Bank Marketing

June 27, 2023

Bank Marketing - LISUM21

Group: ZZJ

Name: Zijian Zhou

Email:zhouziji@usc.edu

Country:US

Specialization: Data Science

1. Business Problem
   Description

Problem Statement: The business problem is a binary classification problem. The classification goal is to predict if the client contacted through the marketing campaign will subscribe a term deposit.

[1]:
```
# importing requierd libraries
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from prettytable import PrettyTable
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import roc_auc_score
from sklearn.metrics import accuracy_score
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import roc_curve
from sklearn.metrics import log_loss
import warnings
```

```
warnings.filterwarnings('ignore')
```

[2]:
```
data = pd.read_csv('bank-full.csv', sep=';')
print('Shape of our data {}'.format(data.shape))
```

```
Shape of our data (45211, 17)
```

[3]:
```
data.head()
```

[3]:
```
   age           job  marital  education default  balance housing loan  \
0   58    management  married   tertiary      no     2143     yes   no
1   44    technician   single  secondary      no       29     yes   no
2   33  entrepreneur  married  secondary      no        2     yes  yes
3   47   blue-collar  married    unknown      no     1506     yes   no
4   33       unknown   single    unknown      no        1      no   no

   contact  day month  duration  campaign  pdays  previous poutcome   y
0  unknown    5   may       261         1     -1         0  unknown  no
1  unknown    5   may       151         1     -1         0  unknown  no
2  unknown    5   may        76         1     -1         0  unknown  no
3  unknown    5   may        92         1     -1         0  unknown  no
4  unknown    5   may       198         1     -1         0  unknown  no
```

Dataset Description

The data is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be (or not) subscribed.

Attribute/Features Description:
Dataset have 17 attributes including one dependent attribute and there are 45211 instances/datapoints. So we have 16 predictor/independent attributes and 1 dependent attribute.
* **bank client attributes**: * age: age of client (numeric)
* job : type of job (categorical: "admin.", "unknown", "unemployed", "management", "housemaid", "entrepreneur", "student", "blue-collar", "self-employed", "retired", "technician", "services")
* marital : marital status (categorical: "married", "divorced", "single")
* education: client highest education (categorical: "unknown", "secondary", "primary", "tertiary")
* default: has credit in default? (binary/2-categories: "yes", "no") * balance: average yearly balance, in euros (numeric)
* housing: has housing loan? (binary/2-categories: "yes", "no")
* loan: has personal loan? (binary/2-categories: "yes", "no")
* **related with the last contact of the current campaign**: * contact: contact communication type (categorical: "unknown", "telephone", "cellular") * day: last contact day of the month (numeric) * month: last contact month of year (categorical: "jan", "feb", "mar", …, "nov", "dec") * duration: last contact duration, in seconds (numeric) * **other attributes**: * campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact) * pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric, -1 means client was not previously contacted) * previous: number of contacts performed before this campaign and for this client (numeric) * poutcome: outcome of the previous marketing

campaign (categorical: 'unknown","other", "failure", "success") * **Output variable (desired target)**: * y: has the client subscribed a term deposit? (binary: "yes", "no")

[4]:
```
data.describe(include='all')
```

[4]:
```
                 age          job  marital  education default        balance  \
count   45211.000000        45211    45211      45211   45211   45211.000000
unique           NaN           12        3          4       2            NaN
top              NaN  blue-collar  married  secondary      no            NaN
freq             NaN         9732    27214      23202   44396            NaN
mean       40.936210          NaN      NaN        NaN     NaN    1362.272058
std        10.618762          NaN      NaN        NaN     NaN    3044.765829
min        18.000000          NaN      NaN        NaN     NaN   -8019.000000
25%        33.000000          NaN      NaN        NaN     NaN      72.000000
50%        39.000000          NaN      NaN        NaN     NaN     448.000000
75%        48.000000          NaN      NaN        NaN     NaN    1428.000000
max        95.000000          NaN      NaN        NaN     NaN  102127.000000

       housing   loan   contact           day  month      duration  \
count    45211  45211     45211  45211.000000  45211  45211.000000
unique       2      2         3           NaN     12           NaN
top        yes     no  cellular           NaN    may           NaN
freq     25130  37967     29285           NaN  13766           NaN
mean       NaN    NaN       NaN     15.806419    NaN    258.163080
std        NaN    NaN       NaN      8.322476    NaN    257.527812
min        NaN    NaN       NaN      1.000000    NaN      0.000000
25%        NaN    NaN       NaN      8.000000    NaN    103.000000
50%        NaN    NaN       NaN     16.000000    NaN    180.000000
75%        NaN    NaN       NaN     21.000000    NaN    319.000000
max        NaN    NaN       NaN     31.000000    NaN   4918.000000

            campaign         pdays      previous poutcome      y
count   45211.000000  45211.000000  45211.000000    45211  45211
unique           NaN           NaN           NaN        4      2
top              NaN           NaN           NaN  unknown     no
freq             NaN           NaN           NaN    36959  39922
mean        2.763841     40.197828      0.580323      NaN    NaN
std         3.098021    100.128746      2.303441      NaN    NaN
min         1.000000     -1.000000      0.000000      NaN    NaN
25%         1.000000     -1.000000      0.000000      NaN    NaN
50%         2.000000     -1.000000      0.000000      NaN    NaN
75%         3.000000     -1.000000      0.000000      NaN    NaN
max        63.000000    871.000000    275.000000      NaN    NaN
```

[5]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45211 entries, 0 to 45210
```

```
Data columns (total 17 columns):
age          45211 non-null int64
job          45211 non-null object
marital      45211 non-null object
education    45211 non-null object
default      45211 non-null object
balance      45211 non-null int64
housing      45211 non-null object
loan         45211 non-null object
contact      45211 non-null object
day          45211 non-null int64
month        45211 non-null object
duration     45211 non-null int64
campaign     45211 non-null int64
pdays        45211 non-null int64
previous     45211 non-null int64
poutcome     45211 non-null object
y            45211 non-null object
dtypes: int64(7), object(10)
memory usage: 5.9+ MB
```

Observation:
Our dataset do not have any null/nan/missing values.

[6]:
```
categorical = ['job', 'marital', 'education', 'default', 'housing', 'loan',␣
↪'contact', 'month', 'poutcome']
numerical = [x for x in data.columns.to_list() if x not in categorical]
numerical.remove('y')
```

[7]:
```
print('Categorical features:', categorical)
print('Numerical features:', numerical)
```

```
Categorical features: ['job', 'marital', 'education', 'default', 'housing',
'loan', 'contact', 'month', 'poutcome']
Numerical features: ['age', 'balance', 'day', 'duration', 'campaign', 'pdays',
'previous']
```

[8]:
```
from matplotlib import pyplot as plt
sns.countplot(x=data['y'])
plt.title('Distribution of classes')
plt.xlabel('Target class')
```

[8]:
```
Text(0.5, 0, 'Target class')
```

[9]:
```
data.y.value_counts()
```

[9]:
```
no     39922
yes     5289
Name: y, dtype: int64
```

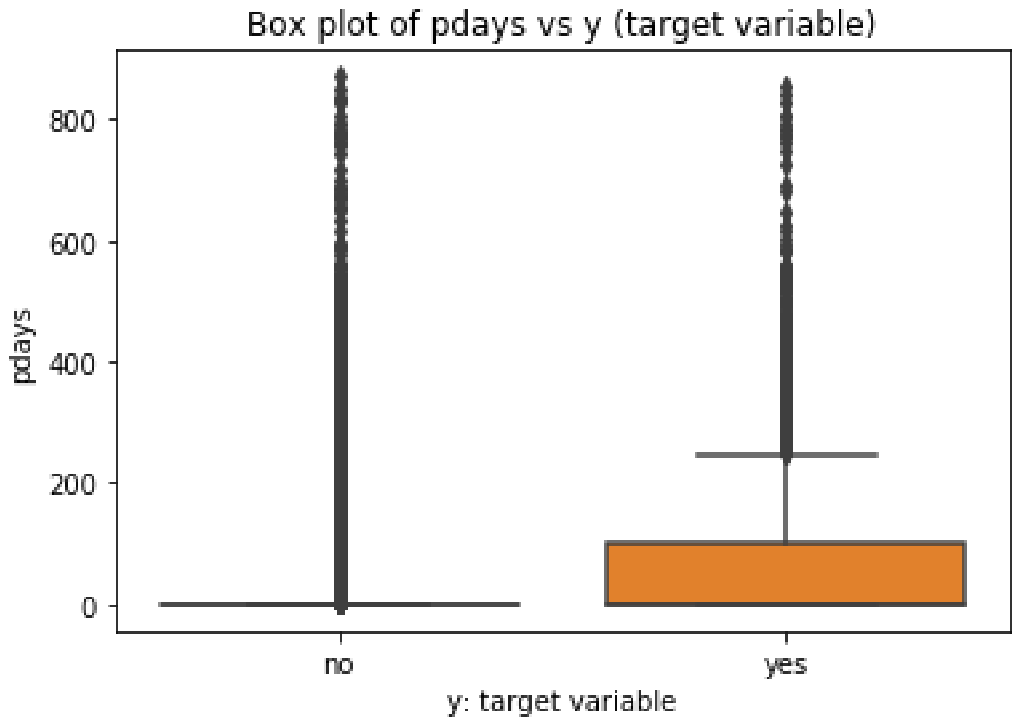
Observation:
Our dataset is highly imbalanced.

Data Analysis

pdays

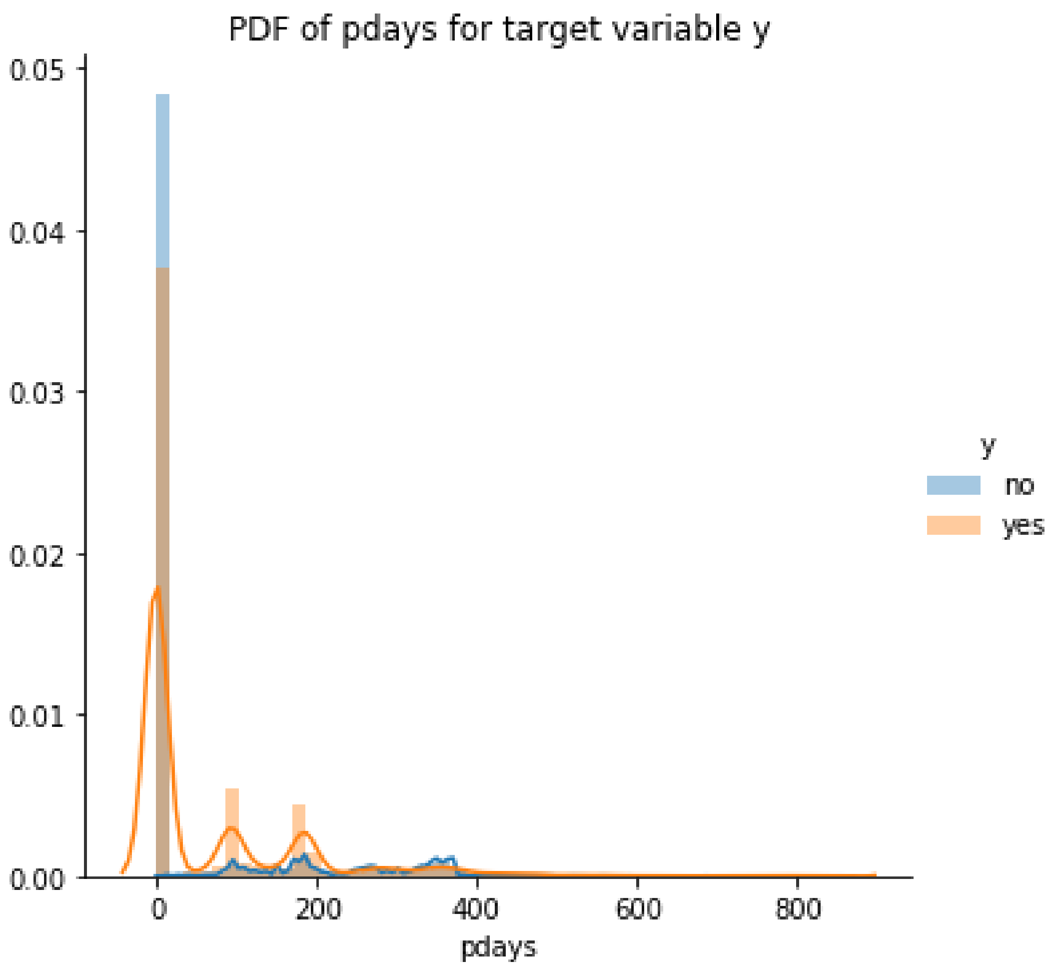
[10]:
```
sns.boxplot(y=data['pdays'], x=data['y'])
plt.title('Box plot of pdays vs y (target variable)')
plt.xlabel('y: target variable')
```

[10]:
```
Text(0.5, 0, 'y: target variable')
```



[11]:
```
sns.FacetGrid(data, hue='y', size=5) \
.map(sns.distplot, 'pdays') \
.add_legend()
plt.title('PDF of pdays for target variable y')
```

[11]: Text(0.5, 1, 'PDF of pdays for target variable y')



[12]:
```
data.pdays.describe()
```

[12]:
```
count    45211.000000
mean        40.197828
std        100.128746
min         -1.000000
25%         -1.000000
50%         -1.000000
75%         -1.000000
max        871.000000
Name: pdays, dtype: float64
```

[13]:
```
for x in range(95, 101 , 1):
    print("{}% of pdays are less than equal to {}".format(x, data.pdays.
 ↪quantile(x/100)))
iqr = data.pdays.quantile(0.75) - data.pdays.quantile(0.25)
print('IQR {}'.format(iqr))
```

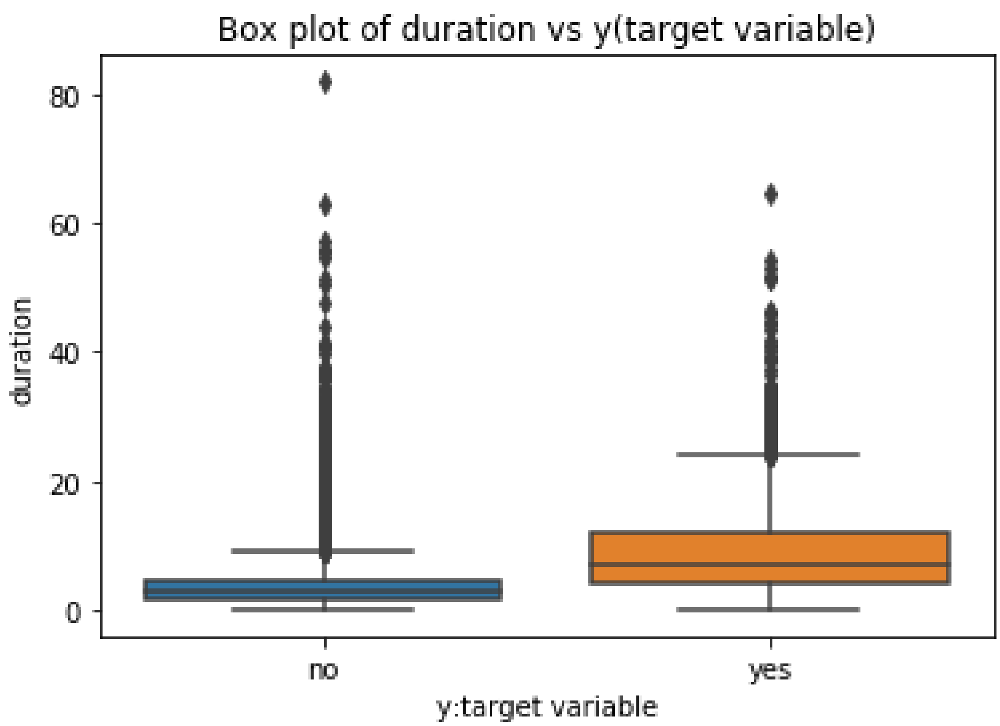
```
95% of pdays are less than equal to 317.0
96% of pdays are less than equal to 337.0
97% of pdays are less than equal to 349.0
98% of pdays are less than equal to 360.0
99% of pdays are less than equal to 370.0
100% of pdays are less than equal to 871.0
IQR 0.0
```

Observation:
* The attribute pdays seems to be important feature as there is a clear distinction in quartile ranges of pdays for target variable yes and no. * 75% clients contacted through campaign are not previously contacted. * Mean of pdays is 40.20 * There are outliers as we can see from boxplot.

duration

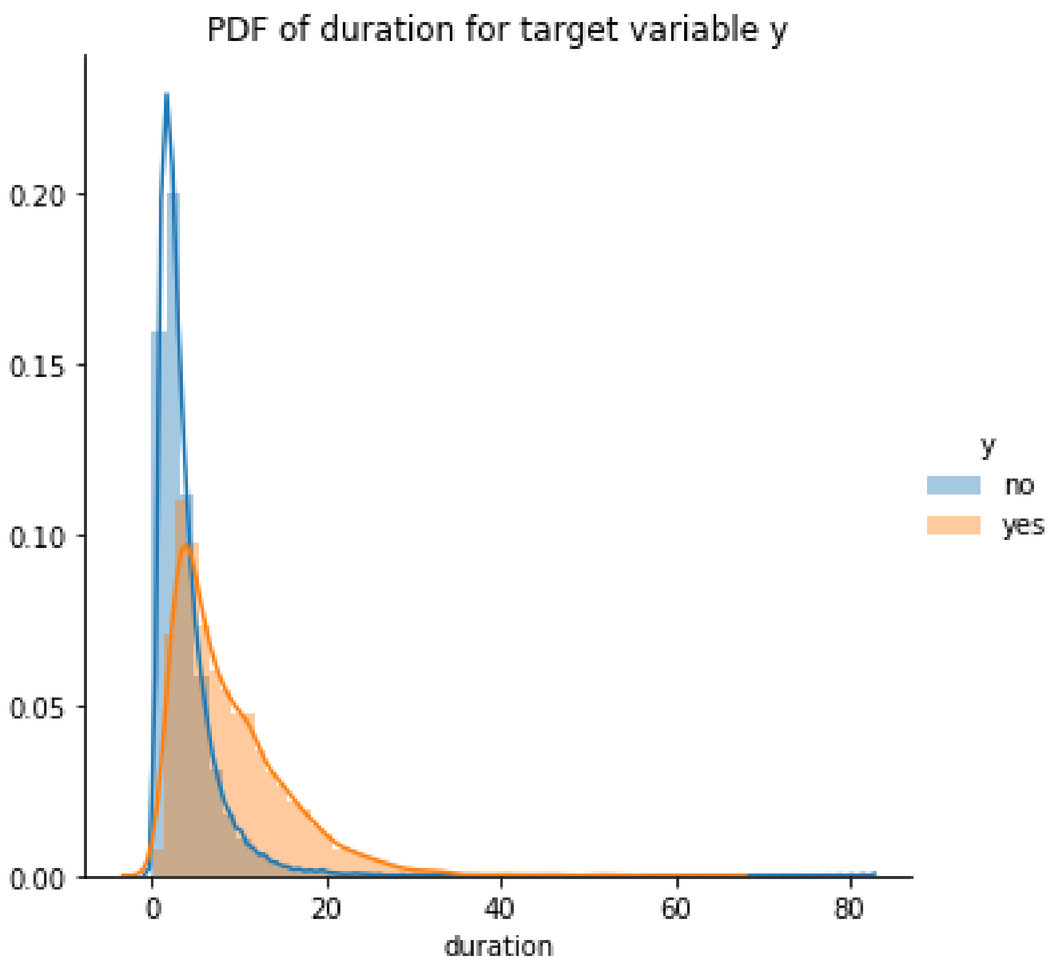
[14]:
```
# converting call duration from seconds to minute
data['duration'] = data['duration']/60
sns.boxplot(y=data['duration'], x=data['y'])
plt.title('Box plot of duration vs y(target variable)')
plt.xlabel('y:target variable')
```

[14]:
```
Text(0.5, 0, 'y:target variable')
```



[17]:
```
sns.FacetGrid(data, hue='y', size=5) \
.map(sns.distplot, 'duration') \
.add_legend()
plt.title('PDF of duration for target variable y')
```

[17]:
```
Text(0.5, 1, 'PDF of duration for target variable y')
```



[18]:
```
data.duration.describe()
```

[18]:
```
count    45211.000000
mean         4.302718
std          4.292130
min          0.000000
25%          1.716667
50%          3.000000
75%          5.316667
max         81.966667
```

