Exercise Python Code

Ying Guo

1 Import all the libraries necessary for this project

In [ ]:
```
# data analysis and wrangling
import pandas as pd
import numpy as np
from scipy import stats


#Virualization
import matplotlib.pyplot as plt
import seaborn as sns

# machine learning
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.feature_selection import RFE
from sklearn.metrics import roc_auc_score
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report, precision_score
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
from imblearn.pipeline import Pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import GradientBoostingClassifier


# For oversampling Library (Dealing with Imbalanced Datasets)
from imblearn.over_sampling import SMOTE
```

2 Import data

In [3]:
```
# Read the file into a DataFrame: df
df = pd.read_csv('exercise_04_train.csv')
dftest = pd.read_csv('exercise_04_test.csv')
```

3 Features

```
In [4]: # To show all the columns, use the following command
        # pd.options.display.max_columns = 4000
        print(df.columns.values)
        print(df.head())
        print(df.shape)
```

```
['x0' 'x1' 'x2' 'x3' 'x4' 'x5' 'x6' 'x7' 'x8' 'x9' 'x10' 'x11' 'x12' 'x13'
 'x14' 'x15' 'x16' 'x17' 'x18' 'x19' 'x20' 'x21' 'x22' 'x23' 'x24' 'x25'
 'x26' 'x27' 'x28' 'x29' 'x30' 'x31' 'x32' 'x33' 'x34' 'x35' 'x36' 'x37'
 'x38' 'x39' 'x40' 'x41' 'x42' 'x43' 'x44' 'x45' 'x46' 'x47' 'x48' 'x49'
 'x50' 'x51' 'x52' 'x53' 'x54' 'x55' 'x56' 'x57' 'x58' 'x59' 'x60' 'x61'
 'x62' 'x63' 'x64' 'x65' 'x66' 'x67' 'x68' 'x69' 'x70' 'x71' 'x72' 'x73'
 'x74' 'x75' 'x76' 'x77' 'x78' 'x79' 'x80' 'x81' 'x82' 'x83' 'x84' 'x85'
 'x86' 'x87' 'x88' 'x89' 'x90' 'x91' 'x92' 'x93' 'x94' 'x95' 'x96' 'x97'
 'x98' 'x99' 'y']
          x0         x1        x2         x3         x4         x5        x6  \
0 -17.933519   6.559220  2.422468 -27.737392 -12.080601  -3.892934  1.067466
1 -37.214754  10.774930  5.404072  21.354738   0.612690  -3.093533  6.161558
2   0.330441 -19.609972 -1.331804 -15.153892  19.710240  19.077300 -1.747110
3 -13.709765  -8.011390 -1.536483  23.129497  27.880879  20.573991 -1.617689
4  -4.202598   7.076210  8.881550  23.600777  26.232164 -14.462320  3.231193

         x7         x8        x9 ...        x91        x92   x93       x94  \
0  0.935953  10.912007  1.107144 ...  11.107047   0.093337  asia  0.421524
1 -0.972156  -5.222169  0.384969 ...  -1.991846  15.666187  asia -0.132764
2  0.545570  -1.464609  3.670570 ...  17.132840  -5.333012  asia  1.432308
3  4.129694   1.139928  2.912838 ...  12.292136   4.177925  asia  0.733069
4 -0.069364  -7.310536 -2.268700 ...   6.218743   8.715709  asia -0.977502

         x95        x96        x97       x98       x99  y
0  35.259947   8.994318 -21.000182 -0.686588  2.949106  1
1  -1.192563   3.885024 -37.886523 -7.730392 -1.107330  0
2  -3.435427  -1.133450   7.426099 -5.945534  1.316312  0
3   4.372964  15.529931  29.712153  2.240740  0.477195  0
4 -30.085932  -8.244312  66.540331 -3.478195 -2.869702  1

[5 rows x 101 columns]
(40000, 101)
```

4 Exploring features

Categorical features: x34, x35, x68, and x93
Numerical features: x1 - x33, x36 - x67, x69 - x92, and x94 - x99
Features need to be cleaned up: x35, x41, x45, and x68

In [5]: df.describe()

Out[5]:

	x0	x1	x2	x3	x4	\
count	39989.000000	39989.000000	39993.000000	39991.000000	39992.000000	
mean	6.159970	-3.568111	0.223336	-1.742588	0.079437	
std	29.098537	17.186748	5.237987	36.601044	21.179065	
min	-106.809919	-72.864290	-21.508799	-157.569819	-79.900790	
25%	-13.617383	-15.148354	-3.295204	-26.465502	-14.215354	
50%	6.247370	-3.660536	0.264994	-1.638876	0.113879	
75%	25.570242	7.807474	3.761013	23.044686	14.365631	
max	134.592465	71.071223	21.060130	145.566756	89.856546	

	x5	x6	x7	x8	x9	\
count	39989.000000	39993.000000	39988.000000	39996.000000	39992.000000	
mean	-0.535399	0.015483	-0.011955	-3.055506	-0.023167	
std	13.602122	4.110412	2.423051	13.450495	2.472008	
min	-55.050043	-15.955862	-9.299563	-54.415601	-9.674058	
25%	-9.771613	-2.770450	-1.644516	-12.055884	-1.683043	
50%	-0.530463	0.015259	-0.002569	-3.069374	-0.039400	
75%	8.673525	2.770460	1.621142	5.910663	1.636558	
max	52.628375	18.546313	11.919020	54.262047	9.492780	

