix_lr[1,0], best_confusionMatrix_lr[1,1])
print("recall: ",recall_lrBest)
f1_score_lrBest = 2*(precision_lrBest*recall_lrBest)/
 ↪(precision_lrBest+recall_lrBest)
print("f1_score: ",f1_score_lrBest)
```

```python
# area under ROC and PR Curve
auroc_lrReg = evaluator.evaluate(best_lrPredictions, {evaluator.metricName:␣
↪"areaUnderROC"})
auprc_lrReg = evaluator.evaluate(best_lrPredictions, {evaluator.metricName:␣
↪"areaUnderPR"})
print("Area under ROC Curve for test data: {:.4f}".format(auroc_lrReg))
print("Area under PR Curve for test data: {:.4f}".format(auprc_lrReg))
```

```
Precision:  0.7333333333333333
recall:  0.3724867724867725
f1_score:  0.4940350877192982
Area under ROC Curve for test data: 0.9575
Area under PR Curve for test data: 0.5367
```

[62]:
```python
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
from sklearn.metrics import precision_recall_curve   # Calculate the␣
↪Precision-Recall curve
from sklearn.metrics import f1_score                 # Calculate the F-score

# creating a user defined function to get the probability of positive class.
secondelement=udf(lambda v:float(v[1]),FloatType())

# creating pandas dataframe y_test and y_pred for sklearn where y_pred contains␣
↪the probability of belonging to positive class
y_prob_lr = best_lrPredictions.select(secondelement('probability')).
↪withColumnRenamed('<lambda>(probability)', 'pos_class_prob').toPandas()
y_test = hold_out_test_cv.select('is_fraud').toPandas()

y_pred_lr = best_lrPredictions.select('prediction')

# Create the Precision-Recall curve
precision, recall, thresholds = precision_recall_curve(y_test, y_prob_lr)

# Plot the ROC curve
df_recall_precision_lr = pd.DataFrame({'Precision':precision[:-1],
                                    'Recall':recall[:-1],
                                    'Threshold':thresholds})
```

We will try to improve our model by using two more classification algorithms, Random Forest and Gradient Boosting.

5.2 Random Forest

[75]:
```
from pyspark.ml import feature, classification

# Default parameters
rf_model = classification.RandomForestClassifier(featuresCol='finalfeatures',␣
↪labelCol='is_fraud', seed = 0).\
    fit(train)
```

[76]:
```
feature_importance = pd.DataFrame(list(zip(cols, rf_model.featureImportances.
↪toArray())),
            columns = ['feature', 'importance']).sort_values('importance',␣
↪ascending=False)
feature_importance
```

[76]:

	feature	importance
4	rolling_24h_avg_amt	0.261579
5	rolling_1_week_avg_amt	0.195794
0	amt	0.173329
2	hour_of_transaction	0.113168
7	number_trans_24h	0.079189
6	rolling_1month_avg_amt	0.055444
12	gender-vector	0.032448
8	number_trans_specific_day	0.022210
10	weekly_avg_amt_over_3_months	0.015922
11	category-vector	0.014870
9	number_trans_month	0.014431
13	age_udf_cat-vector	0.000815
1	city_pop	0.000207
3	distance	0.000000
14	day_of_week-vector	0.000000

We can see that the feature engineering step to study the historical pattern of a credit card was very important.

[77]:
```
from pyspark.ml.evaluation import BinaryClassificationEvaluator
bce = BinaryClassificationEvaluator(labelCol = 'is_fraud', metricName =␣
↪'areaUnderPR')

# Finding out area under PR curve for validation dataset
bce.evaluate(rf_model.transform(validate))
```

[77]: 0.855706712057725

5.2.1 Estimating Generalization Performance

[78]:
```
from pyspark.sql.types import FloatType
from pyspark.mllib.evaluation import MulticlassMetrics

predictions_rf = rf_model.transform(test)

#select only prediction and label columns
preds_and_labels_rf = predictions_rf.select(['prediction','is_fraud']).
↪withColumn('is_fraud', func.col('is_fraud').cast(FloatType())).
↪orderBy('prediction')
```

5.2.2 Confusion Matrix

[79]:
```
confusion_matrix_rf = MulticlassMetrics(preds_and_labels_rf.rdd.map(tuple)).
↪confusionMatrix().toArray()
confusion_matrix_rf
```

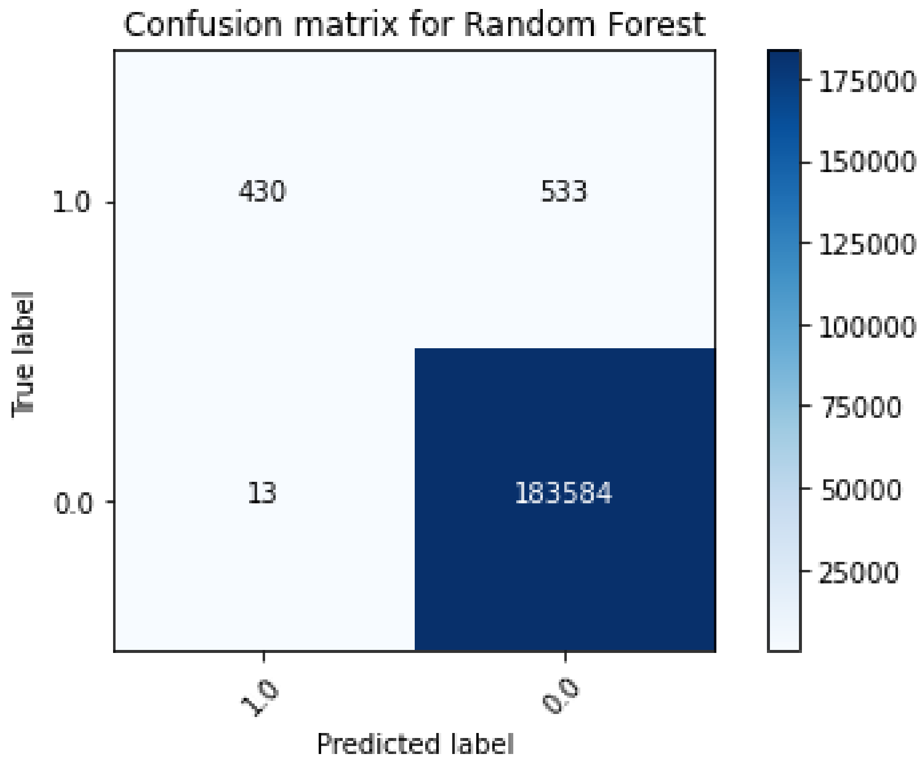
[79]:
```
array([[1.83584e+05, 1.30000e+01],
       [5.33000e+02, 4.30000e+02]])
```

[80]:
```
class_names=[1.0,0.0]
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
```