```
Name: duration, dtype: float64
```

[19]:
```
for x in range(95, 101 , 1):
    print("{}% of calls have duration less than equal to {}".format(x, data.
 ↪duration.quantile(x/100)))
iqr = data.duration.quantile(0.75) - data.duration.quantile(0.25)
print('IQR {}'.format(iqr))
```

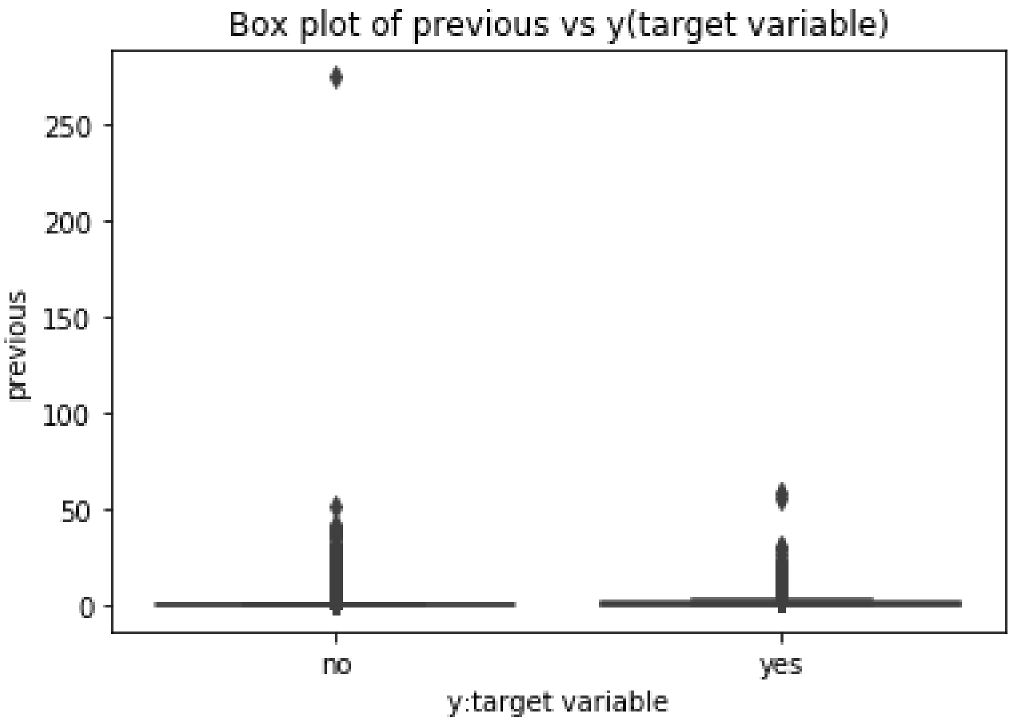
```
95% of calls have duration less than equal to 12.516666666666667
96% of calls have duration less than equal to 13.716666666666667
97% of calls have duration less than equal to 15.244999999999951
98% of calls have duration less than equal to 17.516666666666666
99% of calls have duration less than equal to 21.15
100% of calls have duration less than equal to 81.96666666666667
IQR 3.5999999999999996
```

Observation:
* The attribute duration seems to be important feature as there is a clear distinction in quartile ranges of duration for target variable yes and no. * 75% call duration are less than or equal to 5.32 * duration have a mean of 4.30 and standard-deviation 4.29 * There are outliers points in duration.

previous

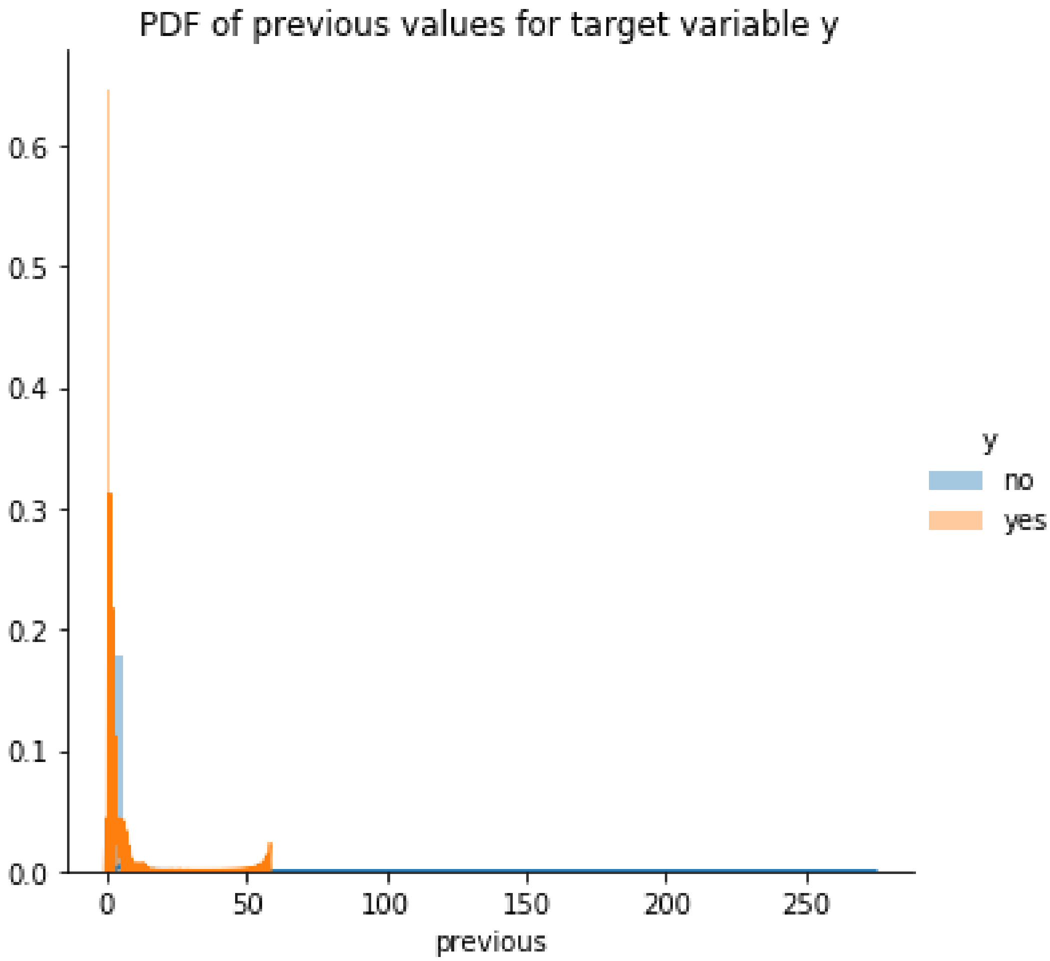
[20]:
```
sns.boxplot(y=data['previous'], x=data['y'])
plt.title('Box plot of previous vs y(target variable)')
plt.xlabel('y:target variable')
```

[20]:
```
Text(0.5, 0, 'y:target variable')
```



[21]:
```
sns.FacetGrid(data, hue='y', size=5) \
.map(sns.distplot, 'previous') \
.add_legend()
plt.title('PDF of previous values for target variable y')
```

[21]: Text(0.5, 1, 'PDF of previous values for target variable y')



[22]:
```
data.previous.describe()
```

[22]:
```
count    45211.000000
mean         0.580323
std          2.303441
min          0.000000
25%          0.000000
50%          0.000000
75%          0.000000
max        275.000000
Name: previous, dtype: float64
```

[23]:
```
for x in range(95, 101 , 1):
    print("{}% of previous values less than equal to {}".format(x, data.
 ↪previous.quantile(x/100)))
iqr = data.previous.quantile(0.75) - data.previous.quantile(0.25)
print('IQR {}'.format(iqr))
```

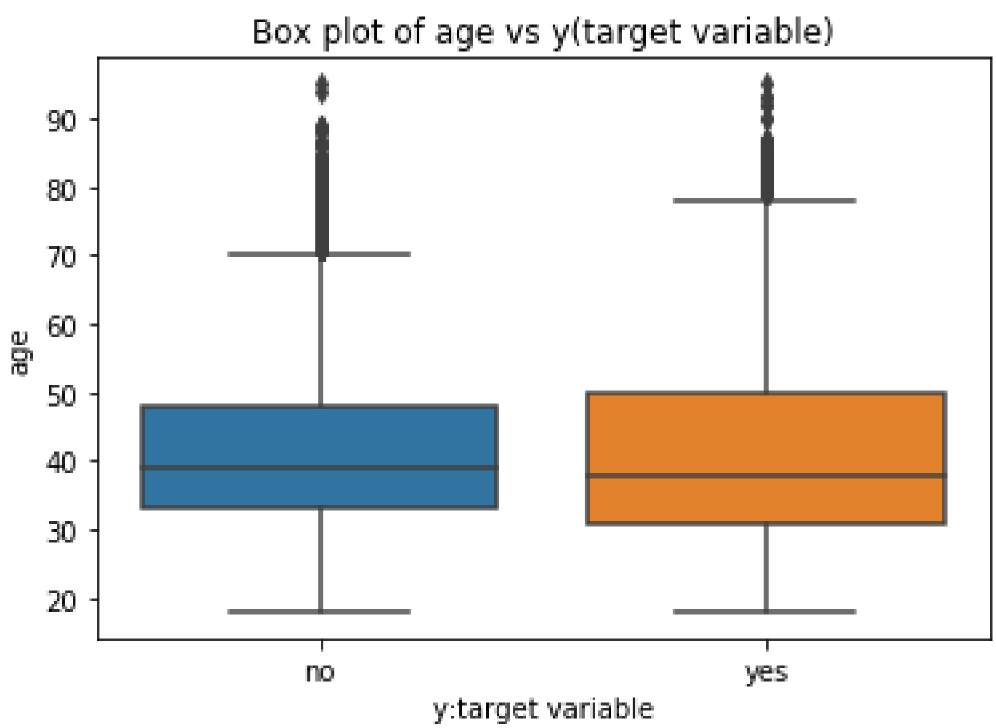
```
95% of previous values less than equal to 3.0
96% of previous values less than equal to 4.0
97% of previous values less than equal to 5.0
98% of previous values less than equal to 6.0
99% of previous values less than equal to 8.900000000001455
100% of previous values less than equal to 275.0
IQR 0.0
```

Observation:
* 75% of previous values equal 0 and 99% values <= 8.90 * duration have a mean of 0.58 and standard-deviation 2.30 * There are outliers points in duration.

age

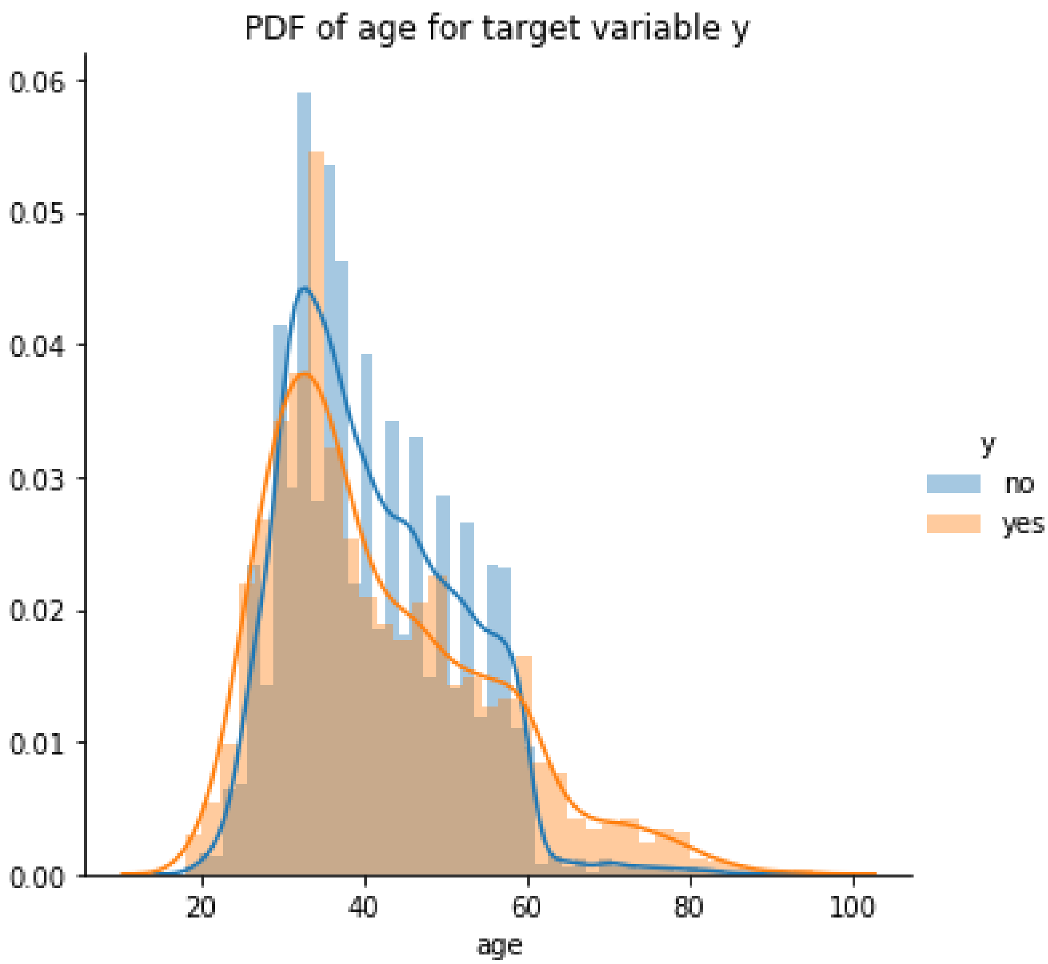
[24]:
```
sns.boxplot(y=data['age'], x=data['y'])
plt.title('Box plot of age vs y(target variable)')
plt.xlabel('y:target variable')
```

[24]:
```
Text(0.5, 0, 'y:target variable')
```



[25]:
```
sns.FacetGrid(data, hue='y', size=5) \
.map(sns.distplot, 'age') \
.add_legend()
plt.title('PDF of age for target variable y')
```

[25]: Text(0.5, 1, 'PDF of age for target variable y')



[26]:
```
data.age.describe()
```

[26]:
```
count    45211.000000
mean        40.936210
std         10.618762
min         18.000000
25%         33.000000
50%         39.000000
75%         48.000000
max         95.000000
Name: age, dtype: float64
```

[27]:
```
for x in range(95, 101 , 1):
    print("{}% of people having age are less than equal to {}".format(x, data.
 ↪age.quantile(x/100)))
```