	...	x90	x91	x92	x94	\
count	...	39995.000000	39995.000000	39992.000000	39990.000000	
mean	...	-7.472520	-0.026534	0.016619	-0.000084	
std	...	85.885663	9.446348	5.585176	1.135819	
min	...	-375.460243	-36.618364	-24.268022	-4.928351	
25%	...	-64.312552	-6.390111	-3.764955	-0.771053	
50%	...	-5.892459	-0.074239	0.025084	0.001850	
75%	...	50.873797	6.360710	3.784911	0.767160	
max	...	336.414571	42.835142	23.505468	4.792344	

	x95	x96	x97	x98	x99	\
count	39992.000000	39985.000000	39991.000000	39995.000000	39990.000000	
mean	0.054600	-0.459762	-4.925135	0.033761	0.120155	
std	22.278277	12.702453	34.931541	5.374336	3.116143	
min	-101.342320	-57.873114	-140.638773	-22.402508	-13.024105	
25%	-14.881499	-8.968785	-28.431741	-3.590052	-1.992603	
50%	0.239447	-0.371605	-5.023371	0.031702	0.115059	
75%	15.109761	8.128631	18.412348	3.663242	2.230546	
max	92.442885	52.159468	147.391902	21.614385	13.208294	

	y
count	40000.000000
mean	0.203000
std	0.402238
min	0.000000
25%	0.000000

```
50%        0.000000
75%        0.000000
max        1.000000

[8 rows x 97 columns]
```

In [6]:
```python
# All features with object data type
df.loc[:, df.dtypes == object].head()
```

Out[6]:

	x34	x35	x41	x45	x68	x93
0	bmw	thur	$-1306.52	-0.01%	sept.	asia
1	Toyota	wednesday	$-24.86	0.0%	July	asia
2	bmw	thurday	$-110.85	0.0%	July	asia
3	Toyota	wed	$-324.43	0.01%	Apr	asia
4	Toyota	wednesday	$1213.37	-0.01%	Aug	asia

In [7]:
```python
# Clean up x35, x41, x45, and x68 on both training and test data
# Training data
Xorg = df
Xorg.x35 = Xorg.x35.replace('wed','wednesday')
Xorg.x35 = Xorg.x35.replace(['thurday', 'thur'],'thursday')
Xorg.x35 = Xorg.x35.replace('fri','friday')
Xorg['x41'] = Xorg['x41'].str.replace('$', '').astype('float')
print(Xorg['x41'].head())
Xorg['x45'] = Xorg['x45'].str.replace('%', '').astype('float')/100
print(Xorg['x45'].head())
Xorg.x68 = Xorg.x68.replace('January','Jan')
Xorg.x68 = Xorg.x68.replace('July','Jul')
Xorg.x68 = Xorg.x68.replace('sept.','Sep')
Xorg.x68 = Xorg.x68.replace('Dev','Dec')
train = Xorg
print(train.shape)

# Test data
Xorg = dftest
Xorg.x35 = Xorg.x35.replace('wed','wednesday')
Xorg.x35 = Xorg.x35.replace(['thurday', 'thur'],'thursday')
Xorg.x35 = Xorg.x35.replace('fri','friday')
Xorg['x41'] = Xorg['x41'].str.replace('$', '').astype('float')
print(Xorg['x41'].head())
Xorg['x45'] = Xorg['x45'].str.replace('%', '').astype('float')/100
print(Xorg['x45'].head())
Xorg.x68 = Xorg.x68.replace('January','Jan')
Xorg.x68 = Xorg.x68.replace('July','Jul')
Xorg.x68 = Xorg.x68.replace('sept.','Sep')
Xorg.x68 = Xorg.x68.replace('Dev','Dec')
testdata = Xorg
print(testdata.shape)
```

```
0   -1306.52
1     -24.86
2    -110.85
3    -324.43
4    1213.37
Name: x41, dtype: float64
0   -0.0001
1    0.0000
2    0.0000
3    0.0001
4   -0.0001
Name: x45, dtype: float64
(40000, 101)
0     124.72
1    1273.04
2   -1651.19
3     896.05
4   -1710.27
Name: x41, dtype: float64
0   -0.0001
1   -0.0001
2    0.0000
3    0.0001
4    0.0001
Name: x45, dtype: float64
(10000, 100)
```

5 Outliers Detection

I applied median-absolute-deviation (MAD) based outlier detection for all numerical features. I used a threshold of 3.5. A data point with Z score whose absolute value larger than 3.5 is labeled as an outlier.

I found 1927 instances with outliers (about 5% of all cases). Without knowing what each feature actually is, it is hard to decide if these outliers are valid data or wrong inputs. I test training data with and without outliers. The results are very similar. Thus I use the dataset with outliers here.

In [8]:
```
allnum = train[train.loc[:, train.dtypes == float].columns.tolist()]
def mad_based_outlier(points, thresh=3.5):
    if len(points.shape) == 1:
        points = points[:,None]
    median = np.nanmedian(points, axis=0)
    diff = np.sum((points - median)**2, axis=-1)
    diff = np.sqrt(diff)
    med_abs_deviation = np.nanmedian(diff)

    modified_z_score = 0.6745 * diff / med_abs_deviation
```

```
        return modified_z_score > thresh

    todrop = list()
    for i in range(len(allnum.columns)):
        ind = allnum.iloc[:,i][mad_based_outlier(allnum.iloc[:,i],thresh=3.5)]
        todrop = list(set(todrop+ind.index.tolist()))
    print(len(todrop))
```

```
1927
```

```
In [9]: noout = train.drop(train.index[[todrop]])
        noout.shape
```

```
Out[9]: (38093, 101)
```