```python
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
```

[81]:
```python
from sklearn.metrics import confusion_matrix

y_true_rf = predictions_rf.select("is_fraud")
y_true_rf = y_true_rf.toPandas()

y_pred_rf = predictions_rf.select("prediction")
y_pred_rf = y_pred_rf.toPandas()

cnf_matrix1 = confusion_matrix(y_true_rf, y_pred_rf,labels=class_names)
#cnf_matrix
plt.figure()
plot_confusion_matrix(cnf_matrix1, classes=class_names,
                      title='Confusion matrix for Random Forest')
plt.show()
```

```
Confusion matrix, without normalization
[[   430    533]
 [    13 183584]]
```



5.2.3 Precision, Recall and F1 Score

[82]:
```
precision_rf = confusion_matrix_rf[1,1]/np.add(confusion_matrix_rf[0,1],␣
 ↪confusion_matrix_rf[1,1])
print("Precision: ",precision_rf)
recall_rf = confusion_matrix_rf[1,1]/np.add(confusion_matrix_rf[1,0],␣
 ↪confusion_matrix_rf[1,1])
print("recall: ",recall_rf)
f1_score_rf = 2*(precision_rf*recall_rf)/(precision_rf+recall_rf)
print("f1_score: ",f1_score_rf)
```

```
Precision:  0.9706546275395034
recall:  0.446521287642783
f1_score:  0.6116642958748223
```

5.2.4 ROC and Precision-Recall Curve

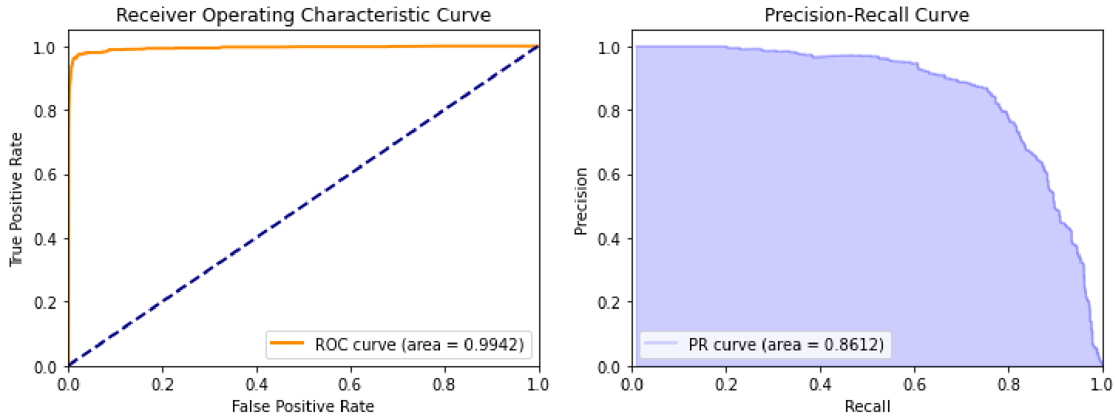
[83]:
```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Let's use the run-of-the-mill evaluator
evaluator = BinaryClassificationEvaluator(labelCol='is_fraud')

# We have only two choices: area under ROC and PR curves :-(
auroc_rf = evaluator.evaluate(predictions_rf, {evaluator.metricName:␣
↪"areaUnderROC"})
auprc_rf = evaluator.evaluate(predictions_rf, {evaluator.metricName:␣
↪"areaUnderPR"})
print("Area under ROC Curve: {:.4f}".format(auroc_rf))
print("Area under PR Curve: {:.4f}".format(auprc_rf))
```

```
Area under ROC Curve: 0.9942
Area under PR Curve: 0.8612
```

[84]:
```
from handyspark import *
# Creates instance of extended version of BinaryClassificationMetrics
# using a DataFrame and its probability and label columns, as the output
# from the classifier
bcm = BinaryClassificationMetrics(predictions_rf, scoreCol='probability',␣
↪labelCol='is_fraud')

# We still can get the same metrics as the evaluator...
print("Area under ROC Curve: {:.4f}".format(bcm.areaUnderROC))
print("Area under PR Curve: {:.4f}".format(bcm.areaUnderPR))

# But now we can PLOT both ROC and PR curves!
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm.plot_roc_curve(ax=axs[0])
bcm.plot_pr_curve(ax=axs[1])
```

```
Area under ROC Curve: 0.9942
Area under PR Curve: 0.8612
```

[84]:
```
<AxesSubplot:title={'center':'Precision-Recall Curve'}, xlabel='Recall',
ylabel='Precision'>
```



As we can see from above performance parameters, the precision, recall, and f-1 score for default Random Forest model gives us better results than hyper-tuned and 3 folds cross validated logistic regression model.

Let's try to tune the hyperparameters and apply 3 folds cross validation for Random Forest model to get the better performance parameters as follows: -

5.2.5 Hyperparameter Tuning on numTrees, impurity and maxDepth to try to improve the model

[85]:
```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

train_cv, hold_out_test_cv = df.randomSplit([0.9, 0.1])

numFolds = 3

rf_model_cv = classification.RandomForestClassifier(labelCol="is_fraud",␣
 ↪featuresCol="finalfeatures", seed = 0)
evaluator_rfcv = BinaryClassificationEvaluator(metricName='areaUnderPR',␣
 ↪labelCol='is_fraud')

pipeline_rfcv = Pipeline(stages=[rf_model_cv])
paramGrid_rfcv = ParamGridBuilder()\
    .addGrid(rf_model_cv.numTrees, [10, 20, 30])\
    .addGrid(rf_model_cv.maxDepth, [3,5,10])\
    .addGrid(rf_model_cv.impurity, ['gini','entropy'])\
    .build()

crossval = CrossValidator(
    estimator=pipeline_rfcv,
    estimatorParamMaps=paramGrid_rfcv,
    evaluator=evaluator_rfcv,
```