```
iqr = data.age.quantile(0.75) - data.age.quantile(0.25)
print('IQR {}'.format(iqr))
```

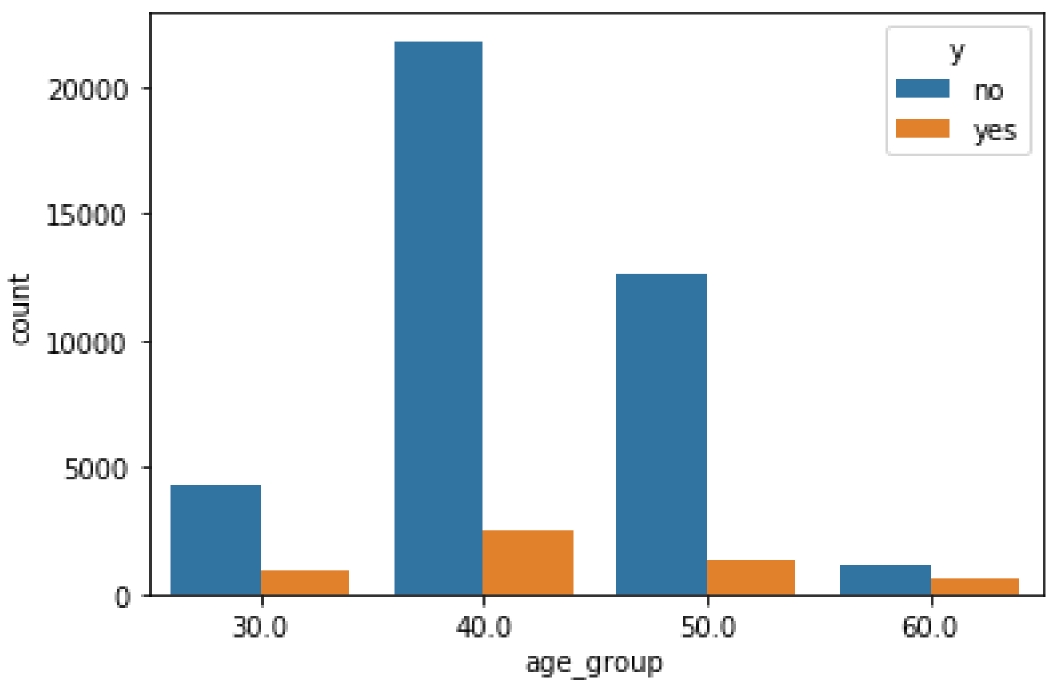
```
95% of people having age are less than equal to 59.0
96% of people having age are less than equal to 59.0
97% of people having age are less than equal to 60.0
98% of people having age are less than equal to 63.0
99% of people having age are less than equal to 71.0
100% of people having age are less than equal to 95.0
IQR 15.0
```

[28]:
```
lst = [data]
for column in lst:
    column.loc[column["age"] < 30,  'age_group'] = 30
    column.loc[(column["age"] >= 30) & (column["age"] <= 44), 'age_group'] = 40
    column.loc[(column["age"] >= 45) & (column["age"] <= 59), 'age_group'] = 50
    column.loc[column["age"] >= 60, 'age_group'] = 60
```

[29]:
```
count_age_response_pct = pd.crosstab(data['y'],data['age_group']).apply(lambda␣
 ↪x: x/x.sum() * 100)
count_age_response_pct = count_age_response_pct.transpose()
```

[30]:
```
sns.countplot(x='age_group', data=data, hue='y')
```

[30]: <matplotlib.axes._subplots.AxesSubplot at 0x7f2574681d68>



[31]:
```python
print('Success rate and total clients contacted for different age_groups:')
print('Clients age < 30 contacted: {}, Success rate: {}'.
 ↪format(len(data[data['age_group'] == 30]), data[data['age_group'] == 30].y.
 ↪value_counts()[1]/len(data[data['age_group'] == 30])))
print('Clients of age 30-45 contacted: {}, Success rate: {}'.
 ↪format(len(data[data['age_group'] == 40]), data[data['age_group'] == 40].y.
 ↪value_counts()[1]/len(data[data['age_group'] == 40])))
print('Clients of age 40-60 contacted: {}, Success rate: {}'.
 ↪format(len(data[data['age_group'] == 50]), data[data['age_group'] == 50].y.
 ↪value_counts()[1]/len(data[data['age_group'] == 50])))
print('Clients of 60+ age contacted: {}, Success rate: {}'.
 ↪format(len(data[data['age_group'] == 60]), data[data['age_group'] == 60].y.
 ↪value_counts()[1]/len(data[data['age_group'] == 60])))
```

```
Success rate and total clients contacted for different age_groups:
Clients age < 30 contacted: 5273, Success rate: 0.1759908970225678
Clients of age 30-45 contacted: 24274, Success rate: 0.10117821537447474
Clients of age 40-60 contacted: 13880, Success rate: 0.09402017291066282
Clients of 60+ age contacted: 1784, Success rate: 0.336322869955157
```

Observation:
* People with age $<$ 30 or 60+ have higher success rate.
* Only 3% of clients have age of 60+

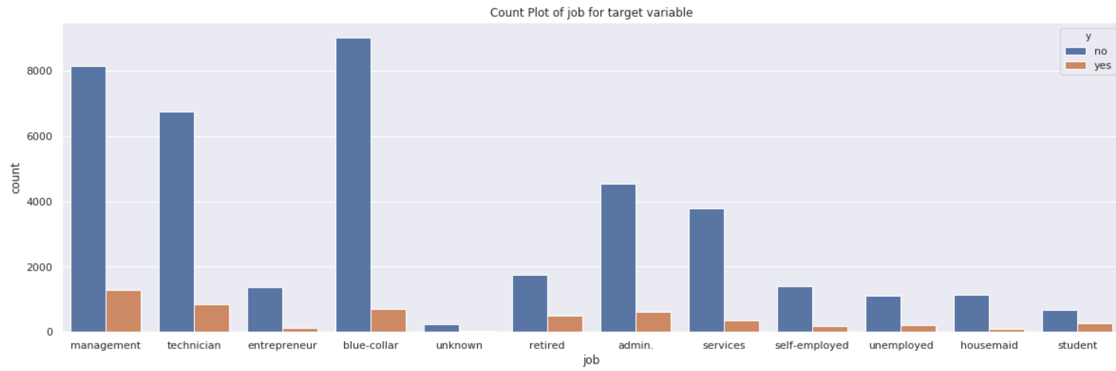
jobs

[32]:
```python
data.job.value_counts()
```

[32]:
```
blue-collar      9732
management       9458
technician       7597
admin.           5171
services         4154
retired          2264
self-employed    1579
entrepreneur     1487
unemployed       1303
housemaid        1240
student           938
unknown           288
Name: job, dtype: int64
```

[33]:
```python
sns.set(rc={'figure.figsize':(20,6)})
sns.countplot(x=data['job'], data=data, hue=data['y'])
plt.title('Count Plot of job for target variable')
```

[33]:
```
Text(0.5, 1.0, 'Count Plot of job for target variable')
```



[34]:

```
table = PrettyTable(['Job', 'Total Clients', 'Success rate'])
table.add_row(['Blue-collar', len(data[data['job'] == 'blue-collar']),␣
 ↪data[data['job'] == 'blue-collar'].y.value_counts()[1]/len(data[data['job']␣
 ↪== 'blue-collar'])])
table.add_row(['Management', len(data[data['job'] == 'management']),␣
 ↪data[data['job'] == 'management'].y.value_counts()[1]/len(data[data['job']␣
 ↪== 'management'])])
table.add_row(['Technician', len(data[data['job'] == 'technician']),␣
 ↪data[data['job'] == 'technician'].y.value_counts()[1]/len(data[data['job']␣
 ↪== 'technician'])])
table.add_row(['Admin', len(data[data['job'] == 'admin.']), data[data['job'] ==␣
 ↪'admin.'].y.value_counts()[1]/len(data[data['job'] == 'admin.'])])
table.add_row(['Services', len(data[data['job'] == 'services']),␣
 ↪data[data['job'] == 'services'].y.value_counts()[1]/len(data[data['job'] ==␣
 ↪'services'])])
table.add_row(['Retired', len(data[data['job'] == 'retired']), data[data['job']␣
 ↪== 'retired'].y.value_counts()[1]/len(data[data['job'] == 'retired'])])
table.add_row(['Self-employed', len(data[data['job'] == 'self-employed']),␣
 ↪data[data['job'] == 'self-employed'].y.value_counts()[1]/
 ↪len(data[data['job'] == 'self-employed'])])
table.add_row(['Entrepreneur', len(data[data['job'] == 'entrepreneur']),␣
 ↪data[data['job'] == 'entrepreneur'].y.value_counts()[1]/len(data[data['job']␣
 ↪== 'entrepreneur'])])
table.add_row(['Unemployed', len(data[data['job'] == 'unemployed']),␣
 ↪data[data['job'] == 'unemployed'].y.value_counts()[1]/len(data[data['job']␣
 ↪== 'unemployed'])])
table.add_row(['Housemaid', len(data[data['job'] == 'housemaid']),␣
 ↪data[data['job'] == 'housemaid'].y.value_counts()[1]/len(data[data['job'] ==␣
 ↪'housemaid'])])
table.add_row(['Student', len(data[data['job'] == 'student']), data[data['job']␣
 ↪== 'student'].y.value_counts()[1]/len(data[data['job'] == 'student'])])
table.add_row(['Unknown', len(data[data['job'] == 'unknown']), data[data['job']␣
 ↪== 'unknown'].y.value_counts()[1]/len(data[data['job'] == 'unknown'])])
```

```python
print(table)
```

```
+----------------+---------------+---------------------+
|      Job       | Total Clients |     Success rate    |
+----------------+---------------+---------------------+
|  Blue-collar   |      9732     | 0.07274969173859433 |
|   Management   |      9458     | 0.13755550856417847 |
|   Technician   |      7597     | 0.11056996182703699 |
|     Admin      |      5171     | 0.12202668729452718 |
|    Services    |      4154     | 0.08883004333172845 |
|    Retired     |      2264     | 0.22791519434628976 |
| Self-employed  |      1579     | 0.11842938568714376 |
|  Entrepreneur  |      1487     | 0.08271687962340282 |
|   Unemployed   |      1303     | 0.15502686108979277 |
|   Housemaid    |      1240     | 0.08790322580645162 |
|    Student     |      938      |  0.2867803837953092 |
|    Unknown     |      288      | 0.11805555555555555 |
+----------------+---------------+---------------------+
```

Observation:
* Top contacted clients are from job type: 'blue-collar', 'management' & 'technician' * Success rate is highest for student

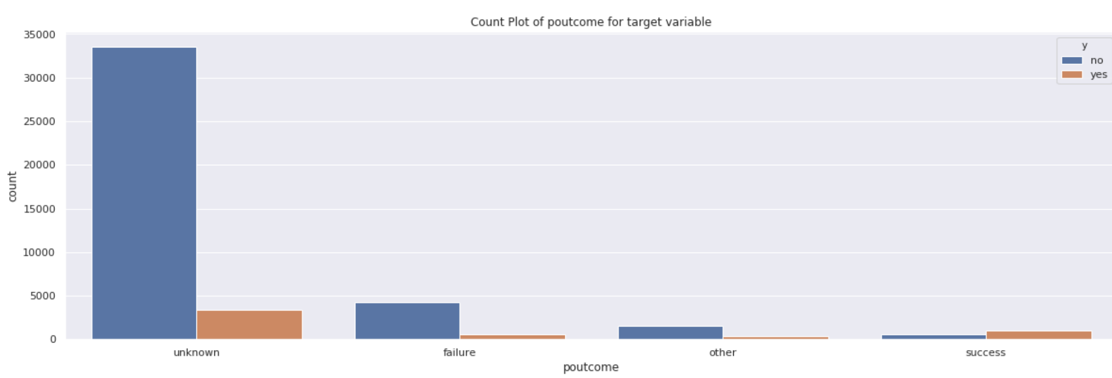
poutcome

[35]:
```python
data.poutcome.value_counts()
```

[35]:
```
unknown    36959
failure     4901
other       1840
success     1511
Name: poutcome, dtype: int64
```

[36]:
```python
sns.countplot(x=data['poutcome'], data=data, hue=data['y'])
plt.title('Count Plot of poutcome for target variable')
```

[36]:
```
Text(0.5, 1.0, 'Count Plot of poutcome for target variable')
```



Observation: * Most of the clients contacted have previous outcome as 'unknown'.

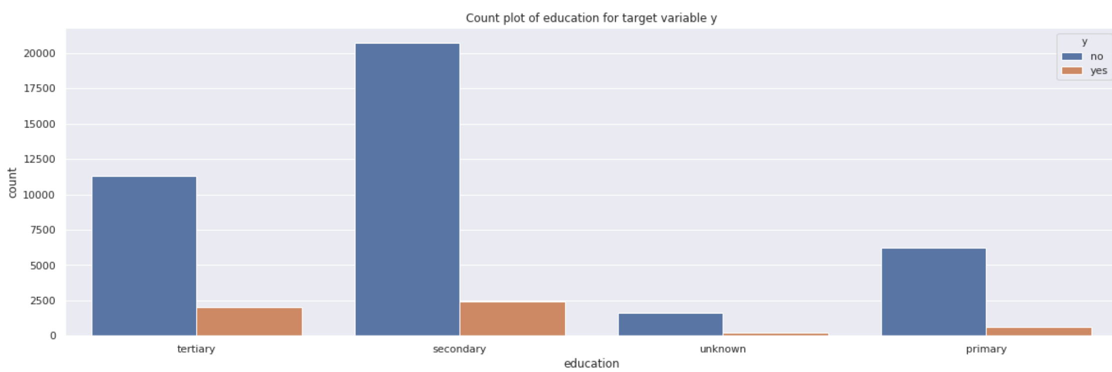
education

[37]:
```
data.education.value_counts()
```

[37]:
```
secondary    23202
tertiary     13301
primary       6851
unknown       1857
Name: education, dtype: int64
```

[38]:
```
sns.countplot(x=data['education'], data=data, hue=data['y'])
plt.title('Count plot of education for target variable y')
```

[38]:
```
Text(0.5, 1.0, 'Count plot of education for target variable y')
```



Observation: * Most of the people who are contacted have tertiray or secondary education.

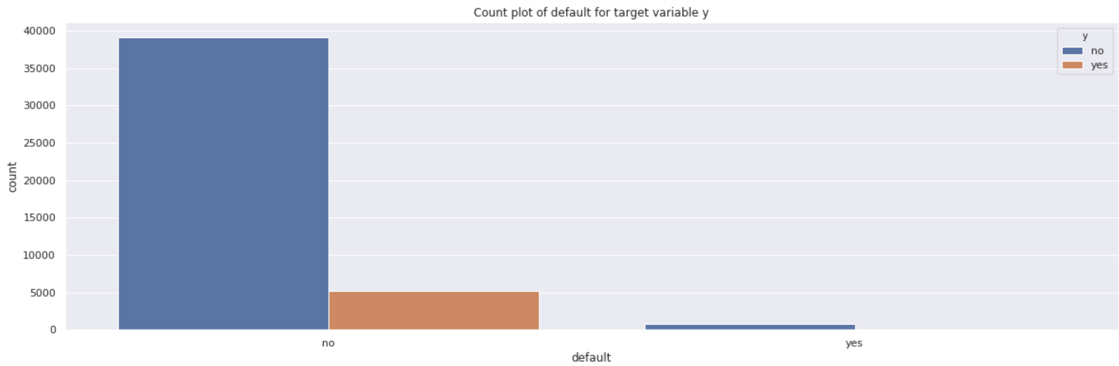
default

[39]:
```
data.default.value_counts()
```

[39]:
```
no     44396
yes      815
Name: default, dtype: int64
```

[40]:
```
sns.countplot(x=data['default'], data=data, hue=data['y'])
plt.title('Count plot of default for target variable y')
```

[40]:
```
Text(0.5, 1.0, 'Count plot of default for target variable y')
```



[41]:
```
data[data['default'] == 'yes'].y.count()
```

[41]: 815

Observation:
Very few clients are contacted who are defaulter,

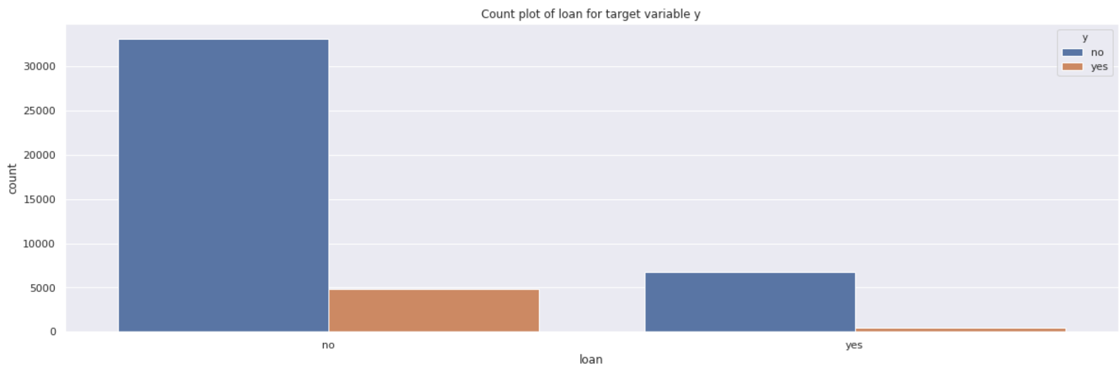
loan

[42]:
```
data.loan.value_counts()
```

[42]:
```
no     37967
yes     7244
Name: loan, dtype: int64
```

[43]:
```
sns.countplot(x=data['loan'], data=data, hue=data['y'])
plt.title('Count plot of loan for target variable y')
```

[43]: Text(0.5, 1.0, 'Count plot of loan for target variable y')



Observation:
* As seen for default variable, less client are contacted who have loan.

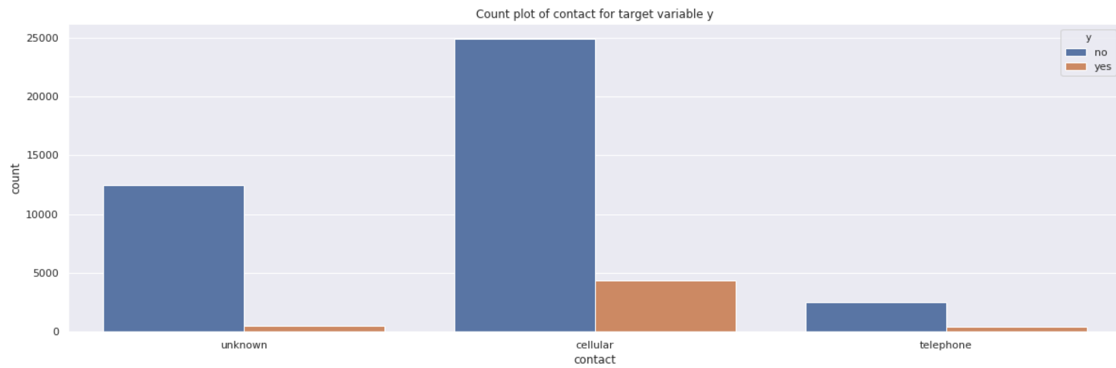
contact

[44]:
```
data.contact.value_counts()
```

[44]:
```
cellular     29285
unknown      13020
telephone     2906
Name: contact, dtype: int64
```

[45]:
```
sns.countplot(x=data['contact'], data=data, hue=data['y'])
plt.title('Count plot of contact for target variable y')
```

[45]:
```
Text(0.5, 1.0, 'Count plot of contact for target variable y')
```



Observation:
Most of the people are contacted through cellular

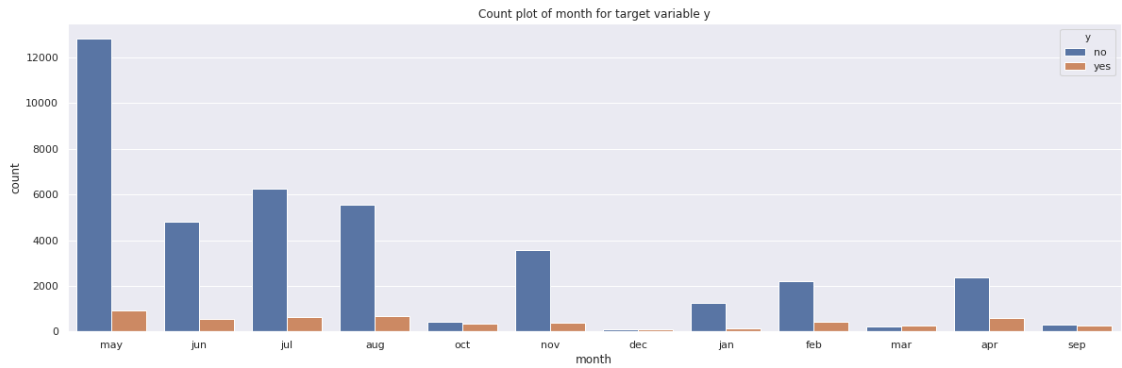
month

[46]:
```
data.month.value_counts()
```

[46]:
```
may    13766
jul     6895
aug     6247
jun     5341
nov     3970
apr     2932
feb     2649
jan     1403
oct      738
sep      579
mar      477
dec      214
Name: month, dtype: int64
```

[47]:
```
sns.countplot(x=data['month'], data=data, hue=data['y'])
plt.title('Count plot of month for target variable y')
```

[47]: Text(0.5, 1.0, 'Count plot of month for target variable y')



[48]:
```
data[data['month'] == 'jan'].y.value_counts()
```

[48]:
```
no     1261
yes     142
Name: y, dtype: int64
```

[49]:
```
print('Success rate and total clients contacted for different months:')
print('Clients contacted in January: {}, Success rate: {}'.
  ↪format(len(data[data['month'] == 'jan']), data[data['month'] == 'jan'].y.
  ↪value_counts()[1]/len(data[data['month'] == 'jan'])))
print('Clients contacted in February: {}, Success rate: {}'.
  ↪format(len(data[data['month'] == 'feb']), data[data['month'] == 'feb'].y.
  ↪value_counts()[1]/len(data[data['month'] == 'feb'])))
print('Clients contacted in March: {}, Success rate: {}'.
  ↪format(len(data[data['month'] == 'mar']), data[data['month'] == 'mar'].y.
  ↪value_counts()[1]/len(data[data['month'] == 'mar'])))
print('Clients contacted in April: {}, Success rate: {}'.
  ↪format(len(data[data['month'] == 'apr']), data[data['month'] == 'apr'].y.
  ↪value_counts()[1]/len(data[data['month'] == 'apr'])))
print('Clients contacted in May: {}, Success rate: {}'.
  ↪format(len(data[data['month'] == 'may']), data[data['month'] == 'may'].y.
  ↪value_counts()[1]/len(data[data['month'] == 'may'])))
print('Clients contacted in June: {}, Success rate: {}'.
  ↪format(len(data[data['month'] == 'jun']), data[data['month'] == 'jun'].y.
  ↪value_counts()[1]/len(data[data['month'] == 'jun'])))
print('Clients contacted in July: {}, Success rate: {}'.
  ↪format(len(data[data['month'] == 'jul']), data[data['month'] == 'jul'].y.
  ↪value_counts()[1]/len(data[data['month'] == 'jul'])))
```