6 Not many missing data in each feature in both training and test data

I will impute the missing data using median in the analysis (more robust to outliers).

```
In [10]: print(train.describe())

         dftest.info()
```

```
                 x0            x1            x2            x3            x4  \
count  39989.000000  39989.000000  39993.000000  39991.000000  39992.000000
mean       6.159970     -3.568111      0.223336     -1.742588      0.079437
std       29.098537     17.186748      5.237987     36.601044     21.179065
min     -106.809919    -72.864290    -21.508799   -157.569819    -79.900790
25%      -13.617383    -15.148354     -3.295204    -26.465502    -14.215354
50%        6.247370     -3.660536      0.264994     -1.638876      0.113879
75%       25.570242      7.807474      3.761013     23.044686     14.365631
max      134.592465     71.071223     21.060130    145.566756     89.856546

                 x5            x6            x7            x8            x9  \
count  39989.000000  39993.000000  39988.000000  39996.000000  39992.000000
mean      -0.535399      0.015483     -0.011955     -3.055506     -0.023167
std       13.602122      4.110412      2.423051     13.450495      2.472008
min      -55.050043    -15.955862     -9.299563    -54.415601     -9.674058
25%       -9.771613     -2.770450     -1.644516    -12.055884     -1.683043
50%       -0.530463      0.015259     -0.002569     -3.069374     -0.039400
75%        8.673525      2.770460      1.621142      5.910663      1.636558
max       52.628375     18.546313     11.919020     54.262047      9.492780

           ...              x90           x91           x92           x94  \
count      ...     39995.000000  39995.000000  39992.000000  39990.000000
mean       ...        -7.472520     -0.026534      0.016619     -0.000084
```

```
std      ...     85.885663    9.446348    5.585176    1.135819
min      ...   -375.460243  -36.618364  -24.268022   -4.928351
25%      ...    -64.312552   -6.390111   -3.764955   -0.771053
50%      ...     -5.892459   -0.074239    0.025084    0.001850
75%      ...     50.873797    6.360710    3.784911    0.767160
max      ...    336.414571   42.835142   23.505468    4.792344

                x95           x96           x97           x98           x99  \
count  39992.000000  39985.000000  39991.000000  39995.000000  39990.000000
mean       0.054600     -0.459762     -4.925135      0.033761      0.120155
std       22.278277     12.702453     34.931541      5.374336      3.116143
min     -101.342320    -57.873114   -140.638773    -22.402508    -13.024105
25%      -14.881499     -8.968785    -28.431741     -3.590052     -1.992603
50%        0.239447     -0.371605     -5.023371      0.031702      0.115059
75%       15.109761      8.128631     18.412348      3.663242      2.230546
max       92.442885     52.159468    147.391902     21.614385     13.208294

                  y
count  40000.000000
mean       0.203000
std        0.402238
min        0.000000
25%        0.000000
50%        0.000000
75%        0.000000
max        1.000000

[8 rows x 97 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 100 columns):
x0     9997 non-null float64
x1     9999 non-null float64
x2     9998 non-null float64
x3     9996 non-null float64
x4     10000 non-null float64
x5     10000 non-null float64
x6     9996 non-null float64
x7     9999 non-null float64
x8     9997 non-null float64
x9     9999 non-null float64
x10    9999 non-null float64
x11    9997 non-null float64
x12    10000 non-null float64
x13    9994 non-null float64
x14    9998 non-null float64
x15    9997 non-null float64
x16    9998 non-null float64
```

x17	9997 non-null float64
x18	9998 non-null float64
x19	9998 non-null float64
x20	9998 non-null float64
x21	10000 non-null float64
x22	10000 non-null float64
x23	9997 non-null float64
x24	9996 non-null float64
x25	9997 non-null float64
x26	9998 non-null float64
x27	9995 non-null float64
x28	9997 non-null float64
x29	9999 non-null float64
x30	10000 non-null float64
x31	9997 non-null float64
x32	9999 non-null float64
x33	9997 non-null float64
x34	9999 non-null object
x35	9998 non-null object
x36	9997 non-null float64
x37	9999 non-null float64
x38	9999 non-null float64
x39	9998 non-null float64
x40	10000 non-null float64
x41	10000 non-null float64
x42	9998 non-null float64
x43	10000 non-null float64
x44	9999 non-null float64
x45	9998 non-null float64
x46	9999 non-null float64
x47	9999 non-null float64
x48	9994 non-null float64
x49	9997 non-null float64
x50	10000 non-null float64
x51	9998 non-null float64
x52	9999 non-null float64
x53	9999 non-null float64
x54	10000 non-null float64
x55	9999 non-null float64
x56	9999 non-null float64
x57	9997 non-null float64
x58	9998 non-null float64
x59	9997 non-null float64
x60	9997 non-null float64
x61	9997 non-null float64
x62	9996 non-null float64
x63	9998 non-null float64
x64	10000 non-null float64

```
x65    9998 non-null float64
x66    9998 non-null float64
x67    9996 non-null float64
x68    10000 non-null object
x69    9997 non-null float64
x70    9998 non-null float64
x71    9999 non-null float64
x72    10000 non-null float64
x73    9995 non-null float64
x74    9997 non-null float64
x75    9998 non-null float64
x76    10000 non-null float64
x77    9995 non-null float64
x78    9999 non-null float64
x79    9997 non-null float64
x80    9998 non-null float64
x81    9998 non-null float64
x82    9997 non-null float64
x83    9999 non-null float64
x84    10000 non-null float64
x85    9997 non-null float64
x86    9997 non-null float64
x87    9998 non-null float64
x88    9998 non-null float64
x89    9998 non-null float64
x90    9997 non-null float64
x91    10000 non-null float64
x92    10000 non-null float64
x93    9999 non-null object
x94    9998 non-null float64
x95    9999 non-null float64
x96    9998 non-null float64
x97    9996 non-null float64
x98    9998 non-null float64
x99    9995 non-null float64
dtypes: float64(96), object(4)
memory usage: 7.6+ MB
```