```
    numFolds=numFolds)

rf_grid_model = crossval.fit(train_cv)
```

[86]:
```
print("The best parameters for the random forest models are:\n ",rf_grid_model.
 ↪getEstimatorParamMaps()[np.argmax(rf_grid_model.avgMetrics)])
```

```
The best parameters for the random forest models are:
  {Param(parent='RandomForestClassifier_37ee73bc47da', name='numTrees',
doc='Number of trees to train (>= 1).'): 30,
Param(parent='RandomForestClassifier_37ee73bc47da', name='maxDepth',
doc='Maximum depth of the tree. (>= 0) E.g., depth 0 means 1 leaf node; depth 1
means 1 internal node + 2 leaf nodes.'): 10,
Param(parent='RandomForestClassifier_37ee73bc47da', name='impurity',
doc='Criterion used for information gain calculation (case-insensitive).
Supported options: entropy, gini'): 'gini'}
```

5.2.6 Creating the best Random Forest Classifier

[87]:
```
best_model_rf = rf_model_cv.setImpurity('gini').setMaxDepth(10).setNumTrees(20).
 ↪fit(train_cv)
```

[88]:
```
# Feature Importance
feature_importance = pd.DataFrame(list(zip(cols, best_model_rf.
 ↪featureImportances.toArray())),
             columns = ['feature', 'importance']).sort_values('importance',␣
 ↪ascending=False)
feature_importance
```

[88]:

	feature	importance
4	rolling_24h_avg_amt	0.202989
5	rolling_1_week_avg_amt	0.185148
0	amt	0.162341
2	hour_of_transaction	0.139226
7	number_trans_24h	0.079660
6	rolling_1month_avg_amt	0.053467
12	gender-vector	0.034443
8	number_trans_specific_day	0.033129
9	number_trans_month	0.024671
10	weekly_avg_amt_over_3_months	0.017363
11	category-vector	0.014238
1	city_pop	0.006614
14	day_of_week-vector	0.002077
3	distance	0.002028
13	age_udf_cat-vector	0.001267

We can again validate that the feature engineering step to study the historical pattern of a credit card was very important.

[89]:
```
# Using hold out test set to evaluate generalized performance.

best_rfPredictions = best_model_rf.transform(hold_out_test_cv)
best_predsAndLabels_rf = best_rfPredictions.select(['prediction','is_fraud']).\
                                withColumn('is_fraud', func.col('is_fraud').
 ↪cast(FloatType())).orderBy('prediction')
```

[90]:
```
# Precision, Recall and F1 Score for the default 0.5 probability

best_confusionMatrix_rf = MulticlassMetrics(best_predsAndLabels_rf.rdd.
 ↪map(tuple)).confusionMatrix().toArray()
precision_rfBest = best_confusionMatrix_rf[1,1]/np.
 ↪add(best_confusionMatrix_rf[0,1], best_confusionMatrix_rf[1,1])
print("Precision: ",precision_rfBest)
recall_rfBest = best_confusionMatrix_rf[1,1]/np.
 ↪add(best_confusionMatrix_rf[1,0], best_confusionMatrix_rf[1,1])
print("recall: ",recall_rfBest)
f1_score_rfBest = 2*(precision_rfBest*recall_rfBest)/
 ↪(precision_rfBest+recall_rfBest)
print("f1_score: ",f1_score_rfBest)
```

```
Precision:  0.9566982408660352
recall:  0.7379958246346555
f1_score:  0.8332351208014143
```

[91]:
```
auroc_rfBest = evaluator.evaluate(best_rfPredictions, {evaluator.metricName:␣
 ↪"areaUnderROC"})
auprc_rfBest = evaluator.evaluate(best_rfPredictions, {evaluator.metricName:␣
 ↪"areaUnderPR"})
print("Area under ROC Curve for test data: {:.4f}".format(auroc_rfBest))
print("Area under PR Curve for test data: {:.4f}".format(auprc_rfBest))
```

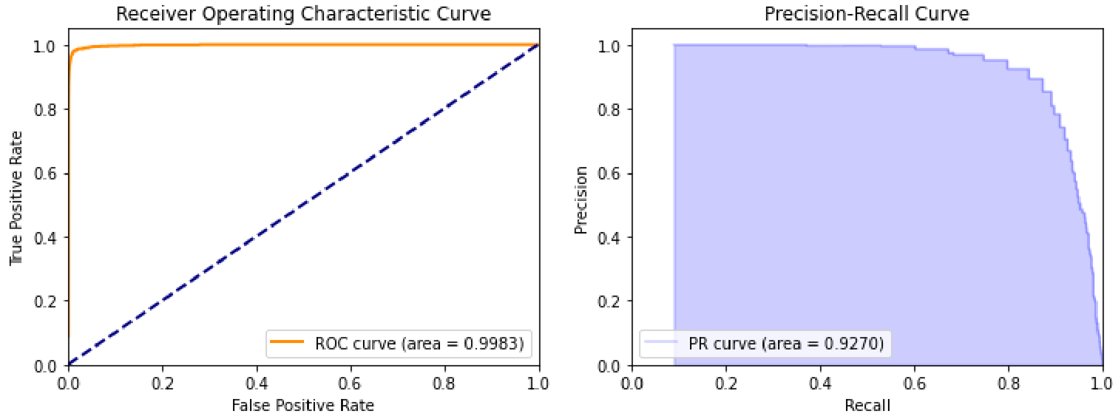
```
Area under ROC Curve for test data: 0.9983
Area under PR Curve for test data: 0.9274
```

[92]:
```
from handyspark import *
# Creates instance of extended version of BinaryClassificationMetrics
# using a DataFrame and its probability and label columns, as the output
# from the classifier
bcm_best_rf = BinaryClassificationMetrics(best_rfPredictions,␣
 ↪scoreCol='probability', labelCol='is_fraud')

# We still can get the same metrics as the evaluator...
print("Area under ROC Curve: {:.4f}".format(bcm_best_rf.areaUnderROC))
```

```
print("Area under PR Curve: {:.4f}".format(bcm_best_rf.areaUnderPR))

# But now we can PLOT both ROC and PR curves!
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm_best_rf.plot_roc_curve(ax=axs[0])
bcm_best_rf.plot_pr_curve(ax=axs[1])
```