```
print('Clients contacted in August: {}, Success rate: {}'.
    format(len(data[data['month'] == 'aug']), data[data['month'] == 'aug'].y.
    value_counts()[1]/len(data[data['month'] == 'aug'])))
print('Clients contacted in September: {}, Success rate: {}'.
    format(len(data[data['month'] == 'sep']), data[data['month'] == 'sep'].y.
    value_counts()[1]/len(data[data['month'] == 'sep'])))
print('Clients contacted in October: {}, Success rate: {}'.
    format(len(data[data['month'] == 'oct']), data[data['month'] == 'oct'].y.
    value_counts()[1]/len(data[data['month'] == 'oct'])))
print('Clients contacted in November: {}, Success rate: {}'.
    format(len(data[data['month'] == 'nov']), data[data['month'] == 'nov'].y.
    value_counts()[1]/len(data[data['month'] == 'nov'])))
print('Clients contacted in December: {}, Success rate: {}'.
    format(len(data[data['month'] == 'dec']), data[data['month'] == 'dec'].y.
    value_counts()[1]/len(data[data['month'] == 'dec'])))
```

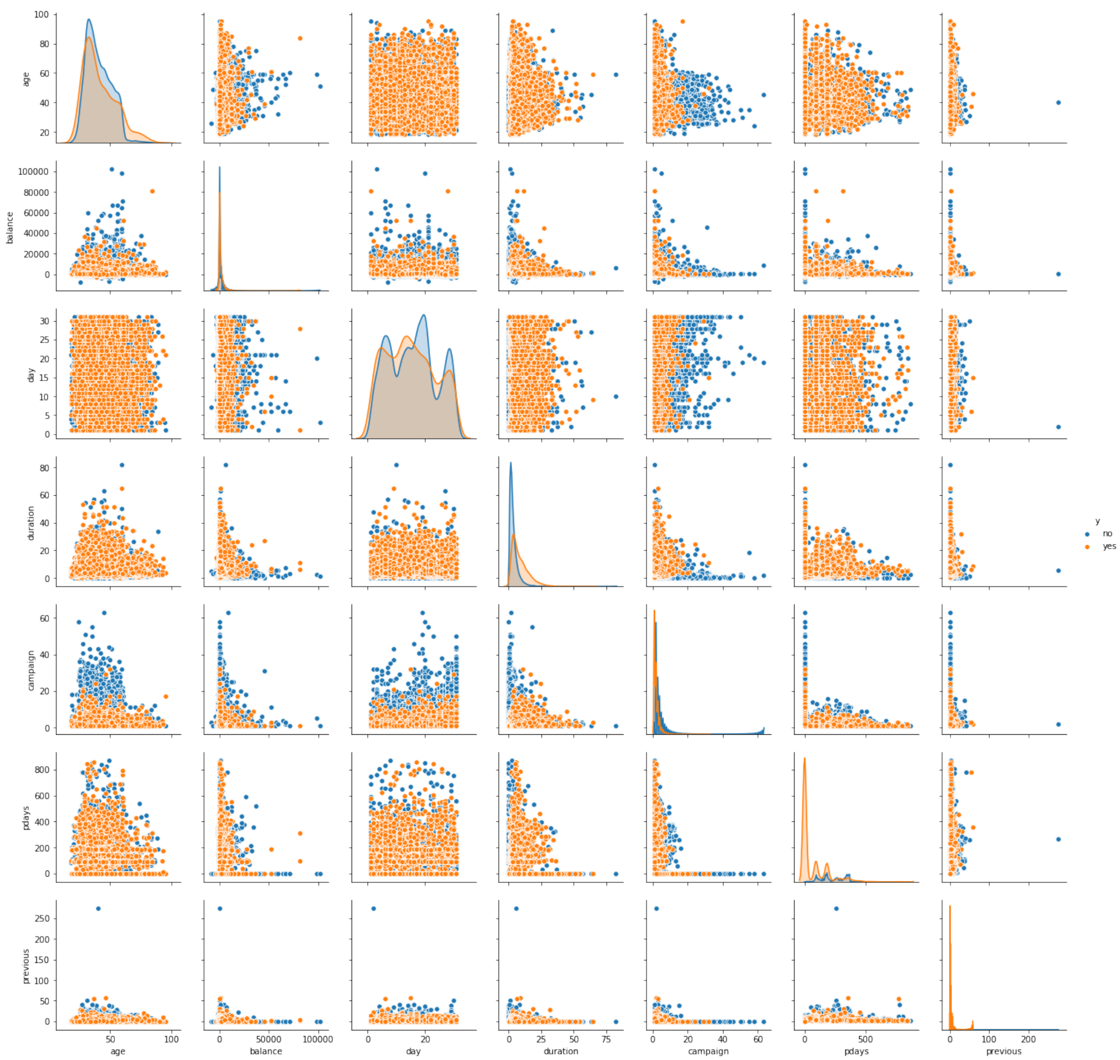
```
Success rate and total clients contacted for different months:
Clients contacted in January: 1403, Success rate: 0.10121168923734854
Clients contacted in February: 2649, Success rate: 0.1664779161947905
Clients contacted in March: 477, Success rate: 0.480083857442348
Clients contacted in April: 2932, Success rate: 0.19679399727148705
Clients contacted in May: 13766, Success rate: 0.06719453726572715
Clients contacted in June: 5341, Success rate: 0.10222804718217562
Clients contacted in July: 6895, Success rate: 0.09093546047860769
Clients contacted in August: 6247, Success rate: 0.11013286377461182
Clients contacted in September: 579, Success rate: 0.46459412780656306
Clients contacted in October: 738, Success rate: 0.43766937669376693
Clients contacted in November: 3970, Success rate: 0.10151133501259446
Clients contacted in December: 214, Success rate: 0.4672897196261682
```

Observation: * Most of the clients (approx 1/3 of total) are contacted in the month of May but the success rate is only 6.7%. * March have highest success rate.

Pairplot

[15]:
```
#data.drop('age_group', axis=1, inplace=True)
sns.pairplot(data, hue='y')
```

[15]: <seaborn.axisgrid.PairGrid at 0x7facfc4866a0>

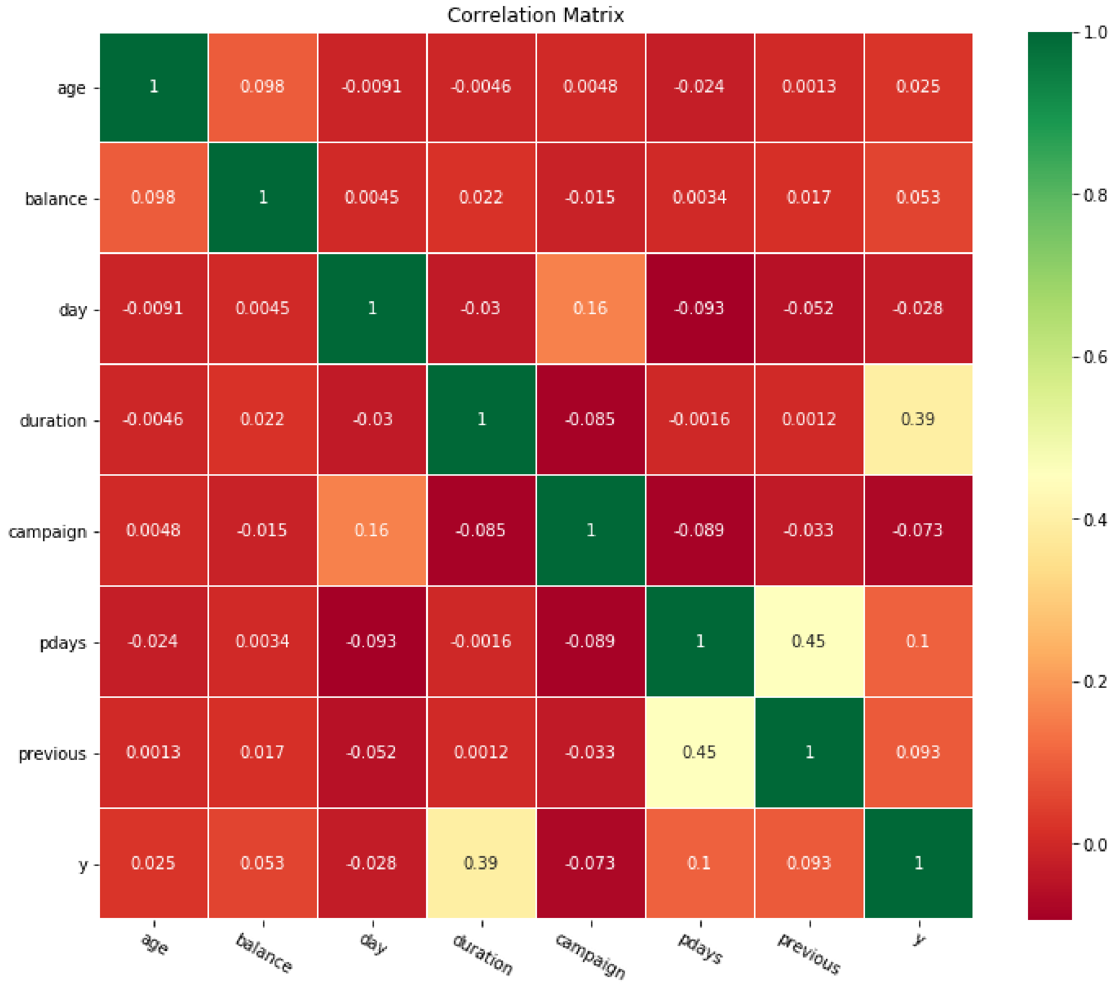


Observation:
* For most of the variables our pair plot is overlapping a lot. * Pair plots of age-campaign and day-campaign are much efficient in distinguishing between different classes with very few overlapes.

Correlation matrix of numerical features

[17]:
```
corr_data = data[numerical + ['y']]
corr = corr_data.corr()
plt.close()
cor_plot = sns.heatmap(corr,annot=True,cmap='RdYlGn',linewidths=0.
 ↪2,annot_kws={'size':10})
fig=plt.gcf()
fig.set_size_inches(12,10)
plt.xticks(fontsize=10,rotation=-30)
plt.yticks(fontsize=10)
```

```
plt.title('Correlation Matrix')
plt.show()
```



Observation:
* Over numerical features have very less correlation between them. * pdays and previous have higher correlation * duration have a higher correlation with our target variable

Outlier detection for numerical attributes using IQR

[18]:
```
# creating new data frame of numerical columns
data_numerical = data[numerical]
print('Shape of numerical dataframe {}'.format(data_numerical.shape))
data_numerical.head()
```

```
Shape of numerical dataframe (45211, 7)
```

[18]:
```
   age  balance  day  duration  campaign  pdays  previous
0   58     2143    5  4.350000         1     -1         0
1   44       29    5  2.516667         1     -1         0
2   33        2    5  1.266667         1     -1         0
3   47     1506    5  1.533333         1     -1         0
4   33        1    5  3.300000         1     -1         0
```

[19]:
```
q3 = data_numerical.quantile(0.75)
q1 = data_numerical.quantile(0.25)
iqr = q3 - q1
print('IQR for numerical attributes')
print(iqr)
```

```
IQR for numerical attributes
age           15.0
balance     1356.0
day           13.0
duration       3.6
campaign       2.0
pdays          0.0
previous       0.0
dtype: float64
```

[20]:
```
data_out = data[~((data_numerical < (q1 - 1.5 * iqr)) |(data_numerical > (q3 +␣
↪1.5 * iqr))).any(axis=1)]
print('{} points are outliers based on IQR'.format(data.shape[0] - data_out.
↪shape[0]))
```

```
17029 points are outliers based on IQR
```

[21]:
```
data.shape
```

[21]:
```
(45211, 17)
```

Preprocessing

Train Test Split

[3]:
```
data.replace(to_replace={'y':'yes'}, value=1, inplace=True)
data.replace(to_replace={'y':'no'}, value=0, inplace=True)
```

[22]:
```
# Convert the columns into categorical variables
data1 = data.copy()
data1['job'] = data1['job'].astype('category').cat.codes
data1['marital'] = data1['marital'].astype('category').cat.codes
data1['education'] = data1['education'].astype('category').cat.codes
data1['contact'] = data1['contact'].astype('category').cat.codes
data1['poutcome'] = data1['poutcome'].astype('category').cat.codes
data1['month'] = data1['month'].astype('category').cat.codes
```