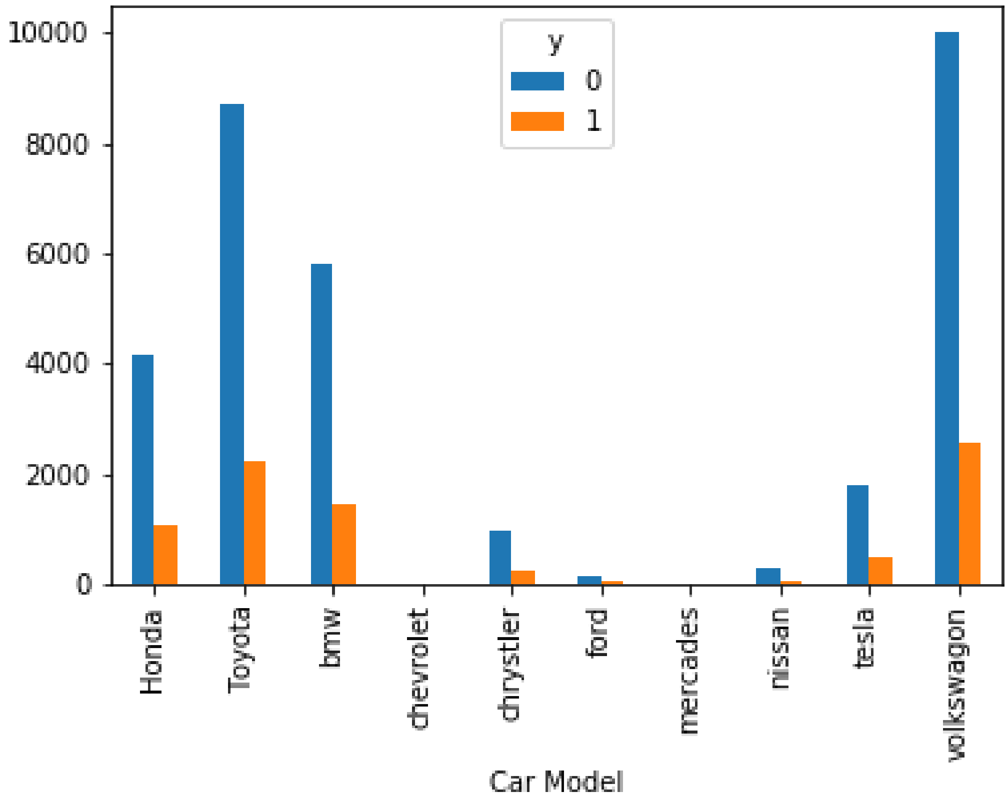
7 check number of instances in each catetory for four categorial features

In [11]:
```
print(train.x34.value_counts(dropna = False))
print(train.x35.value_counts(dropna = False))
print(train.x68.value_counts(dropna = False))
print(train.x93.value_counts(dropna = False))
```

```
volkswagon    12557
Toyota        10922
bmw            7288
Honda          5195
tesla          2286
chrystler      1209
nissan          339
ford            160
mercades         26
chevrolet        10
NaN               8
Name: x34, dtype: int64
wednesday    20756
thursday     17726
tuesday        898
friday         550
monday          59
NaN             11
Name: x35, dtype: int64
Jul    11146
Jun     9289
Aug     8115
May     4833
Sep     3441
Apr     1638
Oct      886
Mar      397
Nov      160
Feb       52
Dec       20
Jan       12
NaN       11
Name: x68, dtype: int64
asia       35434
america     3136
euorpe      1423
NaN            7
Name: x93, dtype: int64
```

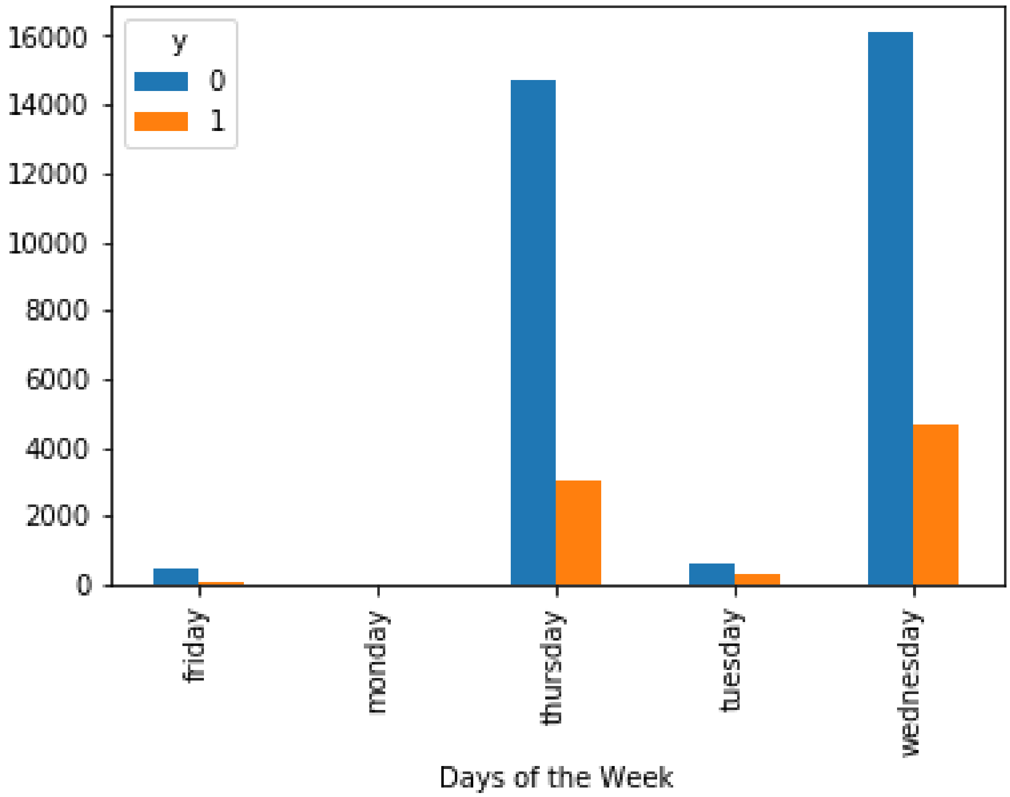
In [12]:
```
%matplotlib inline
pd.crosstab(train.x34,train.y).plot(kind='bar')
#plt.title('')
plt.xlabel('Car Model')
#plt.ylabel('Type of loans')
```