```
Area under ROC Curve: 0.9983
Area under PR Curve: 0.9270
```

[92]:
```
<AxesSubplot:title={'center':'Precision-Recall Curve'}, xlabel='Recall',
ylabel='Precision'>
```



The best model gives us higher recall and precision.

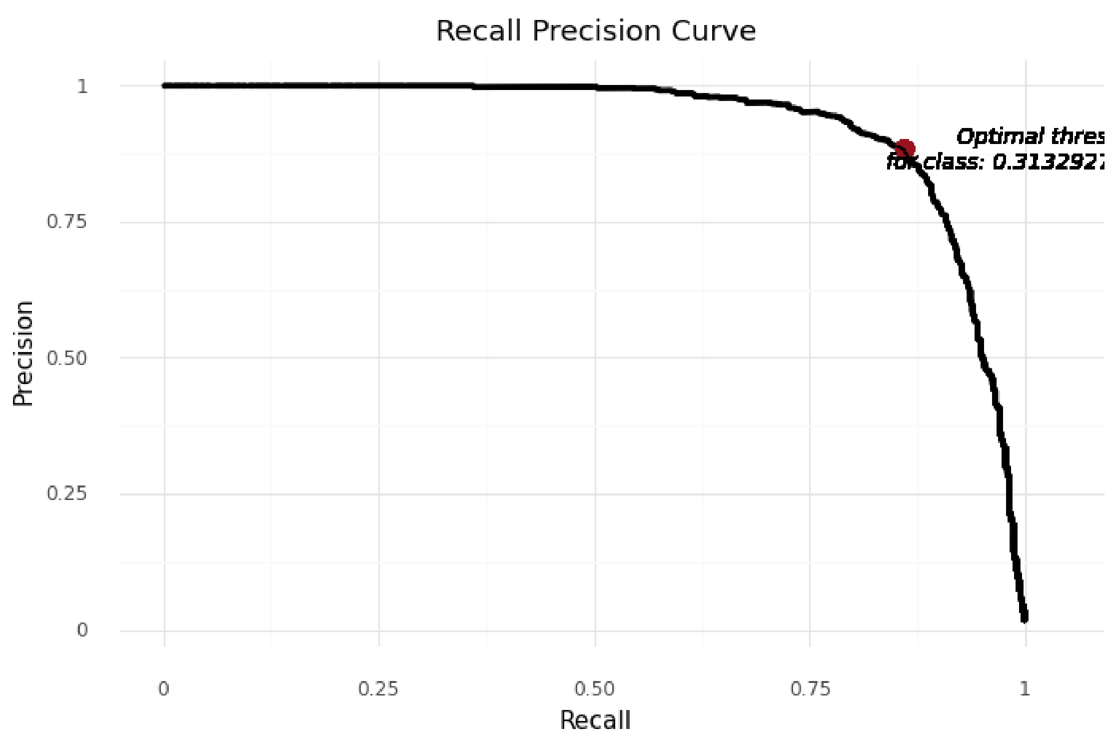
5.2.7 Visualization to understand the Best threshold value for classifying fraudulent and legitimate transactions.

[93]:
```
# The precision-Recall curve for finding the optimal threshold
# https://medium.com/@douglaspsteen/precision-recall-curves-d32e5b290248
#https://towardsdatascience.com/
↪optimal-threshold-for-imbalanced-classification-5884e870c293

from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
from sklearn.metrics import precision_recall_curve    # Calculate the␣
↪Precision-Recall curve
from sklearn.metrics import f1_score                  # Calculate the F-score
from plotnine import *
import plotnine
```

```
# creating a udf for extracting prob belonging to positive class
# https://stackoverflow.com/questions/44425159/
↪access-element-of-a-vector-in-a-spark-dataframe-logistic-regression-probability

secondelement=udf(lambda v:float(v[1]),FloatType())

# creating pandas dataframe y_test and y_pred for sklearn where y_pred contains␣
↪the probability of belonging to positive class
y_prob_rf = best_rfPredictions.select(secondelement('probability')).
↪withColumnRenamed('<lambda>(probability)', 'pos_class_prob').toPandas()
y_test_rf = hold_out_test_cv.select('is_fraud').toPandas()

y_pred_rf = best_rfPredictions.select('prediction')

from sklearn.metrics import precision_recall_curve   # Calculate the␣
↪Precision-Recall curve
from sklearn.metrics import f1_score                 # Calculate the F-score
# Import module for data visualization
from plotnine import *
import plotnine

# Create the Precision-Recall curve
precision, recall, thresholds = precision_recall_curve(y_test_rf, y_prob_rf)

# Plot the ROC curve
df_recall_precision_rf = pd.DataFrame({'Precision':precision[:-1],
                                    'Recall':recall[:-1],
                                    'Threshold':thresholds})

# Calculate the f-score
fscore = (2 * precision * recall) / (precision + recall)

# Find the optimal threshold
index = np.argmax(fscore)
thresholdOpt = thresholds[index]
fscoreOpt = fscore[index]
recallOpt = recall[index]
precisionOpt = precision[index]
print('Best Threshold: {} with F-Score: {}'.format(thresholdOpt, fscoreOpt))
print('Recall: {}, Precision: {}'.format(recallOpt, precisionOpt))

# Create a data viz
plotnine.options.figure_size = (8, 4.8)
(
    ggplot(data = df_recall_precision_rf)+
    geom_point(aes(x = 'Recall',
                   y = 'Precision'),
```

```
                size = 0.4)+
    # Best threshold
    geom_point(aes(x = recallOpt,
                   y = precisionOpt),
                color = '#981220',
                size = 4)+
    geom_line(aes(x = 'Recall',
                  y = 'Precision'))+
    # Annotate the text
    geom_text(aes(x = recallOpt,
                  y = precisionOpt),
               label = 'Optimal threshold \n for class: {}'.format(thresholdOpt),
               nudge_x = 0.18,
               nudge_y = 0,
               size = 10,
               fontstyle = 'italic')+
    labs(title = 'Recall Precision Curve')+
    xlab('Recall')+
    ylab('Precision')+
    theme_minimal()
)
```

```
Best Threshold: 0.3132927417755127 with F-Score: 0.8714965626652564
Recall: 0.860125260960334, Precision: 0.8831725616291533
```



[93]: <ggplot: (8789514648661)>

The optimum threshold value for class to maximize the F-1 score tells us that, the probability of fraud for a given transaction is 0.31.

5.3 Gradient Boosting

5.3.1 Model using default parameters

[94]:
```
from pyspark.ml import feature, classification