```
data1['default'] = data1['default'].astype('category').cat.codes
data1['loan'] = data1['loan'].astype('category').cat.codes
data1['housing'] = data1['housing'].astype('category').cat.codes
```

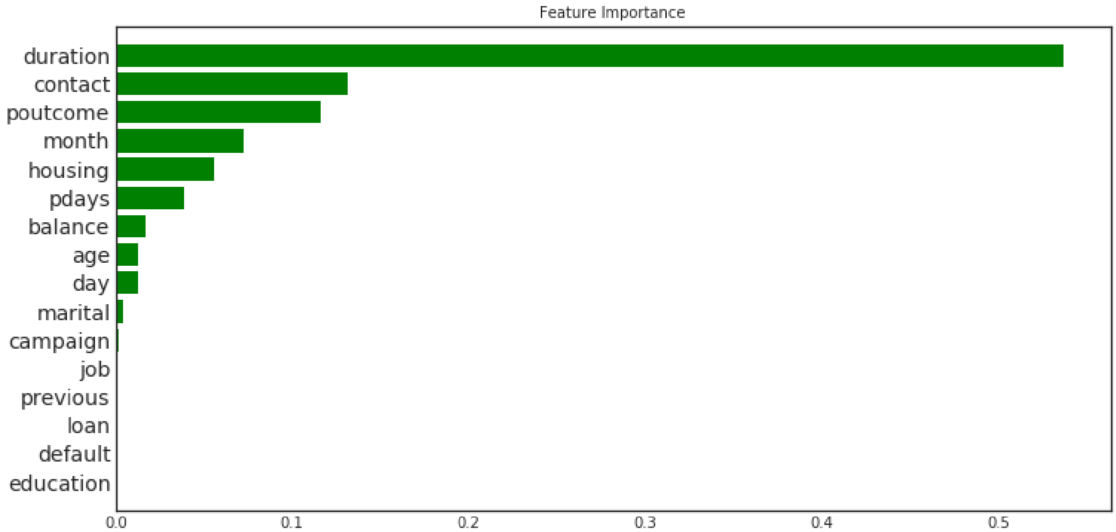
[4]:
```
y = data['y']
x_train, x_test, y_train, y_test = train_test_split(data.drop(['y'], axis=1),␣
↪y, test_size=0.20, random_state=42)
```

[24]:
```
print('Train data shape {} {}'.format(x_train.shape, y_train.shape))
print('Test data shape {} {}'.format(x_test.shape, y_test.shape))
```

```
Train data shape (36168, 16) (36168,)
Test data shape (9043, 16) (9043,)
```

Feature Importance

[25]:
```
plt.style.use('seaborn-white')

clf = DecisionTreeClassifier(class_weight='balanced', min_weight_fraction_leaf␣
↪= 0.01)

clf.fit(x_train, y_train)
importances = clf.feature_importances_
feature_names = data.drop('y', axis=1).columns
indices = np.argsort(importances)

def feature_importance_graph(indices, importances, feature_names):
    plt.figure(figsize=(12,6))
    plt.title("Feature Importance", fontsize=10)
    plt.barh(range(len(indices)), importances[indices], color='g', ␣
↪align="center")
    plt.yticks(range(len(indices)), feature_names[indices],␣
↪rotation='horizontal',fontsize=14)
    plt.ylim([-1, len(indices)])

feature_importance_graph(indices, importances, feature_names)
plt.show()
```



Important features we are going to consider for machine learning models:
* duration
* contact
* poutcome
* month
* housing
* pdays * age
* balance

Encoding data

Encoding categories

[5]:
```
vectorizer = CountVectorizer(vocabulary=x_train.poutcome.unique())
x_train_poutcome = vectorizer.fit_transform(x_train.poutcome)
x_test_poutcome = vectorizer.transform(x_test.poutcome)
```

[6]:
```
vectorizer = CountVectorizer(vocabulary=x_train.contact.unique())
x_train_contact = vectorizer.fit_transform(x_train.contact)
x_test_contact = vectorizer.transform(x_test.contact)
```

[7]:
```
vectorizer = CountVectorizer(vocabulary=x_train.month.unique())
x_train_month = vectorizer.fit_transform(x_train.month)
x_test_month = vectorizer.transform(x_test.month)
```

[8]:
```
vectorizer = CountVectorizer(vocabulary=x_train.housing.unique())
x_train_housing = vectorizer.fit_transform(x_train.housing)
x_test_housing = vectorizer.transform(x_test.housing)
```

Encoding Numerical data using Normalizer()

[9]:
```
vectorizer = Normalizer()
x_train_duration = vectorizer.fit_transform(x_train.duration.values.
↪reshape(1,-1)).transpose()
x_test_duration = vectorizer.transform(x_test.duration.values.reshape(1, -1)).
↪transpose()
```

[10]:
```
vectorizer = Normalizer()
x_train_pdays = vectorizer.fit_transform(x_train.pdays.values.reshape(1,-1)).
↪transpose()
x_test_pdays = vectorizer.transform(x_test.pdays.values.reshape(1, -1)).
↪transpose()
```

[11]:
```
vectorizer = Normalizer()
x_train_age = vectorizer.fit_transform(x_train.age.values.reshape(1,-1)).
↪transpose()
x_test_age = vectorizer.transform(x_test.age.values.reshape(1, -1)).transpose()
```

[12]:
```
vectorizer = Normalizer()
x_train_balance = vectorizer.fit_transform(x_train.balance.values.
↪reshape(1,-1)).transpose()
x_test_balance = vectorizer.transform(x_test.balance.values.reshape(1, -1)).
↪transpose()
```

[13]:
```
from scipy.sparse import hstack

train = hstack((x_train_contact, x_train_poutcome, x_train_month,␣
↪x_train_housing, x_train_duration, x_train_pdays, x_train_age,␣
↪x_train_balance)).tocsr()

test = hstack((x_test_contact, x_test_poutcome, x_test_month, x_test_housing,␣
↪x_test_duration, x_test_pdays, x_test_age, x_test_balance)).tocsr()
```

Machine Learning Models

[15]:
```
# dictionary to store accuracy and roc score for each model
score = {}
```

Logistic Regression

Hyperparameter tuning Logistic Regression

[37]:
```
parameters = {'C':[(10**i)*x for i in range(-4, 1) for x in [1,3,5]]}

model = LogisticRegression(class_weight='balanced')
clf = RandomizedSearchCV(model, parameters, cv=5, scoring='roc_auc',␣
↪return_train_score=True, n_jobs=-1)
clf.fit(train, y_train)
print('Best parameters:  {}'.format(clf.best_params_))
```

```python
print('Best score: {}'.format(clf.best_score_))
```

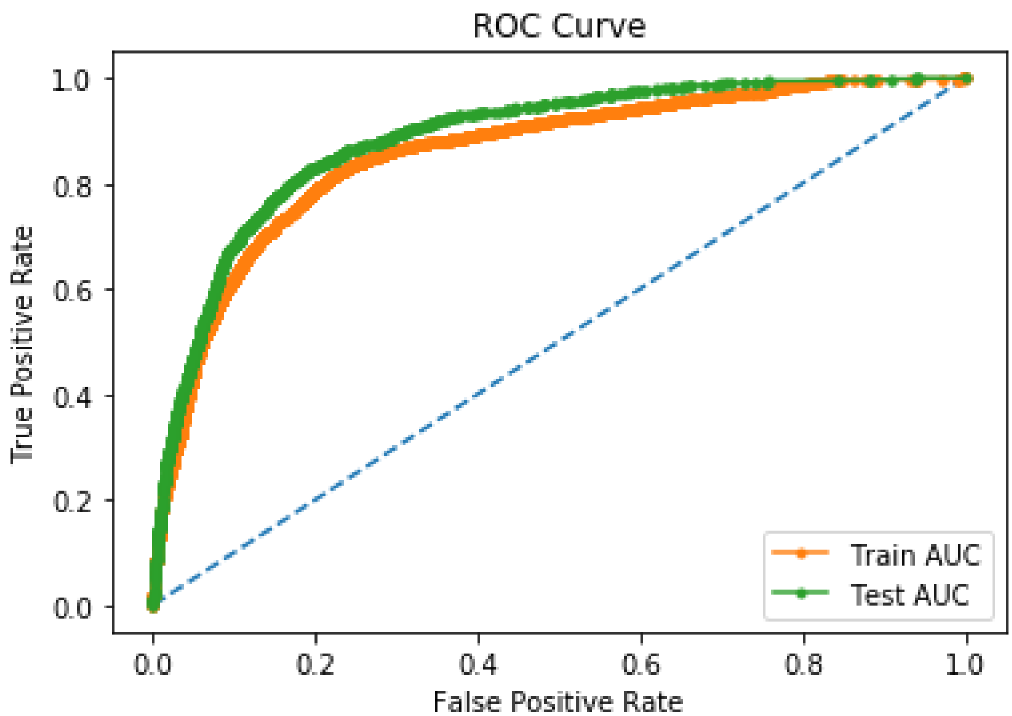
```
Best parameters:  {'C': 3}
Best score: 0.8496734277981354
```

Training Logistic Regression with best hyperparameters

[16]:
```python
from sklearn.metrics import log_loss

model = LogisticRegression(C=3, class_weight='balanced', n_jobs=-1)
model.fit(train, y_train)
y_probs_train = model.predict_proba(train)
y_probs_test = model.predict_proba(test)
y_predicted_train = model.predict(train)
y_predicted_test = model.predict(test)

# keep probabilities for the positive outcome only
y_probs_train = y_probs_train[:, 1]
y_probs_test = y_probs_test[:, 1]

# calculate AUC and Accuracy
train_auc = roc_auc_score(y_train, y_probs_train)
test_auc = roc_auc_score(y_test, y_probs_test)
train_acc = accuracy_score(y_train, y_predicted_train)
test_acc = accuracy_score(y_test, y_predicted_test)
print('*'*50)
print('Train AUC: %.3f' % train_auc)
print('Test AUC: %.3f' % test_auc)
print('*'*50)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % test_acc)

score['Logistic Regression'] = [test_auc, test_acc]

# calculate roc curve
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_probs_train)
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_probs_test)
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(train_fpr, train_tpr, marker='.', label='Train AUC')
plt.plot(test_fpr, test_tpr, marker='.', label='Test AUC')
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.show()
```

```
**************************************************
Train AUC: 0.859
```

```
Test AUC: 0.887
**************************************************
Train Accuracy: 0.833
Test Accuracy: 0.746
```

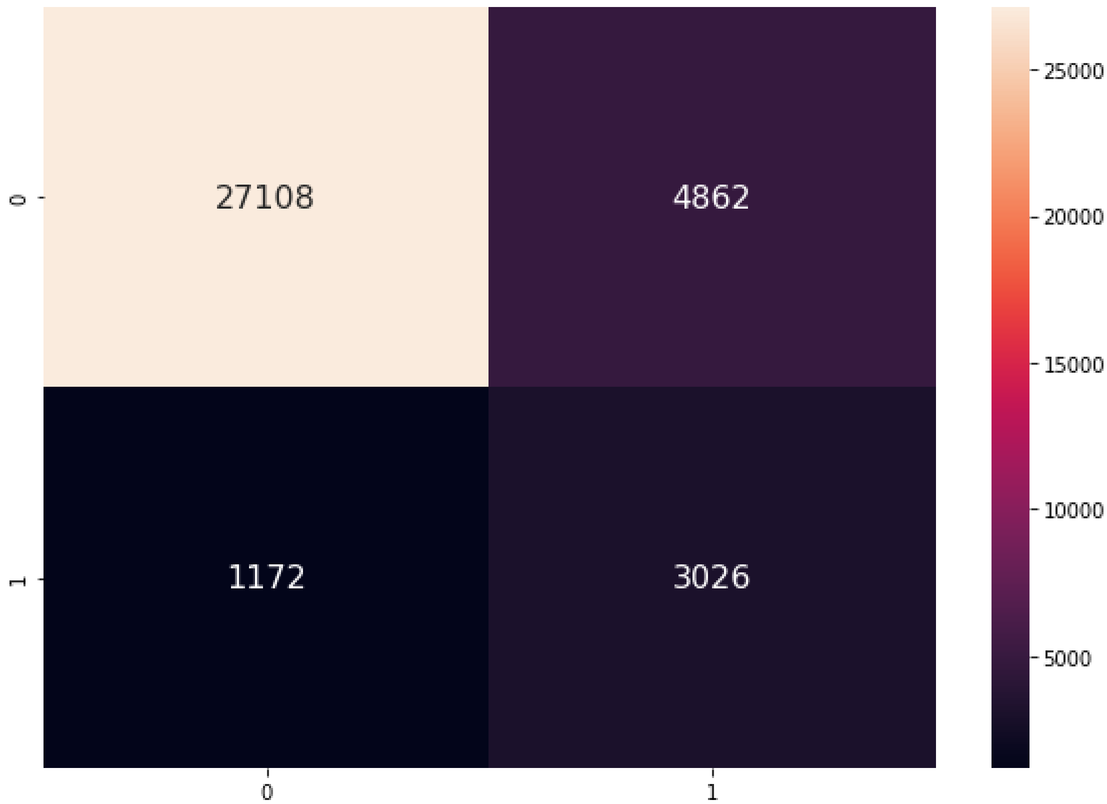


Train Confusion Matrix

[19]:
```python
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_train, y_predicted_train)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[27108  4862]
 [ 1172  3026]]
```

[19]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3acc186438>
```

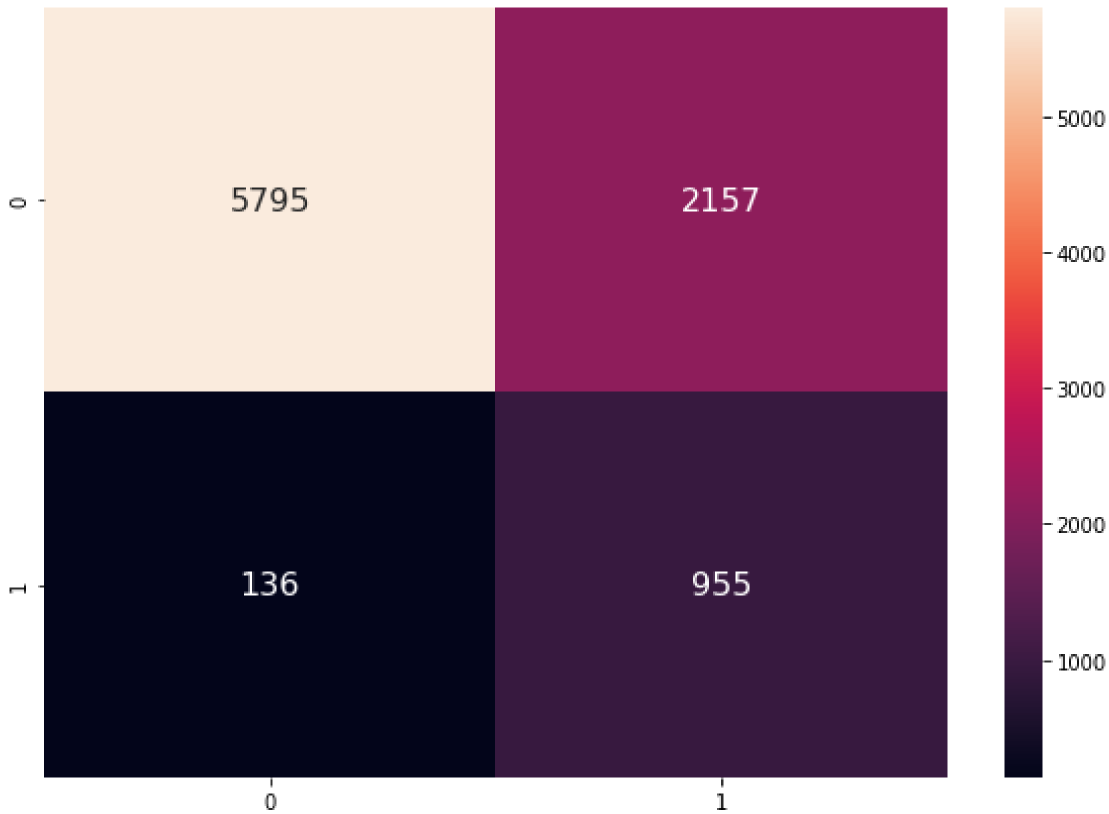


Test Confusion Matrix

[20]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_test, y_predicted_test)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[5795 2157]
 [ 136  955]]
```

[20]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3acc4a3ba8>



Random Forest

Hyperparameter tuning Random Forest

[40]:
```
params = {'n_estimators':[75, 100, 250, 500], 'max_depth':[3, 5, 10, 15, 25]}
model = RandomForestClassifier(class_weight='balanced', n_jobs=-1)
clf = RandomizedSearchCV(model, param_distributions=params, cv=5,␣
 ↪scoring='roc_auc', random_state=42, n_jobs=-1, return_train_score=True)
clf.fit(train, y_train)
print('Best parameters:  {}'.format(clf.best_params_))
print('Best score: {}'.format(clf.best_score_))
```

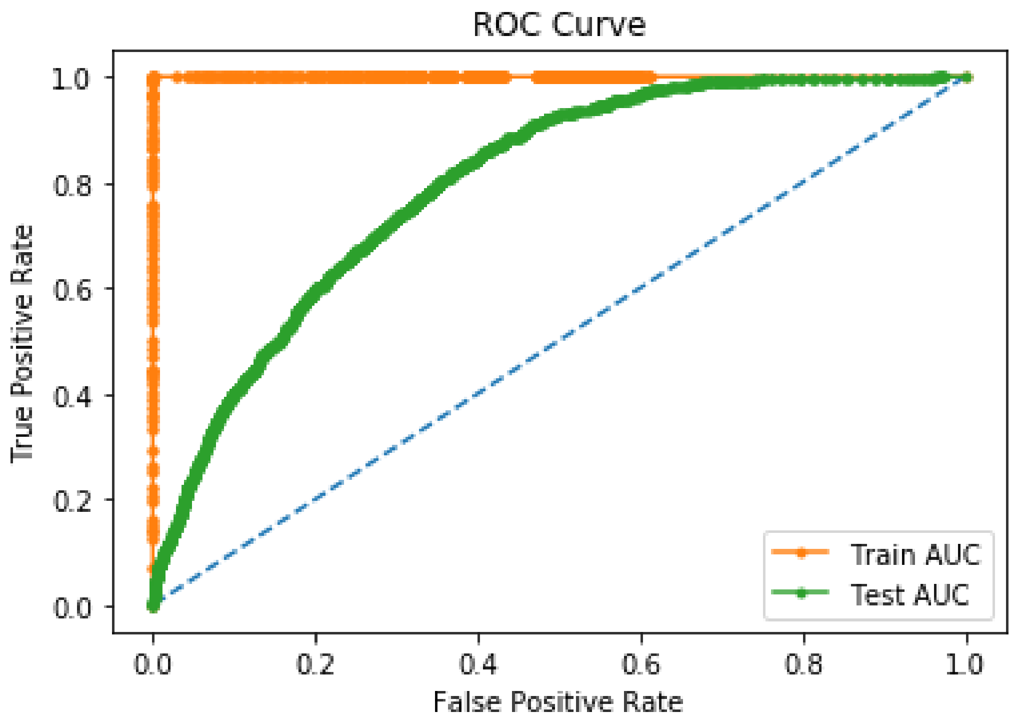
```
Best parameters:  {'n_estimators': 250, 'max_depth': 25}
Best score: 0.918424480262767
```

Training random forest with best hyperparameters

[21]:
```
model = RandomForestClassifier(n_estimators=250, max_depth=25,␣
 ↪class_weight='balanced', n_jobs=-1)
model.fit(train, y_train)
y_probs_train = model.predict_proba(train)
y_probs_test = model.predict_proba(test)
y_predicted_train = model.predict(train)
```