Out[12]: Text(0.5,0,'Car Model')



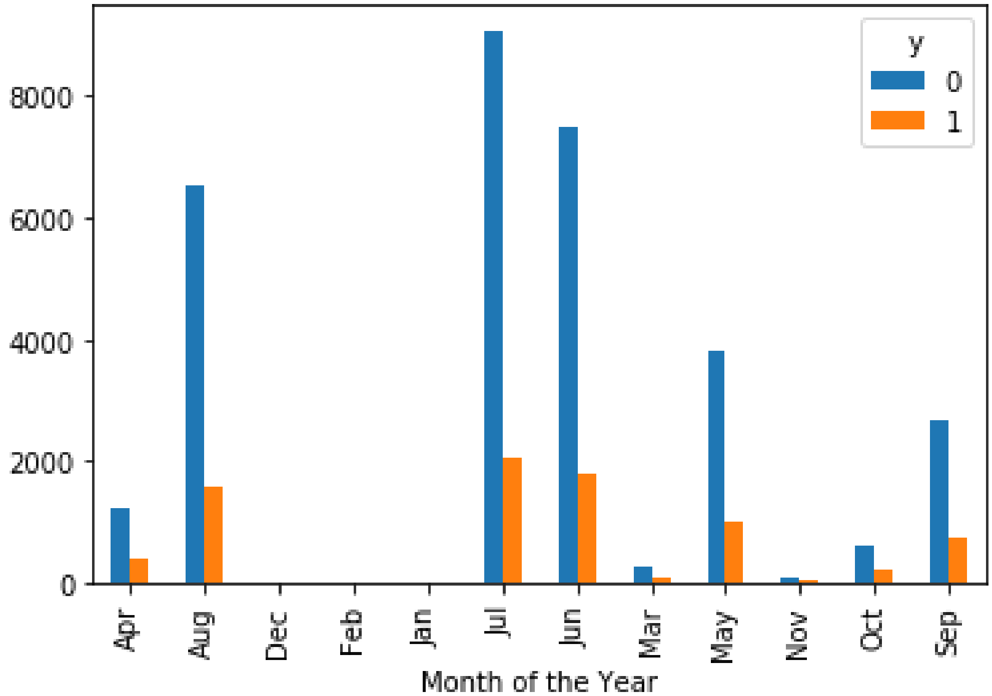
In [13]:
```
%matplotlib inline
pd.crosstab(train.x35,train.y).plot(kind='bar')
#plt.title('')
plt.xlabel('Days of the Week')
#plt.ylabel('Type of loans')
```

Out[13]: Text(0.5,0,'Days of the Week')



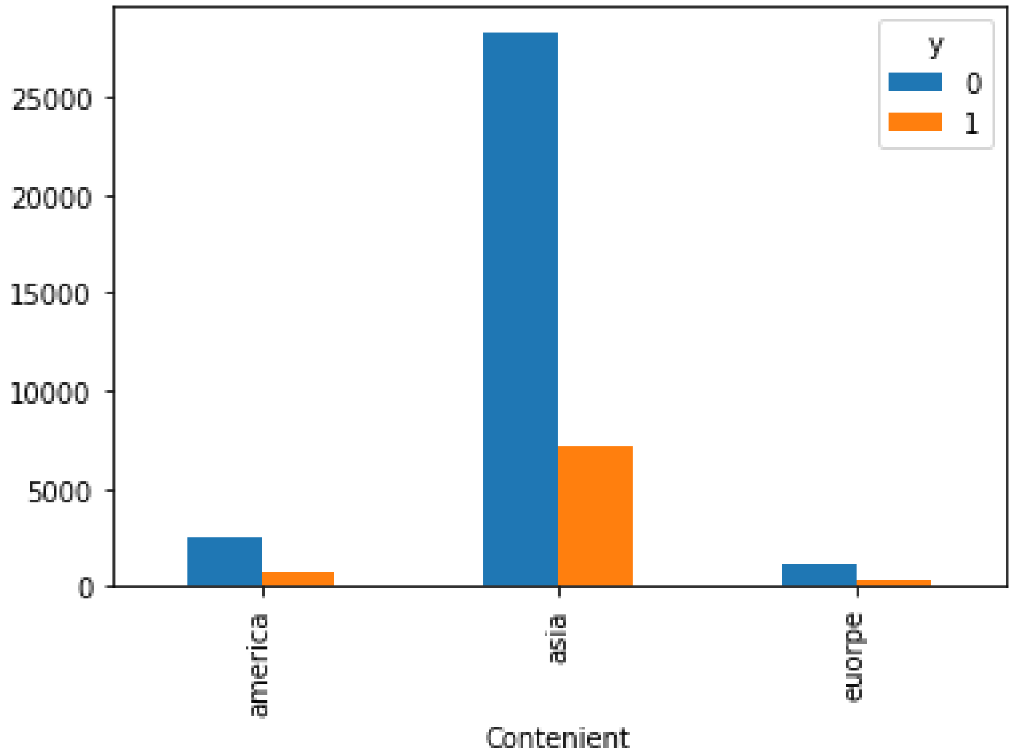

```
In [14]: %matplotlib inline
         pd.crosstab(train.x68,train.y).plot(kind='bar')
         #plt.title('')
         plt.xlabel('Month of the Year')
         #plt.ylabel('Type of loans')

Out[14]: Text(0.5,0,'Month of the Year')
```




```
In [15]: %matplotlib inline
         pd.crosstab(train.x93,train.y).plot(kind='bar')
         #plt.title('')
         plt.xlabel('Contenient')
         #plt.ylabel('Type of loans')

Out[15]: Text(0.5,0,'Contenient')
```