# default parameters
gbt_model = classification.GBTClassifier(featuresCol='finalfeatures',␣
↪labelCol='is_fraud').\
    fit(train)
```

[95]:
```
feature_importance_gbt = pd.DataFrame(list(zip(cols, gbt_model.
↪featureImportances.toArray())),
            columns = ['feature', 'importance']).sort_values('importance',␣
↪ascending=False)
feature_importance_gbt
```

[95]:

	feature	importance
6	rolling_1month_avg_amt	0.154364
4	rolling_24h_avg_amt	0.141913
0	amt	0.136946
7	number_trans_24h	0.124302
12	gender-vector	0.110096
2	hour_of_transaction	0.091832
5	rolling_1_week_avg_amt	0.088127
8	number_trans_specific_day	0.052384
11	category-vector	0.043017
9	number_trans_month	0.023230
10	weekly_avg_amt_over_3_months	0.004996
1	city_pop	0.003467
13	age_udf_cat-vector	0.000833
14	day_of_week-vector	0.000175
3	distance	0.000006

we can see that Gradient boosting model follows the similar trend that the feature engineering step to study the historical pattern of a credit card was very important. The feature importance shed a light that these features are important from classification point of view and since they are same to that of best Random Forest model, it supports the fact that analyzing the customer's historical behavior based on transaction pattern is important in identifying whether a certain transaction is fraudulent or legitimate.

Let's evaluate the model

[96]:
```
print('Area under PR curve for Gradient Boosted Trees on validation set: {0}'.
 ↪format(bce.evaluate(gbt_model.transform(validate))))
```

```
Area under PR curve for Gradient Boosted Trees on validation set:
0.9019509057018289
```

[97]:
```
from pyspark.sql.types import FloatType
from pyspark.mllib.evaluation import MulticlassMetrics

predictions_gbt = gbt_model.transform(test)
preds_and_labels_gbt = predictions_gbt.select(['prediction','is_fraud']).
 ↪withColumn('is_fraud', func.col('is_fraud').cast(FloatType())).
 ↪orderBy('prediction')

# getting the predicted probabilities
prob_gbt = predictions_gbt.select('probability')
```

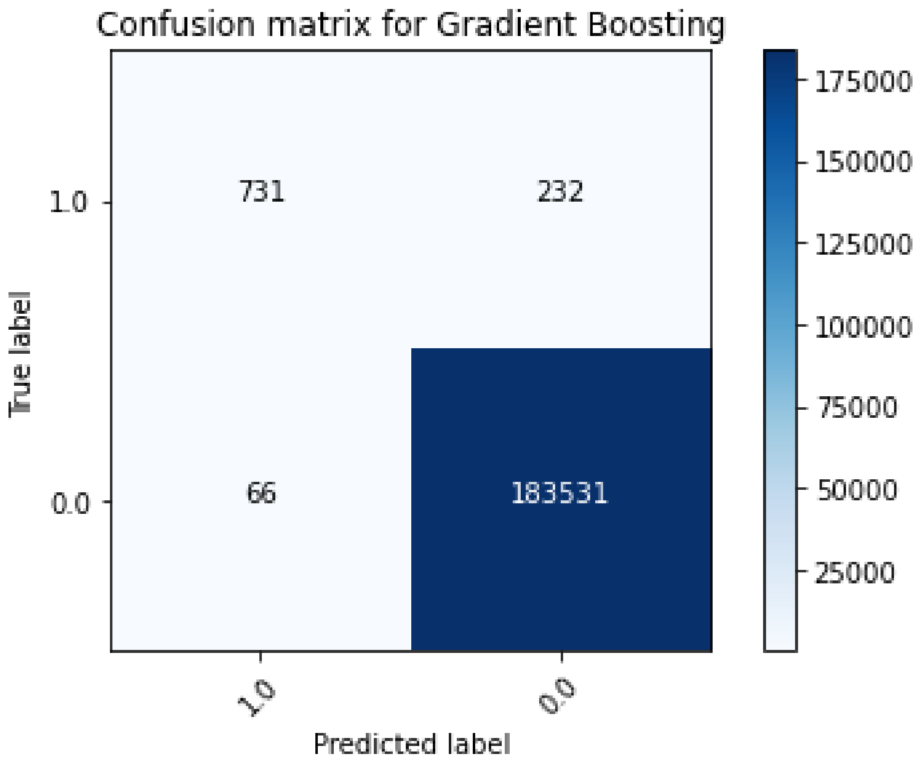
5.3.2 Confusion Matrix

[98]:
```
confusion_matrix_gbt = MulticlassMetrics(preds_and_labels_gbt.rdd.map(tuple)).
 ↪confusionMatrix().toArray()
confusion_matrix_gbt
```

[98]:
```
array([[1.83531e+05, 6.60000e+01],
       [2.32000e+02, 7.31000e+02]])
```

[99]:
```
from sklearn.metrics import confusion_matrix

y_true_gbt = predictions_gbt.select("is_fraud")
y_true_gbt = y_true_gbt.toPandas()

y_pred_gbt = predictions_gbt.select("prediction")
y_pred_gbt = y_pred_gbt.toPandas()

cnf_matrix2 = confusion_matrix(y_true_gbt, y_pred_gbt,labels=class_names)
#cnf_matrix
plt.figure()
plot_confusion_matrix(cnf_matrix2, classes=class_names,
                      title='Confusion matrix for Gradient Boosting')
plt.show()
```

```
Confusion matrix, without normalization
[[   731    232]
 [    66 183531]]
```



5.3.3 Precision, Recall and F1 Score

[100]:
```
precision_gbt = confusion_matrix_gbt[1,1]/np.add(confusion_matrix_gbt[0,1],␣
↪confusion_matrix_gbt[1,1])
print("Precision: ",precision_gbt)
recall_gbt = confusion_matrix_gbt[1,1]/np.add(confusion_matrix_gbt[1,0],␣
↪confusion_matrix_gbt[1,1])
print("recall: ",recall_gbt)
f1_score_gbt = 2*(precision_gbt*recall_gbt)/(precision_gbt+recall_gbt)
print("f1_score: ",f1_score_gbt)
```