```
y_predicted_test = model.predict(test)

# keep probabilities for the positive outcome only
y_probs_train = y_probs_train[:, 1]
y_probs_test = y_probs_test[:, 1]

# calculate AUC and Accuracy
train_auc = roc_auc_score(y_train, y_probs_train)
test_auc = roc_auc_score(y_test, y_probs_test)
train_acc = accuracy_score(y_train, y_predicted_train)
test_acc = accuracy_score(y_test, y_predicted_test)
print('*'*50)
print('Train AUC: %.3f' % train_auc)
print('Test AUC: %.3f' % test_auc)
print('*'*50)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % test_acc)

score['Random Forest'] = [test_auc, test_acc]

# calculate roc curve
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_probs_train)
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_probs_test)
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(train_fpr, train_tpr, marker='.', label='Train AUC')
plt.plot(test_fpr, test_tpr, marker='.', label='Test AUC')
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.show()
```

```
**************************************************
Train AUC: 1.000
Test AUC: 0.799
**************************************************
Train Accuracy: 0.999
Test Accuracy: 0.812
```

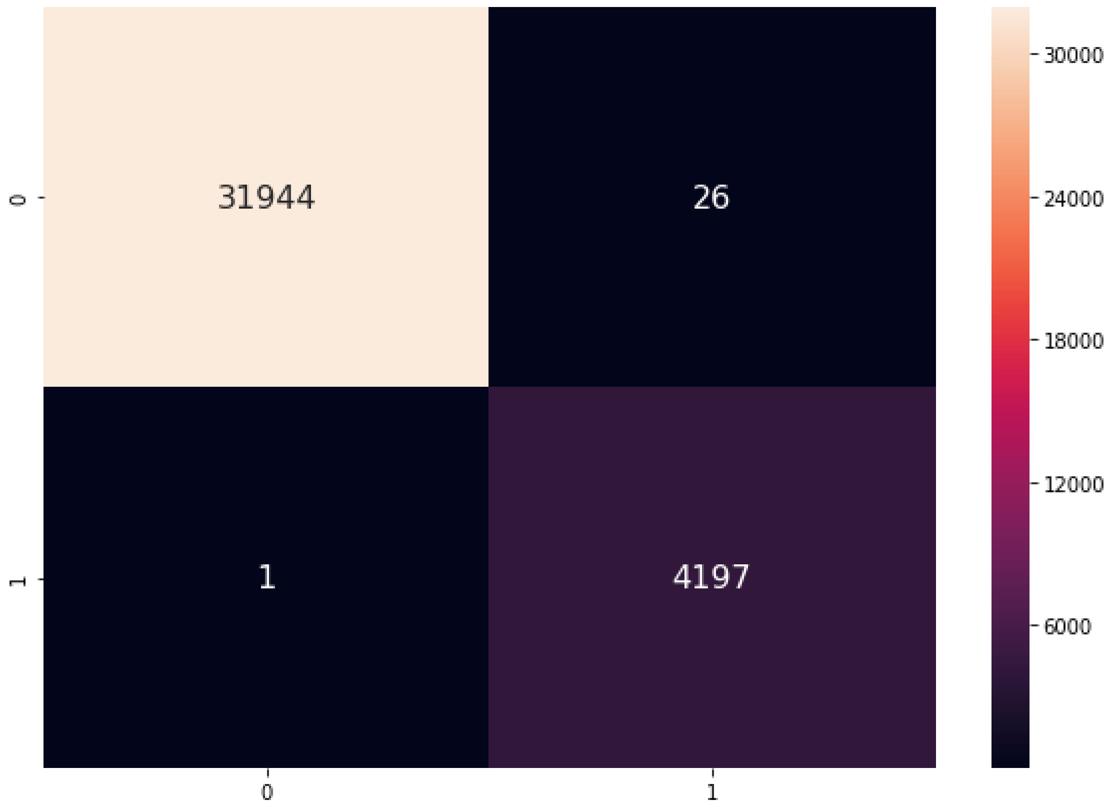


Train Confusion Matrix

[22]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_train, y_predicted_train)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[31944    26]
 [    1  4197]]
```

[22]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3acc1a0ef0>
```

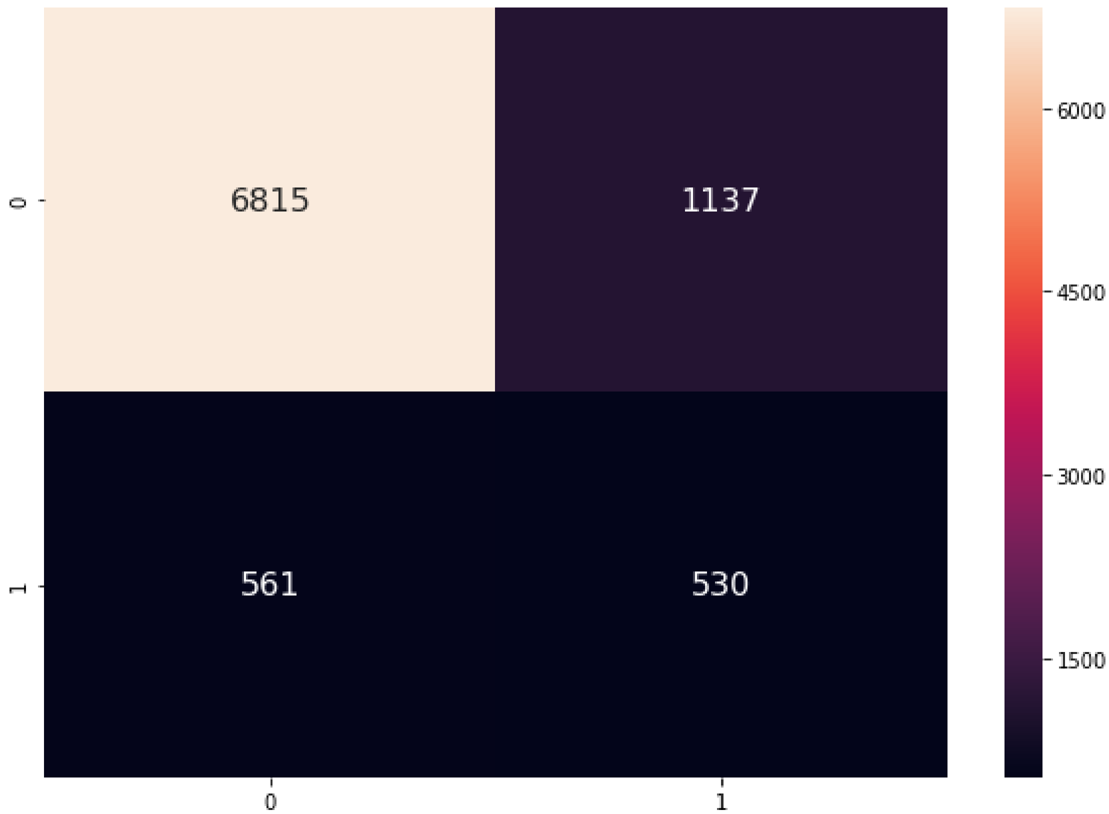


Test Confusion Matrix

[24]:
```python
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_test, y_predicted_test)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[6815 1137]
 [ 561  530]]
```

[24]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3abc9e4cf8>
```



SVM

Hyperparameter tuning SVM

[43]:
```
params = {'alpha': [10**i for i in range(-4, 5)]}

model = SGDClassifier(class_weight='balanced', n_jobs=-1)
clf = RandomizedSearchCV(model, param_distributions=params, cv=5,␣
 ↪scoring='roc_auc', random_state=42, n_jobs=-1, return_train_score=True)
clf.fit(train, y_train)
print('Best parameters:  {}'.format(clf.best_params_))
print('Best score: {}'.format(clf.best_score_))
```

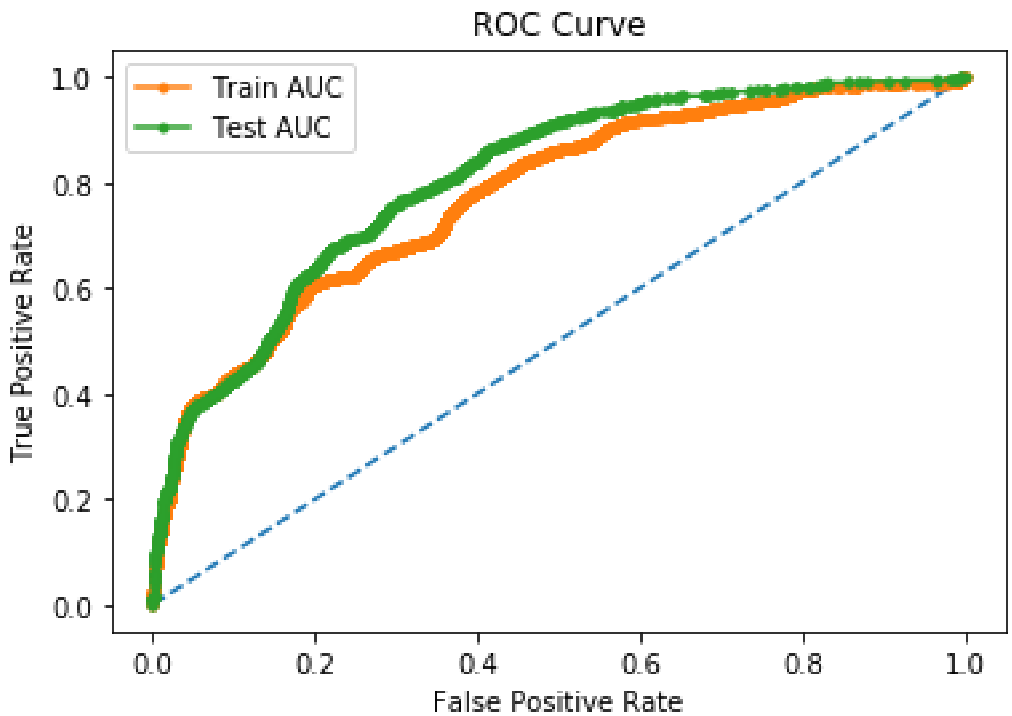
```
Best parameters:  {'alpha': 0.0001}
Best score: 0.7767774627832619
```

Training SVM with best hyperparameters

[25]:
```
model = SGDClassifier(alpha=0.0001, class_weight='balanced', n_jobs=-1)
model.fit(train, y_train)
y_probs_train = model.decision_function(train)
y_probs_test = model.decision_function(test)
y_predicted_train = model.predict(train)
```

```python
y_predicted_test = model.predict(test)

# calculate AUC and Accuracy
train_auc = roc_auc_score(y_train, y_probs_train)
test_auc = roc_auc_score(y_test, y_probs_test)
train_acc = accuracy_score(y_train, y_predicted_train)
test_acc = accuracy_score(y_test, y_predicted_test)
print('*'*50)
print('Train AUC: %.3f' % train_auc)
print('Test AUC: %.3f' % test_auc)
print('*'*50)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % test_acc)

score['SVM'] = [test_auc, test_acc]

# calculate roc curve
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_probs_train)
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_probs_test)
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(train_fpr, train_tpr, marker='.', label='Train AUC')
plt.plot(test_fpr, test_tpr, marker='.', label='Test AUC')
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.show()
```

```
**************************************************
Train AUC: 0.776
Test AUC: 0.808
**************************************************
Train Accuracy: 0.809
Test Accuracy: 0.805
```

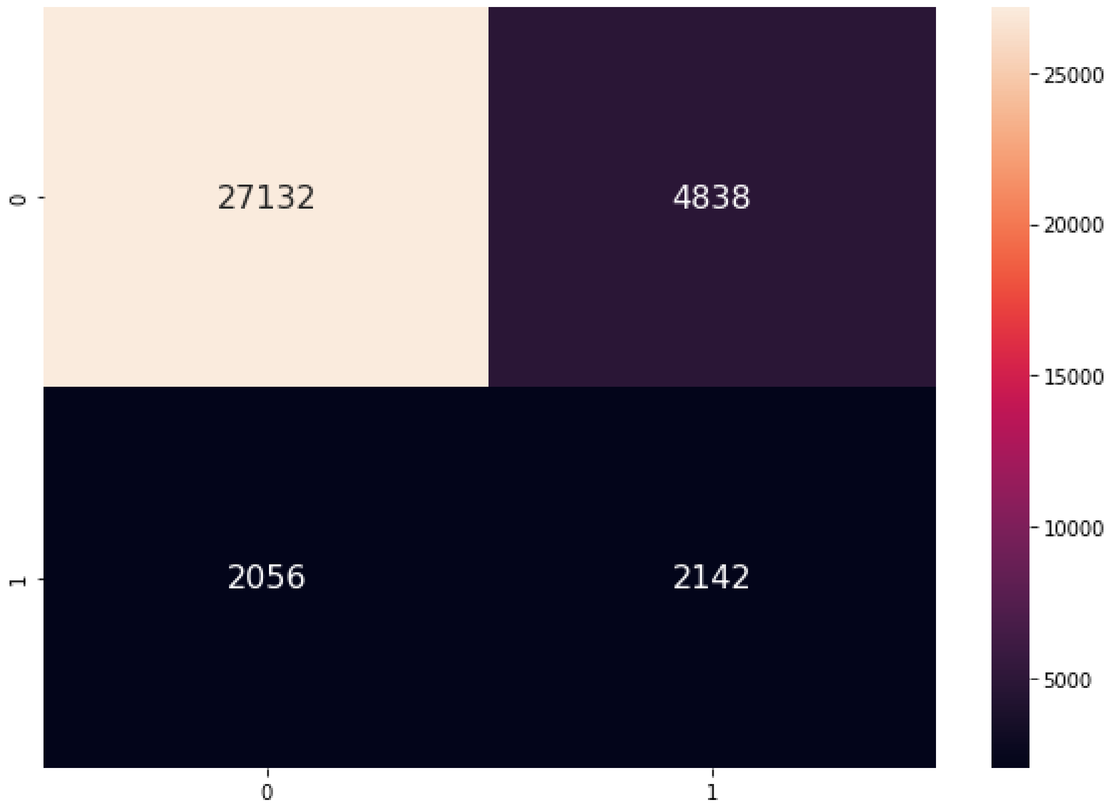


Train Confusion Matrix

[26]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_train, y_predicted_train)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[27132  4838]
 [ 2056  2142]]
```

[26]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3abc909a90>
```

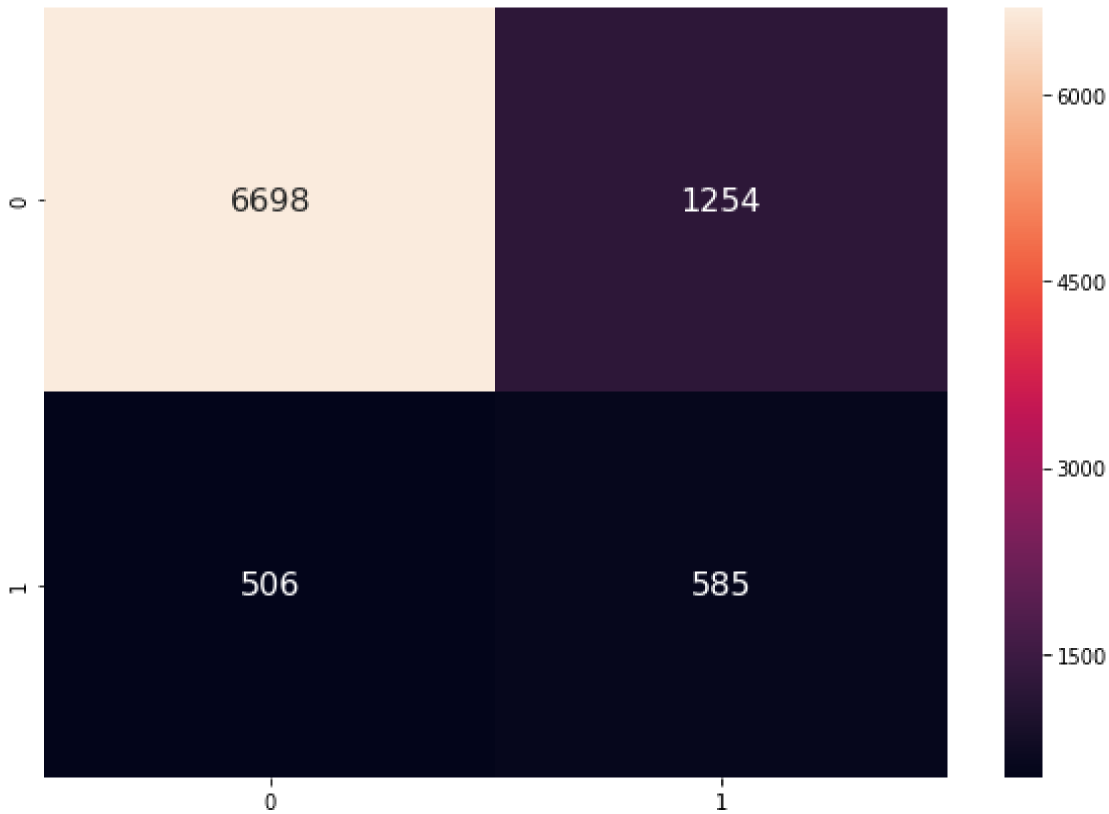


Test Confusion Matrix

[27]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_test, y_predicted_test)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[6698 1254]
 [ 506  585]]
```

[27]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3abc83ac18>
```