In [16]: train.groupby(['y'], as_index=False).mean()

```
Out[16]:    y        x0        x1        x2        x3        x4        x5        x6  \
         0  0  7.131102 -4.431030 -0.010624  0.107336  0.164672  0.077979  0.018320
         1  1  2.347929 -0.179797  1.141973 -9.004659 -0.255125 -2.943127  0.004344

                  x7        x8   ...          x89       x90       x91       x92  \
         0 -0.019425 -2.995378   ...     0.007417 -7.688762 -0.016384  0.036807
         1  0.017378 -3.291546   ...    -0.005237 -6.623663 -0.066380 -0.062670

                 x94       x95       x96       x97       x98       x99
         0 -0.001939  0.054440  0.139299 -8.166448  0.015663 -0.034353
         1  0.007200  0.055231 -2.812450  7.798984  0.104801  0.726767

         [2 rows x 97 columns]
```

In [17]: train[["x34", "y"]].groupby(['x34'], as_index=False).mean()

```
Out[17]:          x34         y
         0      Honda  0.203465
         1     Toyota  0.204633
         2        bmw  0.200878
         3  chevrolet  0.000000
```

```
4    chrystler  0.205955
5         ford  0.193750
6     mercades  0.115385
7       nissan  0.194690
8        tesla  0.206474
9   volkswagon  0.202437
```

In [18]:
```
train[["x35", "y"]].groupby(['x35'], as_index=False).mean()
```

Out[18]:
```
         x35         y
0     friday  0.174545
1     monday  0.423729
2   thursday  0.171782
3    tuesday  0.344098
4  wednesday  0.223694
```

In [19]:
```
train[["x68", "y"]].groupby(['x68'], as_index=False).mean()
```

Out[19]:
```
    x68         y
0   Apr  0.254579
1   Aug  0.194701
2   Dec  0.400000
3   Feb  0.442308
4   Jan  0.333333
5   Jul  0.186793
6   Jun  0.195177
7   Mar  0.282116
8   May  0.213946
9   Nov  0.337500
10  Oct  0.277652
11  Sep  0.217088
```

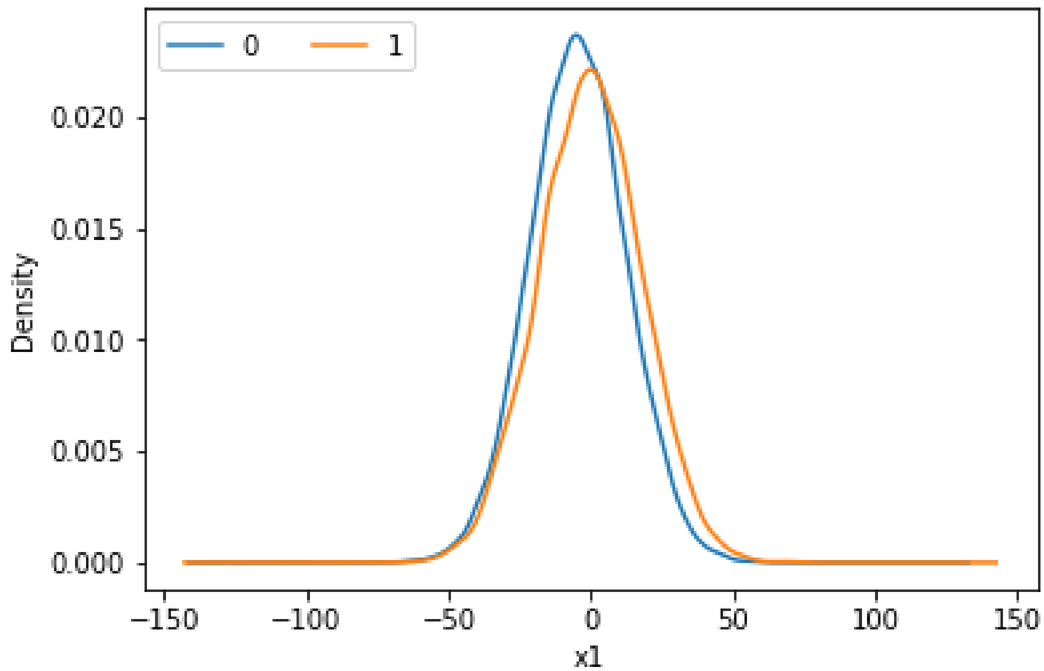
In [20]:
```
train[["x93", "y"]].groupby(['x93'], as_index=False).mean()
```

Out[20]:
```
       x93         y
0  america  0.215242
1     asia  0.202150
2   euorpe  0.196767
```