```
Precision:  0.917189460476788
recall:  0.7590861889927311
f1_score:  0.8306818181818182
```

5.3.4 Area under PR Curve and ROC

[101]:
```
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Let's use the run-of-the-mill evaluator
evaluator = BinaryClassificationEvaluator(labelCol='is_fraud')

# We have only two choices: area under ROC and PR curves :-(
auroc_gbt = evaluator.evaluate(predictions_gbt, {evaluator.metricName:␣
↪"areaUnderROC"})
auprc_gbt = evaluator.evaluate(predictions_gbt, {evaluator.metricName:␣
↪"areaUnderPR"})
print("Area under ROC Curve: {:.4f}".format(auroc_gbt))
print("Area under PR Curve: {:.4f}".format(auprc_gbt))
```

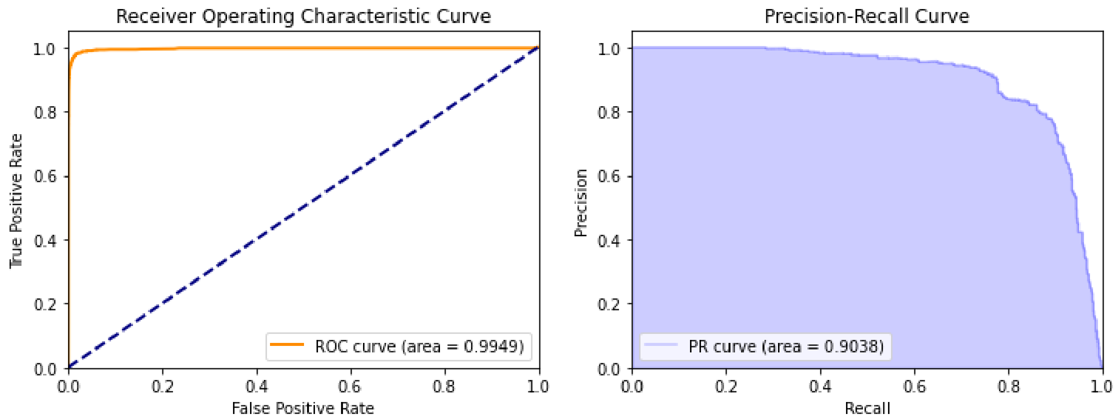
```
Area under ROC Curve: 0.9949
Area under PR Curve: 0.9038
```

5.3.5 Precision-Recall Curve and ROC

[102]:
```
from handyspark import *
# Creates instance of extended version of BinaryClassificationMetrics
# using a DataFrame and its probability and label columns, as the output
# from the classifier
bcm = BinaryClassificationMetrics(predictions_gbt, scoreCol='probability',␣
↪labelCol='is_fraud')

# We still can get the same metrics as the evaluator...
print("Area under ROC Curve: {:.4f}".format(bcm.areaUnderROC))
print("Area under PR Curve: {:.4f}".format(bcm.areaUnderPR))

# But now we can PLOT both ROC and PR curves!
fig, axs = plt.subplots(1, 2, figsize=(12, 4))
bcm.plot_roc_curve(ax=axs[0])
bcm.plot_pr_curve(ax=axs[1])

# We can also get all metrics (FPR, Recall and Precision) by threshold
#bcm.getMetricsByThreshold().filter('fpr between 0.19 and 0.21').toPandas()

# And get the confusion matrix for any threshold we want
#bcm.print_confusion_matrix(.415856)
```

```
Area under ROC Curve: 0.9949
Area under PR Curve: 0.9038
```

[102]: <AxesSubplot:title={'center':'Precision-Recall Curve'}, xlabel='Recall', ylabel='Precision'>



The area under PR curve value observed is very close to the one observed for best Random Forest model. Since the Precision, recall and area under PR curve values are high by using default parameters we have decided not to tune the hyperparameters and apply k-fold cross validation on the data.

Let's proceed to find the optimum threshold value to maximize f-1 score.

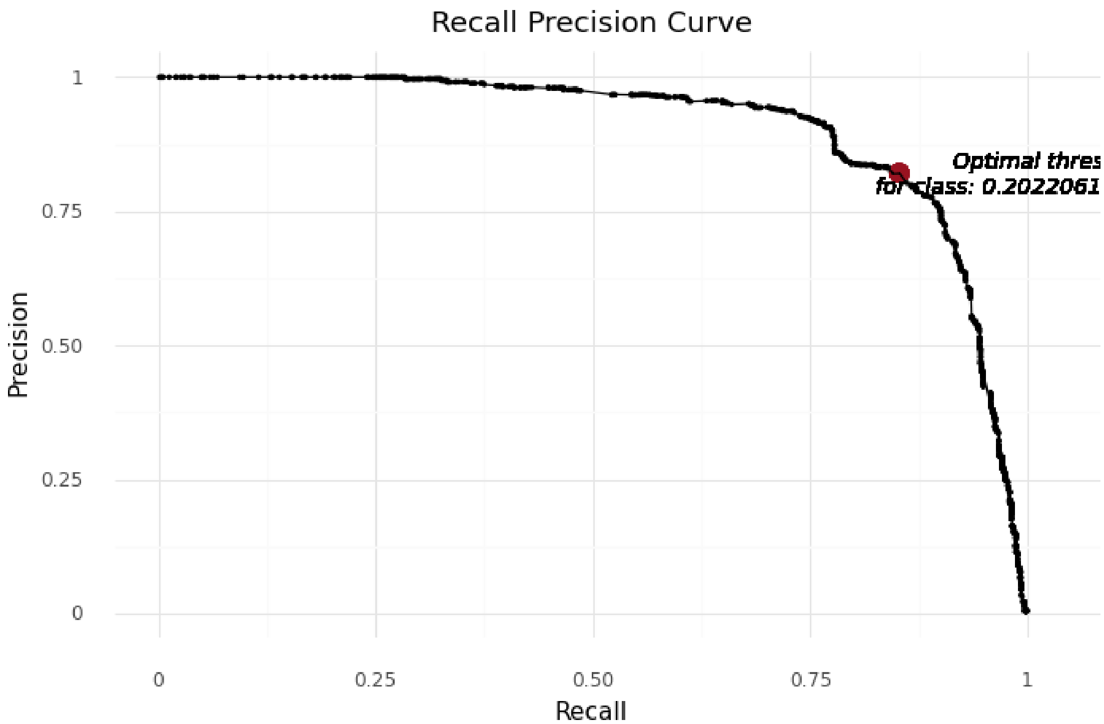
5.3.6 Finding the optimal threshold for classifying a transaction as fraud to achieve the highest F1 score

[103]:
```
# The precision-Recall curve for finding the optimal threshold