XGBoost

Hyperparameter tuning XGBClassifier

[45]:
```
from xgboost import XGBClassifier

params = {'max_depth': [5, 10, 15], 'n_estimators': [10, 100, 500]}

model = XGBClassifier(class_weight='balanced', n_jobs=-1)
clf = RandomizedSearchCV(model, param_distributions=params, cv=5,␣
 ↪scoring='roc_auc', random_state=42, n_jobs=-1, return_train_score=True)
clf.fit(train, y_train)
print('Best parameters:  {}'.format(clf.best_params_))
print('Best score: {}'.format(clf.best_score_))
```

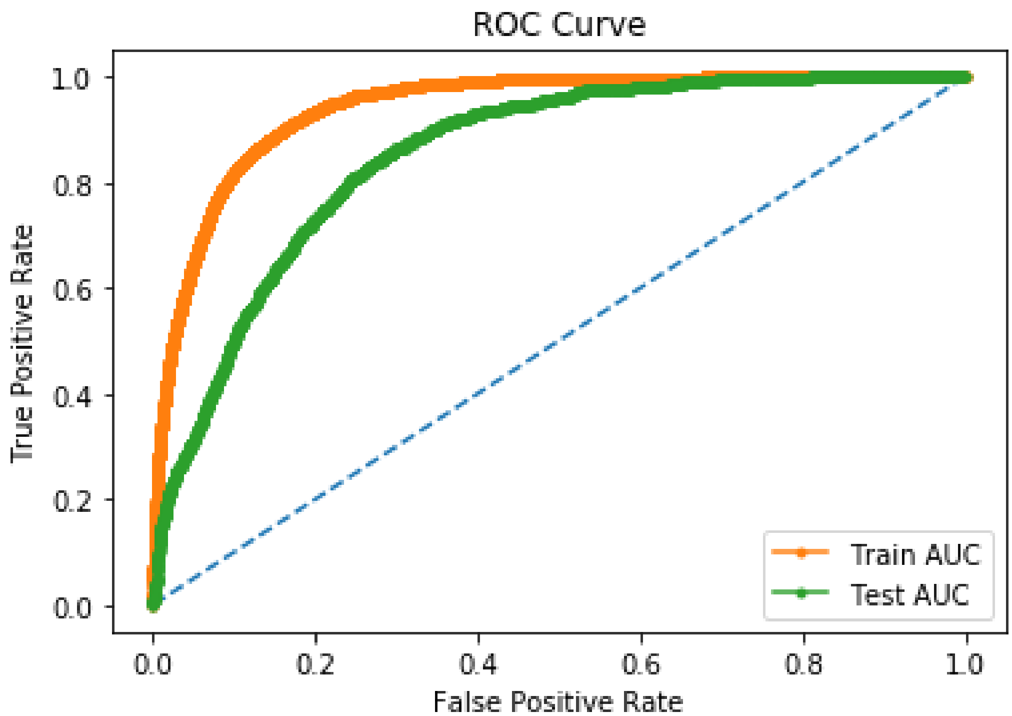
```
Best parameters:  {'n_estimators': 100, 'max_depth': 5}
Best score: 0.924594961178063
```

Training XGBClassifier with best hyperparameters

[29]:
```
from xgboost import XGBClassifier
```

```python
model = XGBClassifier(max_depth=5, n_estimators=100 ,class_weight='balanced',␣
 ↪n_jobs=-1)
model.fit(train, y_train)
y_probs_train = model.predict_proba(train)
y_probs_test = model.predict_proba(test)
y_predicted_train = model.predict(train)
y_predicted_test = model.predict(test)

# keep probabilities for the positive outcome only
y_probs_train = y_probs_train[:, 1]
y_probs_test = y_probs_test[:, 1]

# calculate AUC and Accuracy
train_auc = roc_auc_score(y_train, y_probs_train)
test_auc = roc_auc_score(y_test, y_probs_test)
train_acc = accuracy_score(y_train, y_predicted_train)
test_acc = accuracy_score(y_test, y_predicted_test)
print('*'*50)
print('Train AUC: %.3f' % train_auc)
print('Test AUC: %.3f' % test_auc)
print('*'*50)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % test_acc)

score['XGBoost'] = [test_auc, test_acc]

# calculate roc curve
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_probs_train)
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_probs_test)
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(train_fpr, train_tpr, marker='.', label='Train AUC')
plt.plot(test_fpr, test_tpr, marker='.', label='Test AUC')
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.show()
```

```
**************************************************
Train AUC: 0.942
Test AUC: 0.854
**************************************************
Train Accuracy: 0.920
Test Accuracy: 0.785
```

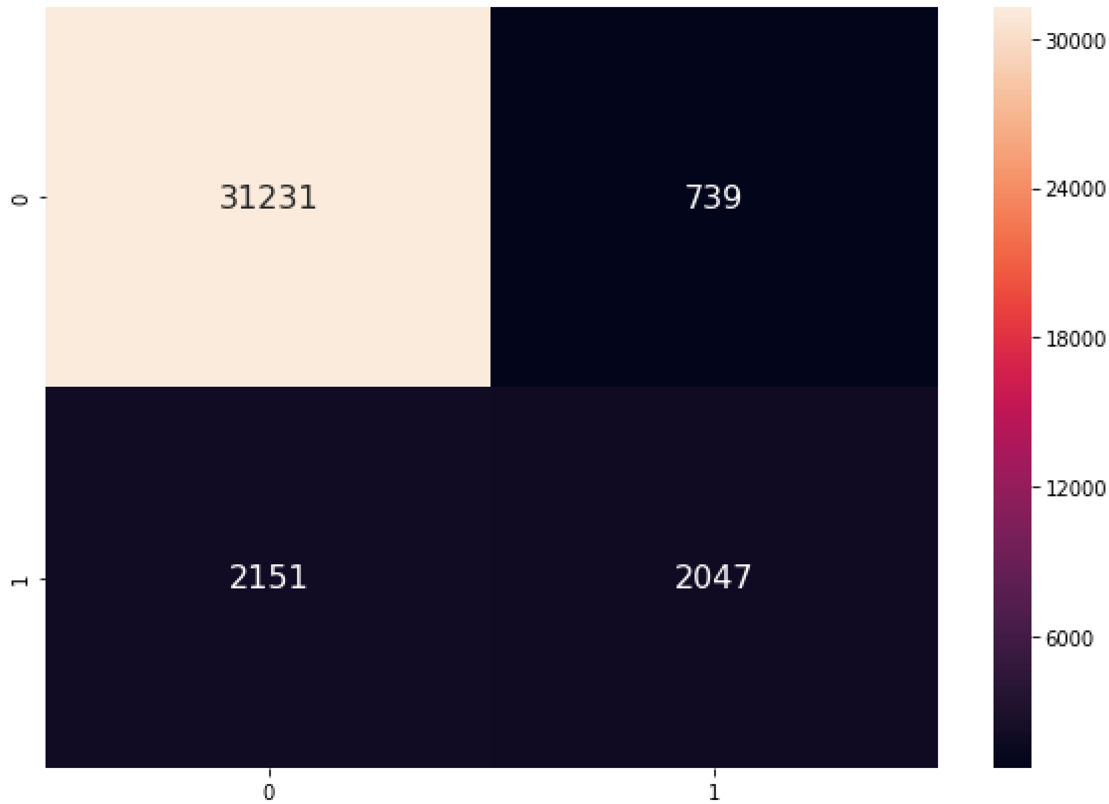


Train Confusion Matrix

[30]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_train, y_predicted_train)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[31231   739]
 [ 2151  2047]]
```

[30]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3acc6eebe0>
```

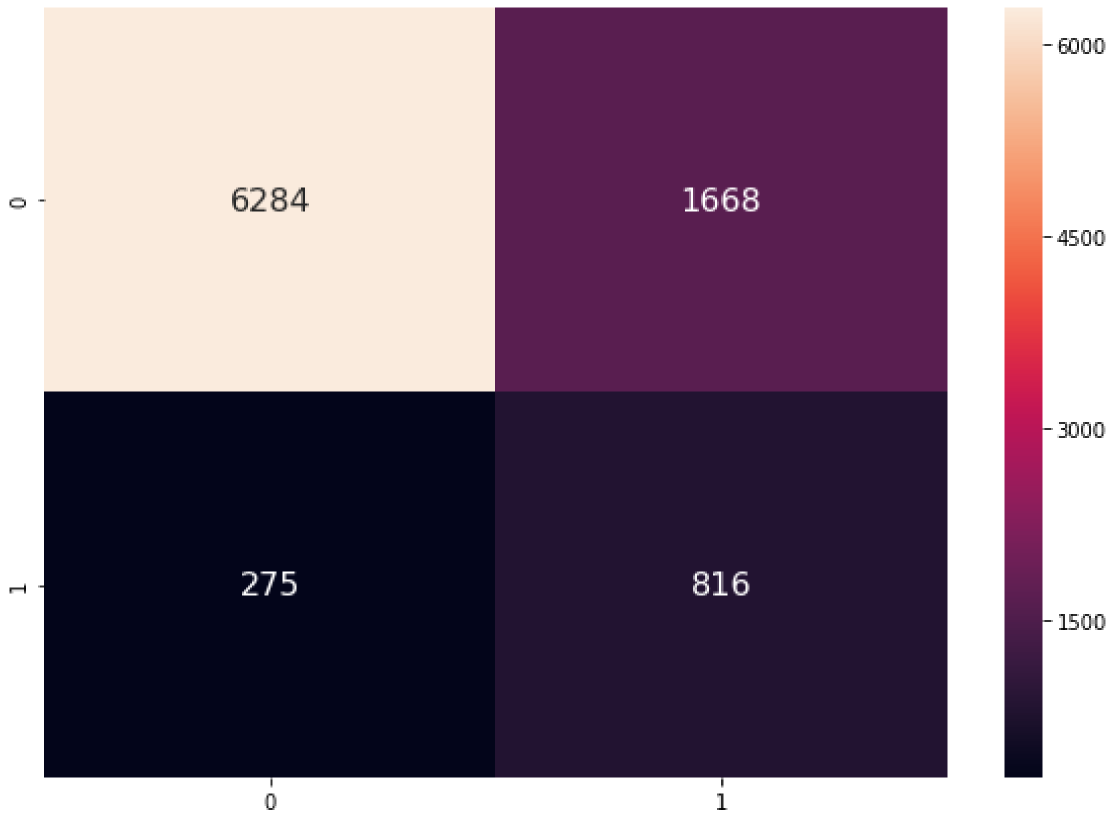


Test Confusion Matrix

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

cma = confusion_matrix(y_test, y_predicted_test)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[6284 1668]
 [ 275  816]]
```

```
[31]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3acc446be0>
```



Stacking Classifier

Hyperparameter tuning meta-classifier (Logistic Regression)

[34]:
```
from sklearn.calibration import CalibratedClassifierCV
from mlxtend.classifier import StackingClassifier

model_1 = LogisticRegression(C=3, class_weight='balanced', n_jobs=-1)
model_1.fit(train, y_train)
clf_1 = CalibratedClassifierCV(model_1, method='sigmoid')

model_2 = RandomForestClassifier(n_estimators=250, max_depth=25,␣
 ↪class_weight='balanced', n_jobs=-1)
model_2.fit(train, y_train)
clf_2 = CalibratedClassifierCV(model_2, method='sigmoid')

model_3 = SGDClassifier(alpha=0.0001, class_weight='balanced', n_jobs=-1)
model_3.fit(train, y_train)
clf_3 = CalibratedClassifierCV(model_3, method='sigmoid')

model_4 = XGBClassifier(max_depth=5, n_estimators=100 ,class_weight='balanced',␣
 ↪n_jobs=-1)
```

```
model_4.fit(train, y_train)
clf_4 = CalibratedClassifierCV(model_4, method='sigmoid')

C = [0.0001,0.001,0.01,0.1,1,10]
roc = 0
best_C = 0
for i in C:
    log_reg = LogisticRegression(C=i, n_jobs=-1)
    model = StackingClassifier(classifiers=[clf_1, clf_2, clf_3, clf_4],␣
 ↪meta_classifier=log_reg, use_probas=True)
    model.fit(train, y_train)
    model_roc = roc_auc_score(y_test, model.predict_proba(test)[:, 1])
    if roc < model_roc:
        roc = model_roc
        best_C = i
```

[35]:
```
best_C
```

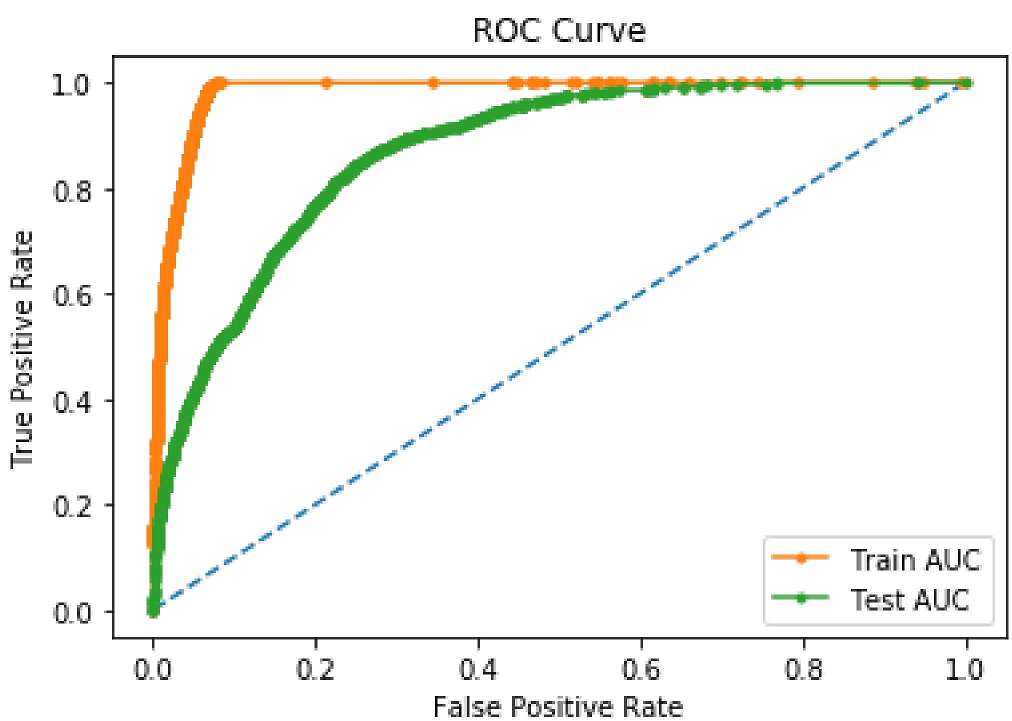
[35]: 0.0001

Training stacking classifier with best hyperparameter for meta-classifier

[36]:
```
from mlxtend.classifier import StackingClassifier

log_reg = LogisticRegression(C=0.0001, n_jobs=-1)
stack_clf = StackingClassifier(classifiers=[clf_1, clf_2, clf_3, clf_4],␣
 ↪meta_classifier=log_reg, use_probas=True)
stack_clf.fit(train, y_train)

y_probs_train = stack_clf.predict_proba(train)
y_probs_test = stack_clf.predict_proba(test)
y_predicted_train = stack_clf.predict(train)
y_predicted_test = stack_clf.predict(test)

# keep probabilities for the positive outcome only
y_probs_train = y_probs_train[:, 1]
y_probs_test = y_probs_test[:, 1]

# calculate AUC and Accuracy
train_auc = roc_auc_score(y_train, y_probs_train)
test_auc = roc_auc_score(y_test, y_probs_test)
train_acc = accuracy_score(y_train, y_predicted_train)
test_acc = accuracy_score(y_test, y_predicted_test)
print('*'*50)
print('Train AUC: %.3f' % train_auc)
print('Test AUC: %.3f' % test_auc)
print('*'*50)
```

```
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % test_acc)

score['Stacking Classifier'] = [test_auc, test_acc]

# calculate roc curve
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_probs_train)
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_probs_test)
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the stack_clf
plt.plot(train_fpr, train_tpr, marker='.', label='Train AUC')
plt.plot(test_fpr, test_tpr, marker='.', label='Test AUC')
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.show()
```