8 Exploring Numerical Features : a few example

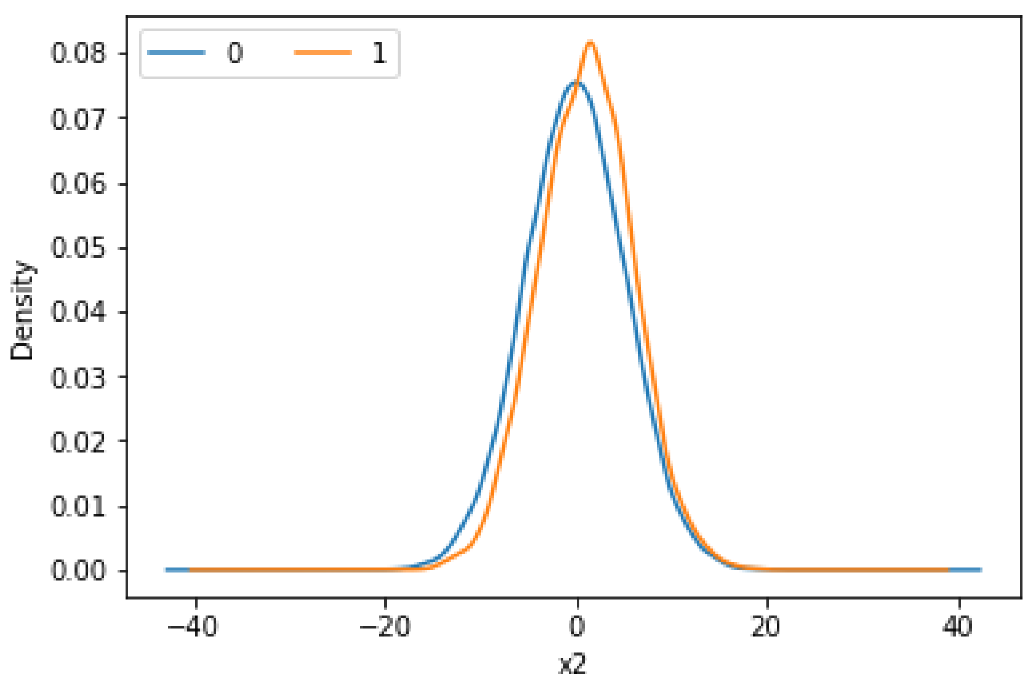
In [21]:
```
train.groupby('y').x1.plot(kind='kde')
plt.legend('01', ncol=2, loc='upper left')
plt.xlabel('x1')
```

Out[21]:
```
Text(0.5,0,'x1')
```



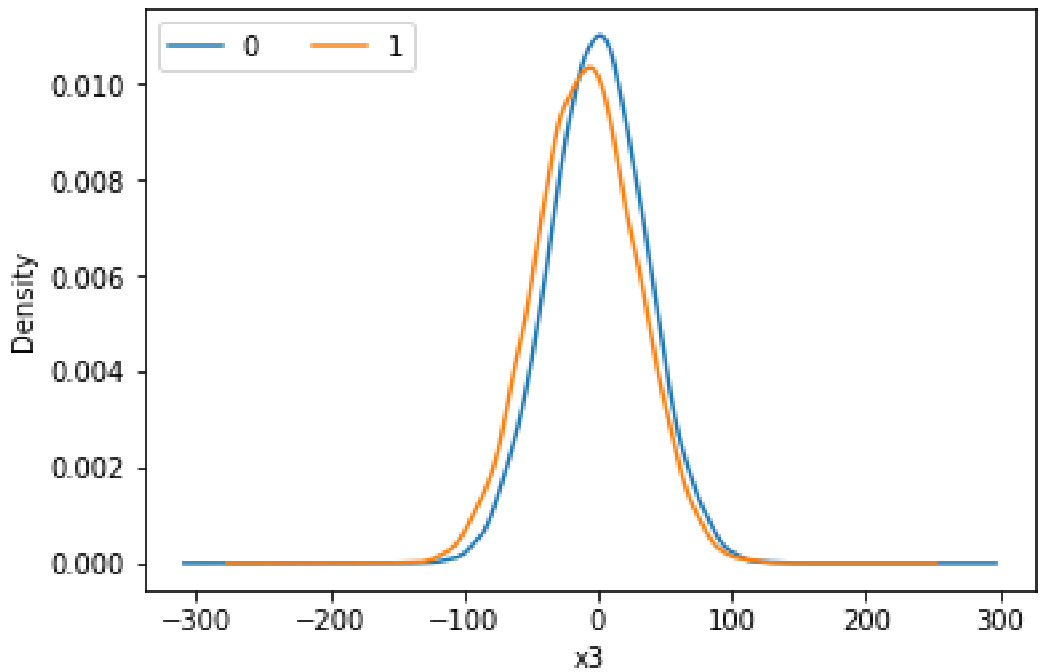
In [22]:
```
train.groupby('y').x2.plot(kind='kde')
plt.legend('01', ncol=2, loc='upper left')
plt.xlabel('x2')
```

Out[22]: Text(0.5,0,'x2')



```
In [23]: train.groupby('y').x3.plot(kind='kde')
         plt.legend('01', ncol=2, loc='upper left')
         plt.xlabel('x3')
```

```
Out[23]: Text(0.5,0,'x3')
```



9 Correlation among Features

I want to find if there are any highly correlated features. I do not want collinearilty in the feature space.

The highest absolute correlation for a pair of features is 0.412, suggesting the feature space does not contain highly correlated features.

```
In [24]: def get_redundant_pairs(df):
             '''Get diagonal and lower triangular pairs of correlation matrix'''
             pairs_to_drop = set()
             cols = df.columns
             for i in range(0, df.shape[1]):
                 for j in range(0, i+1):
                     pairs_to_drop.add((cols[i], cols[j]))
             return pairs_to_drop
```

```
def get_top_abs_correlations(df, n=5):
    au_corr = df.corr().abs().unstack()
    labels_to_drop = get_redundant_pairs(df)
    au_corr = au_corr.drop(labels=labels_to_drop).sort_values(ascending=False)
    return au_corr[0:n]

print("Top Absolute Correlations")
print(get_top_abs_correlations(allnum, 3))
```

```
Top Absolute Correlations
x41  x44    0.411508
x90  x95    0.375148
x80  x90    0.373060
dtype: float64
```