# creating pandas dataframe y_test and y_pred for sklearn where y_pred contains␣
↪the probability of belonging to positive class
y_prob_gbt = predictions_gbt.select(secondelement('probability')).
↪withColumnRenamed('<lambda>(probability)', 'pos_class_prob').toPandas()
y_test_gbt = test.select('is_fraud').toPandas()

y_pred_gbt = predictions_gbt.select('prediction')

from sklearn.metrics import precision_recall_curve   # Calculate the␣
↪Precision-Recall curve
from sklearn.metrics import f1_score                 # Calculate the F-score
# Import module for data visualization
from plotnine import *
import plotnine

# Create the Precision-Recall curve
precision, recall, thresholds = precision_recall_curve(y_test_gbt, y_prob_gbt)
```

```
# Plot the ROC curve
df_recall_precision_gbt = pd.DataFrame({'Precision':precision[:-1],
                                        'Recall':recall[:-1],
                                        'Threshold':thresholds})

# Calculate the f-score
fscore = (2 * precision * recall) / (precision + recall)

# Find the optimal threshold
index = np.argmax(fscore)
thresholdOpt = thresholds[index]
fscoreOpt = fscore[index]
recallOpt = recall[index]
precisionOpt = precision[index]
print('Best Threshold: {} with F-Score: {}'.format(thresholdOpt, fscoreOpt))
print('Recall: {}, Precision: {}'.format(recallOpt, precisionOpt))

# Create a data viz
plotnine.options.figure_size = (8, 4.8)
(
    ggplot(data = df_recall_precision_gbt)+
    geom_point(aes(x = 'Recall',
                   y = 'Precision'),
               size = 0.4)+
    # Best threshold
    geom_point(aes(x = recallOpt,
                   y = precisionOpt),
               color = '#981220',
               size = 4)+
    geom_line(aes(x = 'Recall',
                  y = 'Precision'))+
    # Annotate the text
    geom_text(aes(x = recallOpt,
                  y = precisionOpt),
              label = 'Optimal threshold \n for class: {}'.format(thresholdOpt),
              nudge_x = 0.18,
              nudge_y = 0,
              size = 10,
              fontstyle = 'italic')+
    labs(title = 'Recall Precision Curve')+
    xlab('Recall')+
    ylab('Precision')+
    theme_minimal()
)
```

```
Best Threshold: 0.20220613479614258 with F-Score: 0.836474783494651
```

```
Recall: 0.8525441329179647, Precision: 0.821
```



[103]: <ggplot: (8789558509993)>

The optimum threshold value for class to maximize the F-1 score tells us that, the probability of fraud for a given transaction is 0.20.

6 Combining Model Evaluation Results

[104]:
```
# initialize list of lists
data = [['Logistic Regression', precision_lrBest, recall_lrBest,␣
 ↪f1_score_lrBest, auroc_lrReg, auprc_lrReg], \
        ['Random Forest', precision_rf, recall_rf, f1_score_rf, auroc_rf,␣
 ↪auprc_rf], \
        ['Random Forest with Tuning', precision_rfBest, recall_rfBest,␣
 ↪f1_score_rfBest, auroc_rfBest, auprc_rfBest], \
        ['Gradient Boosting Trees', precision_gbt, recall_gbt, f1_score_gbt,␣
 ↪auroc_gbt, auprc_gbt]]

# Create the pandas DataFrame
results_df = pd.DataFrame(data, columns = ['Model', 'Precision',  'Recall',␣
 ↪'F1Score', 'AreaROC', 'AreaPRCurve'])
```

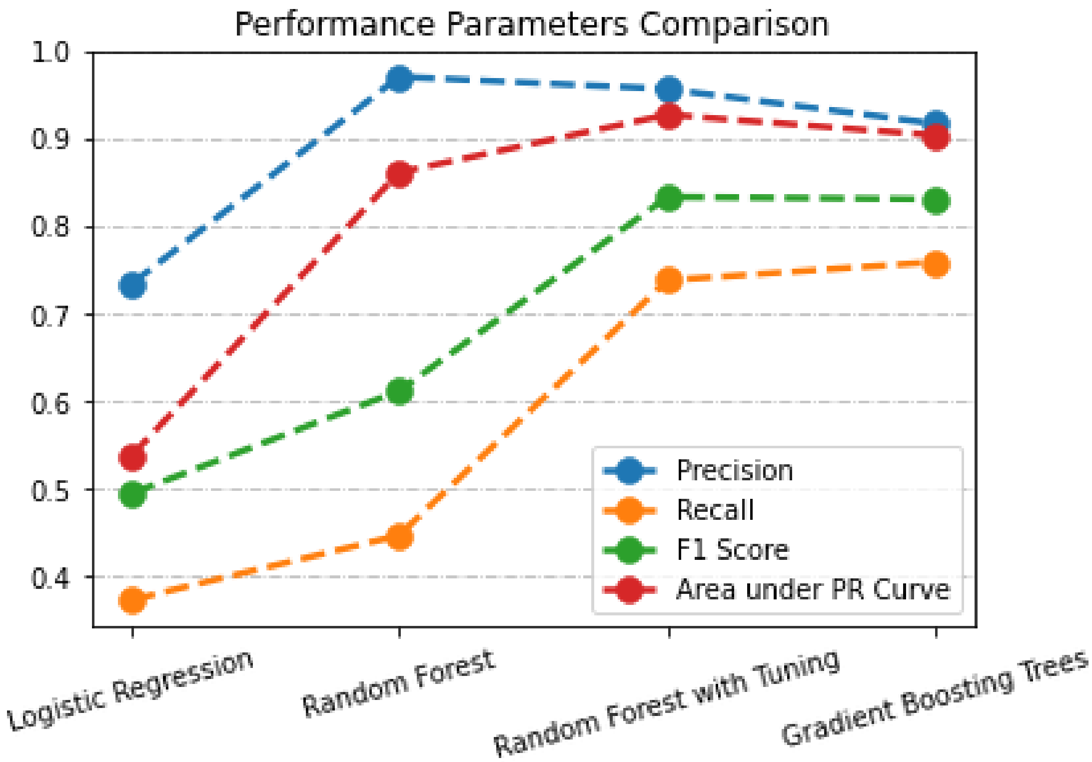
```
# print dataframe.
results_df
```

[104]:

```
                       Model  Precision    Recall   F1Score   AreaROC  \
0        Logistic Regression   0.733333  0.372487  0.494035  0.957463
1              Random Forest   0.970655  0.446521  0.611664  0.994168
2  Random Forest with Tuning   0.956698  0.737996  0.833235  0.998267
3     Gradient Boosting Trees  0.917189  0.759086  0.830682  0.994899

   AreaPRCurve
0     0.536749
1     0.861244
2     0.927412
3     0.903785
```

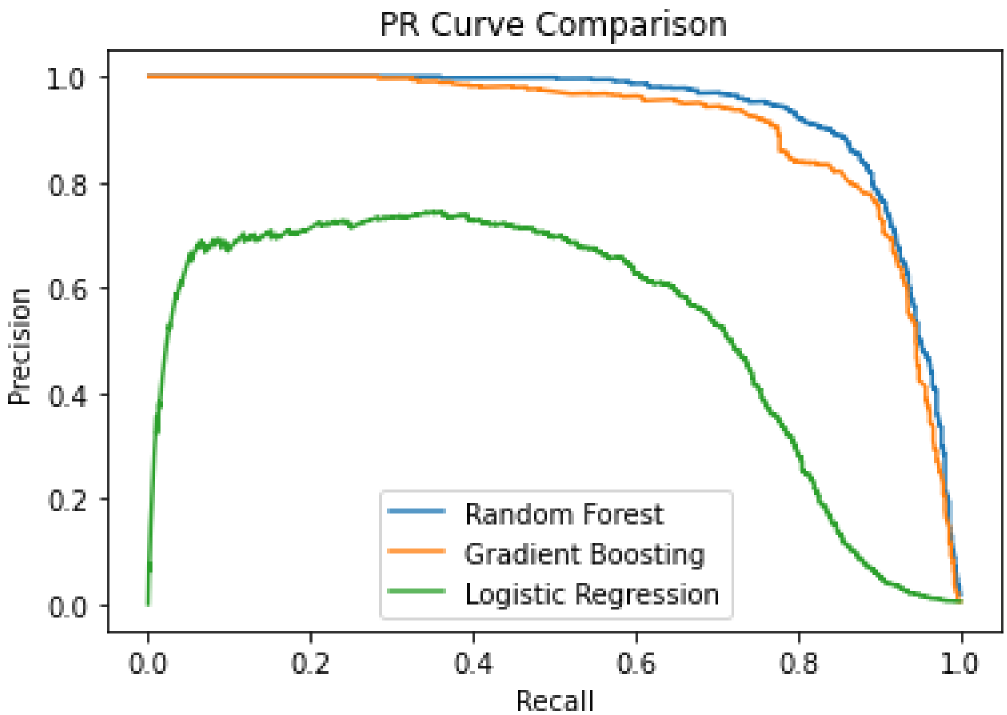
[105]:

```
# plot lines
plt.figure(figsize=(6,4))
plt.plot(results_df.Model, results_df.Precision, label = "Precision",␣
 ↪linestyle="--", linewidth=2.5, marker = 'o', markersize=10)
plt.plot(results_df.Model, results_df.Recall, label = "Recall", linestyle="--",␣
 ↪linewidth=2.5, marker = 'o', markersize=10)
plt.plot(results_df.Model, results_df['F1Score'], label = "F1 Score",␣
 ↪linestyle='--', linewidth=2.5, marker = 'o', markersize=10)
plt.plot(results_df.Model, results_df.AreaPRCurve, label = "Area under PR␣
 ↪Curve", linestyle='--', linewidth=2.5, marker = 'o', markersize=10)
plt.legend()
plt.xticks(rotation = 15) # Rotates X-Axis Ticks by 45-degrees
plt.grid(axis = 'y', linestyle='-.', linewidth=0.7)
plt.title('Performance Parameters Comparison')
plt.show()
```



6.1 Consolidating Precision Recall curves for all the models to compare the highest PR curve model

[106]:
```
# Plot precision-recall curve

fig, ax = plt.subplots(figsize=(6,4))
ax.plot(df_recall_precision_rf.Recall, df_recall_precision_rf.Precision,␣
 ↪label='Random Forest')
ax.plot(df_recall_precision_gbt.Recall, df_recall_precision_gbt.Precision,␣
 ↪label='Gradient Boosting')
ax.plot(df_recall_precision_lr.Recall, df_recall_precision_lr.Precision,␣
 ↪label='Logistic Regression')

# baseline = len(y_test[y_test==1]) / len(y_test)
# ax.plot([0, 1], [baseline, baseline], linestyle='--', label='Baseline')
ax.set_xlabel('Recall')
ax.set_ylabel('Precision')
ax.set_title('PR Curve Comparison')
ax.legend();
```



7 Conclusion

All in all, we can say from the above model evaluation results of all classification models, that the Random Forest model classifies the number of fraud and legitimate transactions in a best way than Logistic Regression and Gradient Boosting models. The 95% precision and 73% recall for best Random Forest model is obtained at 0.5 threshold. Whereas, by keeping the optimal threshold value in the range of 0.28 to 0.35 we can classify the fraudulent and legitimate transactions by maximizing the F1 Score.

8 Recommendations

The feature importance and exploratory analysis gives us vital insights about certain patterns resulting in the credit card fraud transactions. If we utilize our best Random Forest model during those hours or certain days or months, we can detect and reduce the no. of credit card frauds to some extent.

The exploratory analysis and the various categories of the customers available in our data led us to suggest some ways through which we can alert and avoid the credit card frauds.

- Don't use unsecure websites and beware of phishing scams.
- Be on the lookout for skimmers and don't post sensitive information on social media.

- Don’t save your credit card information online and never use debit cards for online purchases.
- Get a chip card with PIN capacity so one can make a habit to shop in stores that have chip readers.
- Don’t trust public Wi-Fi for financial transactions and set up a fraud alert or credit freeze if your card is lost or stolen.
- Audit your online financial accounts and credit card activity online weekly