```
**************************************************
Train AUC: 0.983
Test AUC: 0.871
**************************************************
Train Accuracy: 0.884
Test Accuracy: 0.879
```

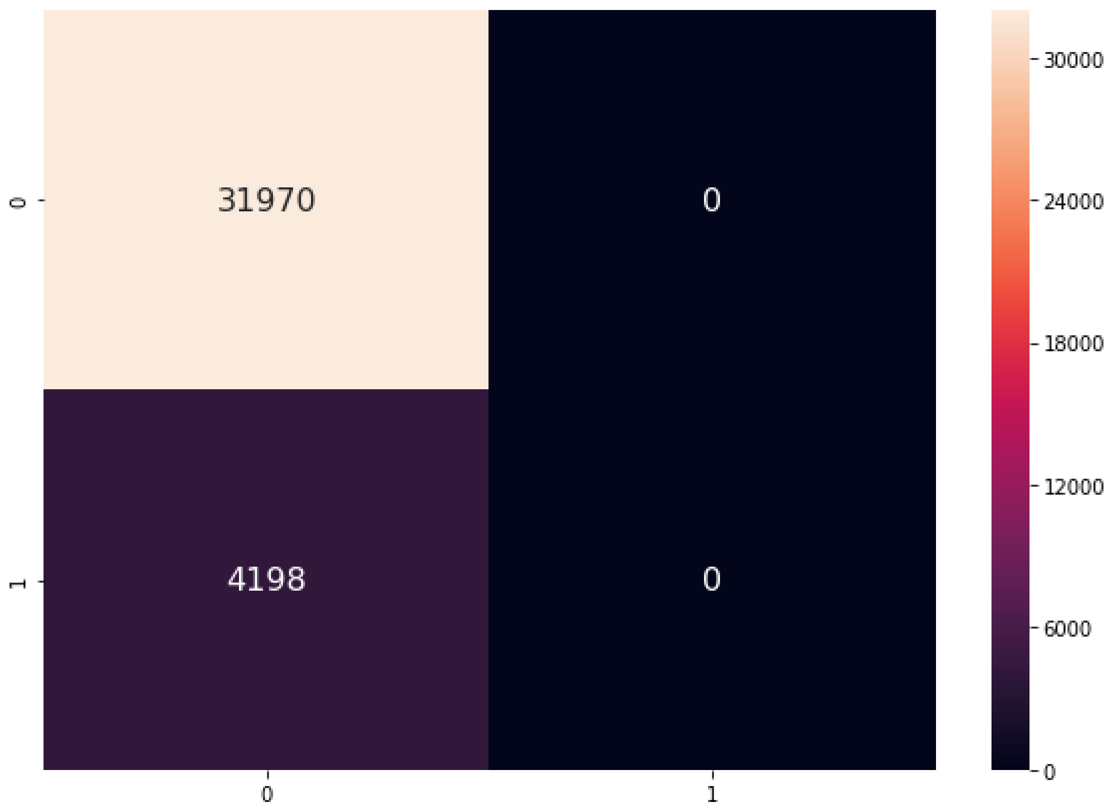


Train Confusion Matrix

[37]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_train, y_predicted_train)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[31970     0]
 [ 4198     0]]
```

[37]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3abc696fd0>



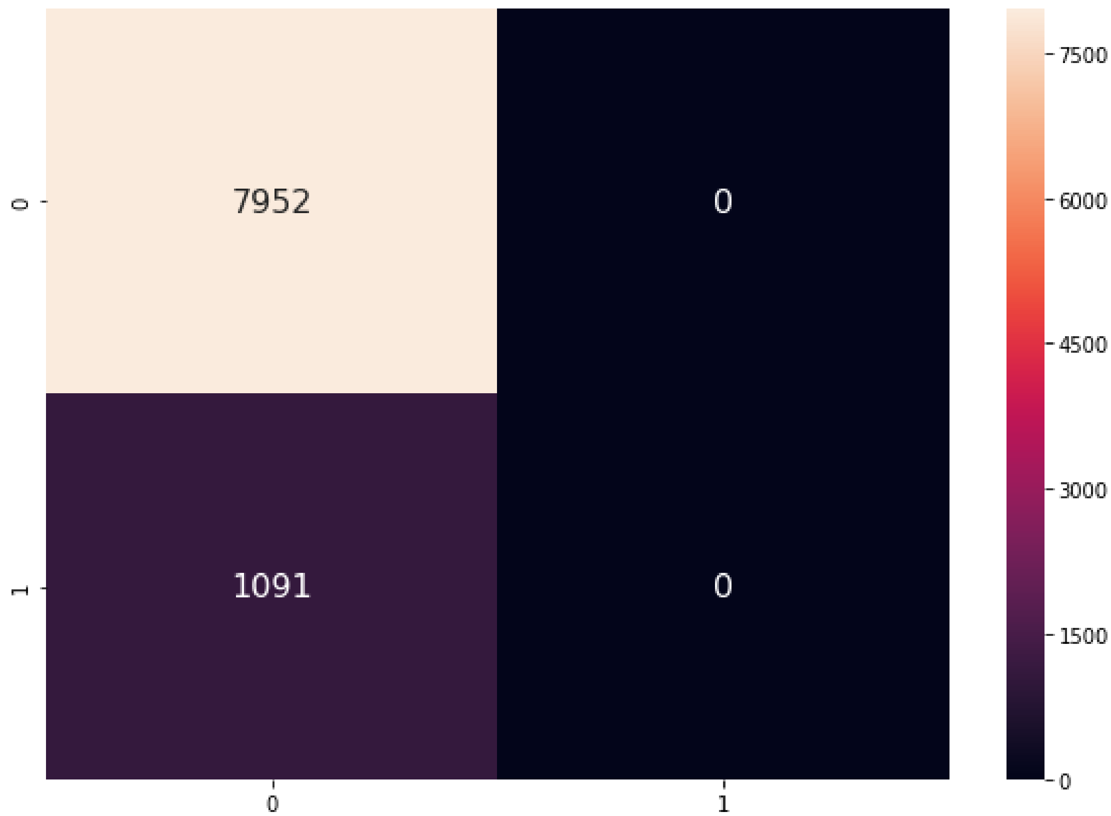
Test Confusion Matrix

[38]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_test, y_predicted_test)
print('Confusion matrix:\n', cma)
```

```
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[7952    0]
 [1091    0]]
```

[38]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3ab5f84978>

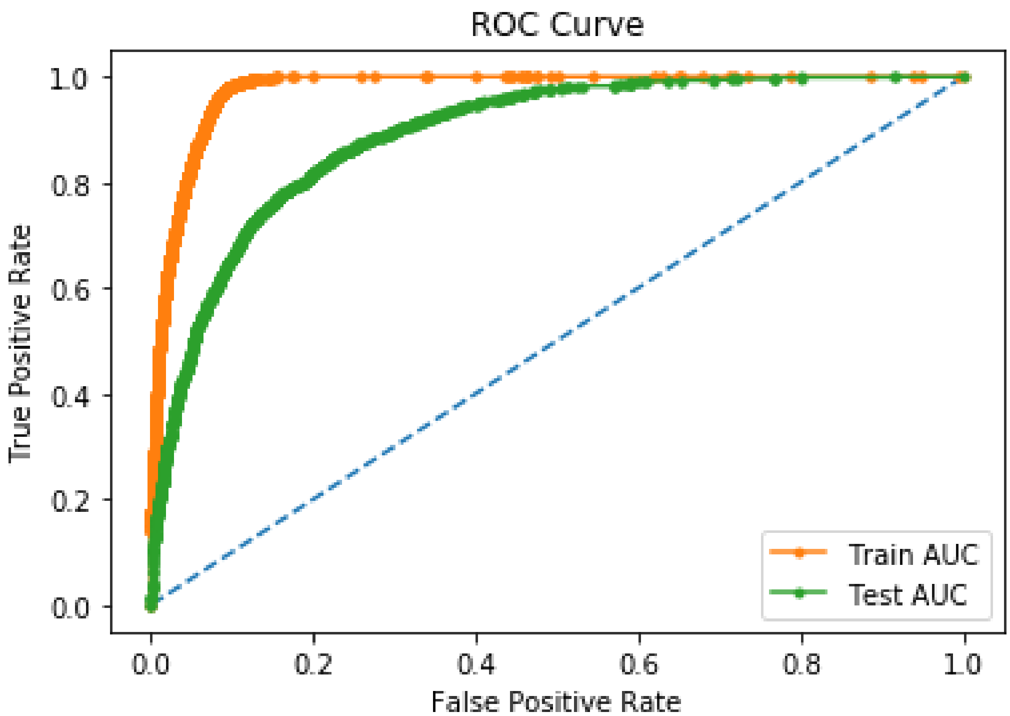


Voting Classifier

[40]:
```
from sklearn.ensemble import VotingClassifier
model = VotingClassifier(estimators=[('log_reg', clf_1), ('rf', model_2),␣
 ↪('stack', stack_clf), ('xgb', model_4), ('log_reg_1', model_1)],␣
 ↪voting='soft')
model.fit(train, y_train)

y_probs_train = model.predict_proba(train)
y_probs_test = model.predict_proba(test)
y_predicted_train = model.predict(train)
y_predicted_test = model.predict(test)
```

```python
# keep probabilities for the positive outcome only
y_probs_train = y_probs_train[:, 1]
y_probs_test = y_probs_test[:, 1]

# calculate AUC and Accuracy
train_auc = roc_auc_score(y_train, y_probs_train)
test_auc = roc_auc_score(y_test, y_probs_test)
train_acc = accuracy_score(y_train, y_predicted_train)
test_acc = accuracy_score(y_test, y_predicted_test)
print('*'*50)
print('Train AUC: %.3f' % train_auc)
print('Test AUC: %.3f' % test_auc)
print('*'*50)
print('Train Accuracy: %.3f' % train_acc)
print('Test Accuracy: %.3f' % test_acc)

score['Voting Classifier'] = [test_auc, test_acc]

# calculate roc curve
train_fpr, train_tpr, train_thresholds = roc_curve(y_train, y_probs_train)
test_fpr, test_tpr, test_thresholds = roc_curve(y_test, y_probs_test)
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(train_fpr, train_tpr, marker='.', label='Train AUC')
plt.plot(test_fpr, test_tpr, marker='.', label='Test AUC')
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.show()
```

```
**************************************************
Train AUC: 0.977
Test AUC: 0.892
**************************************************
Train Accuracy: 0.929
Test Accuracy: 0.894
```

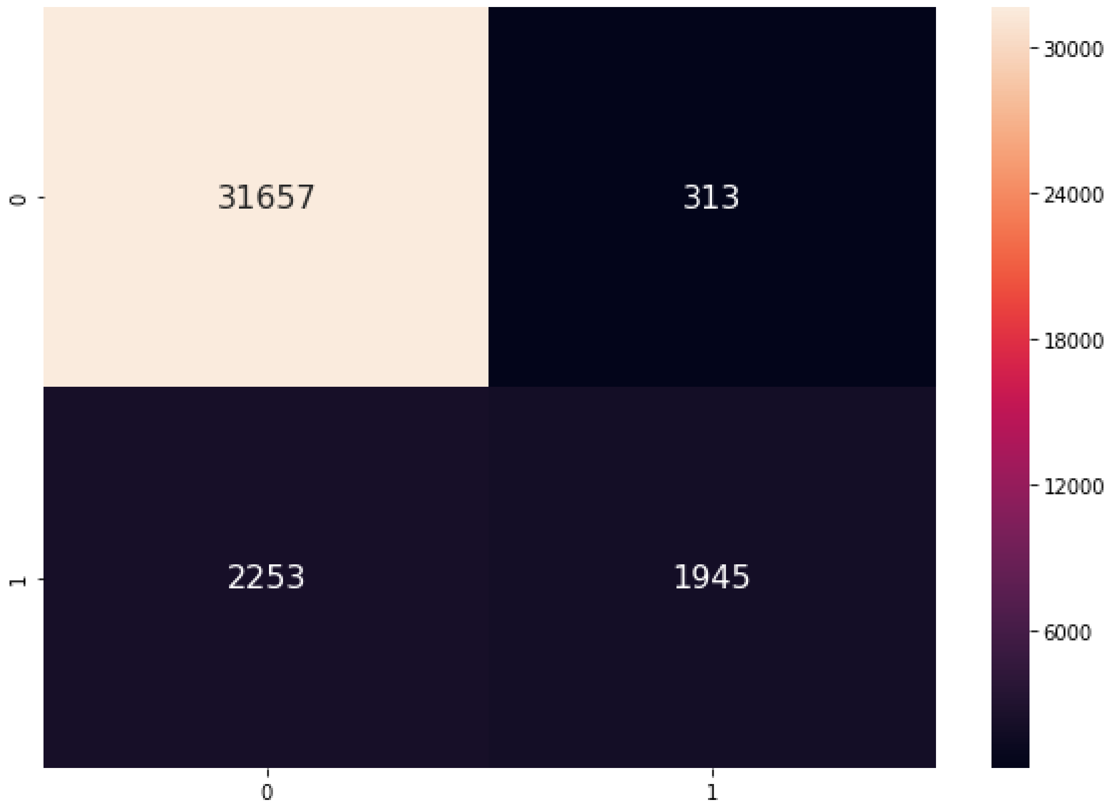


Train Confusion Matrix

[41]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_train, y_predicted_train)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[31657   313]
 [ 2253  1945]]
```

[41]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3abc1dcc50>
```

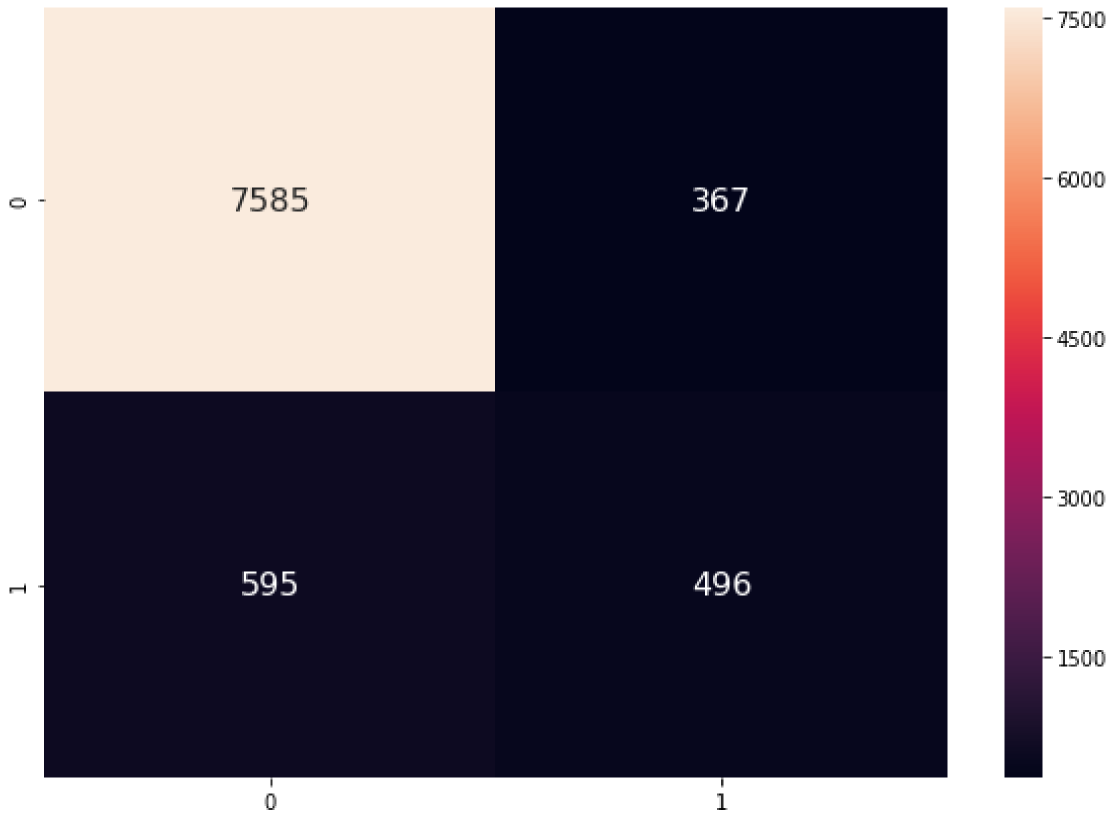


Test Confusion Matrix

[42]:
```
from sklearn.metrics import confusion_matrix

cma = confusion_matrix(y_test, y_predicted_test)
print('Confusion matrix:\n', cma)
df_cm = pd.DataFrame(cma, range(2), columns=range(2))
plt.figure(figsize = (10,7))
sns.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

```
Confusion matrix:
 [[7585  367]
 [ 595  496]]
```

[42]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3ac1115e10>



Conclusion

- It was a great learning experience working on a financial dataset.
- Our dataset consist of categorical and numerical features.
- We have 16 independent features, out of these only half of them are important.
- 'duration' is the most important feature while 'education' is the least important feature.
- Month of May have seen the highest number of clients contacted but have the least success rate. Highest success rate is observed for end month of the financial year as well as the calendar year. So one can say that our dataset have some kind of seasonality.
- When visualized age in groups, it is found that clients with age less than 30 and greater than 60 are less contacted through the campaign but have a higher success rate.
- Different machine learning models are trained and tested on the dataset. Out of those Voting Classifier performs best. Logistic Regression is also an important model as it results in high AUC score.
- Different models are summarized in table below.

[55]:
```
print('***************  Comparison of different models  ****************')
table = PrettyTable(['Model', 'Test AUC', 'Test Accuracy'])
for item in score.items():
```

```python
    table.add_row([item[0], item[1][0], item[1][1]])
print(table)
```

```
***************  Comparison of different models  ****************
+---------------------+--------------------+--------------------+
|        Model        |      Test AUC      |   Test Accuracy    |
+---------------------+--------------------+--------------------+
| Logistic Regression | 0.8867956824355852 | 0.7464337056286631 |
|    Random Forest    | 0.7991005150979203 | 0.8122304544951896 |
|         SVM         | 0.8075457787974408 | 0.8053743226805263 |
|       XGBoost       | 0.853617292665249  | 0.7851376755501492 |
| Stacking Classifier | 0.871481409077748  | 0.8793541966161672 |
|  Voting Classifier  | 0.8921652047943022 | 0.8936193741015149 |
+---------------------+--------------------+--------------------+
```

References/Citations

1. [Moro et al., 2014] S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014

2. https://archive.ics.uci.edu