10 Divide data into features and target

Here I used all data with and without outliers. The results are similar. Here I used the data with outliers.

```
In [25]: # Keep all outliers in the data
         x = train.drop("y", axis=1)
         y = train["y"]
         print(x.shape)
         print(y.shape)

         # Remove all outliers in the data
         #x = noout.drop("y", axis=1)
         #y = noout["y"]
         #print(x.shape)
         #print(y.shape)
```

```
(40000, 100)
(40000,)
```

```
In [26]: # Split the original training data into 70% training data and 30% test data
         x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, stratify = y,
                                             random_state = 42)
```

11 Building Model 1 : Random Forest with PCA

I built the RF classifier on the reduced dimensions provided by PCA, tuning hyper-parameters with grid search on the number of components for PCA, the number of trees, quality of the split criteria, number of features to consider when looking for the best split for RF, with 5-fold cross-validation. SMOTE was performed right after PCA. The best performing model selects 110 components from PCA, and uses 500 trees, entropy as split criteria, and the square root of the total number of features as the number of features to consider when looking for the best split for RF. The accuracy score is 0.965, and the ROC AUC score is 0.9824.

In [27]:
```
from imblearn.pipeline import Pipeline

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = train.loc[:, train.dtypes == float].columns.tolist()
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['x34', 'x35', 'x68','x93']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
rf = RandomForestClassifier()
pca = PCA()
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('pca', pca),
                      ('sampling', SMOTE()),
                      ('classifier', rf)])


param_grid = { 'pca__n_components': [50,100, 110,124],
    'classifier__n_estimators': [500],
   'classifier__max_features': ['sqrt'],
    'classifier__criterion' :[ 'entropy']
}

rf_cv = GridSearchCV(clf, param_grid, cv = 5)

rf_cv.fit(x_train, y_train.values.ravel())



y_pred_proba = rf_cv.predict_proba(x_test)[:,1]
# Evaluate test-set roc_auc_score
```

```
    rf_roc_auc = roc_auc_score(y_test, y_pred_proba)

    print("model score: %.3f" % rf_cv.score(x_test, y_test))
    # Print roc_auc_score
    print('ROC AUC score: {:.4f}'.format(rf_roc_auc))
    print("Tuned Random Forest Parameter: {}".format(rf_cv.best_params_))
    print("Tuned Random Forest Accuracy: {}".format(rf_cv.best_score_))
```

```
model score: 0.966
ROC AUC score: 0.9821
Tuned Random Forest Parameter: {'classifier__criterion': 'entropy',
'classifier__max_features': 'sqrt', 'classifier__n_estimators': 500,
'pca__n_components': 110}
Tuned Random Forest Accuracy: 0.9617857142857142
```

In [28]:
```
predictions_rf = rf_cv.predict(x_test)

    # Classification Report of Prediction
    print("Classification Report:")
    print(classification_report(y_test, predictions_rf))
    # Confusion Matrix for predictions made
    conf2 = confusion_matrix(y_test,predictions_rf)
    print(conf2)
    # Plot Confusion Matrix
    label = ["0","1"]
    sns.heatmap(conf2, annot=True, xticklabels=label, yticklabels=label,fmt='g')
```

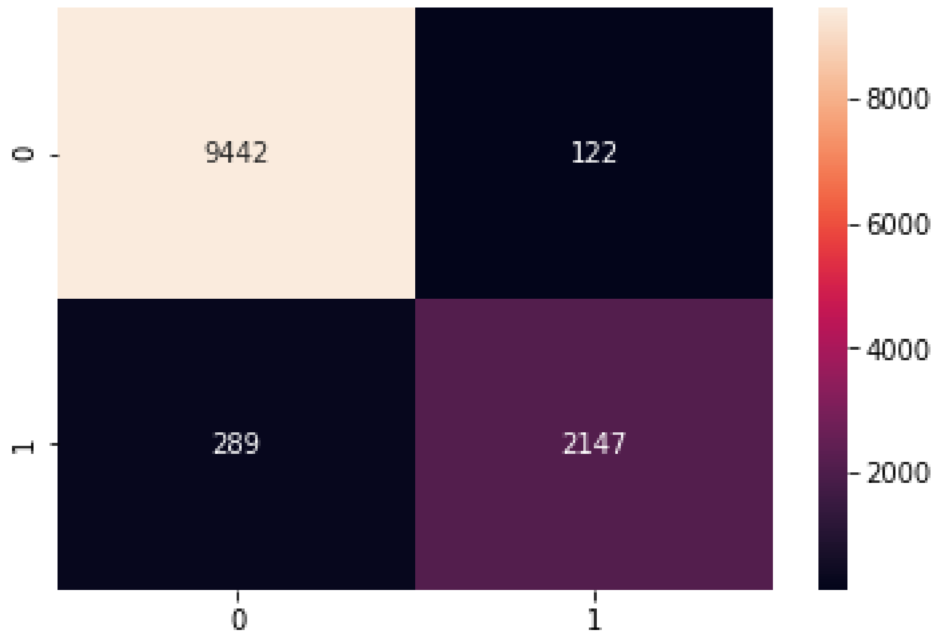
```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98      9564
           1       0.95      0.88      0.91      2436

   micro avg       0.97      0.97      0.97     12000
   macro avg       0.96      0.93      0.95     12000
weighted avg       0.97      0.97      0.97     12000

[[9442  122]
 [ 289 2147]]
```

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x10a170b38>




```
In [30]: floatlist = train.loc[:, train.dtypes == float].columns.tolist()
         numcat = rf_cv.best_estimator_.named_steps['preprocessor'].named_transformers_
         bb=numcat['cat'].named_steps["onehot"].get_feature_names()
         feature = np.concatenate((floatlist,bb))
         print(feature.shape)
         #feature
```

```
(126,)
```

```
In [31]: pcaname = rf_cv.best_estimator_.named_steps['pca']
         #pcaname.explained_variance_ratio_
```

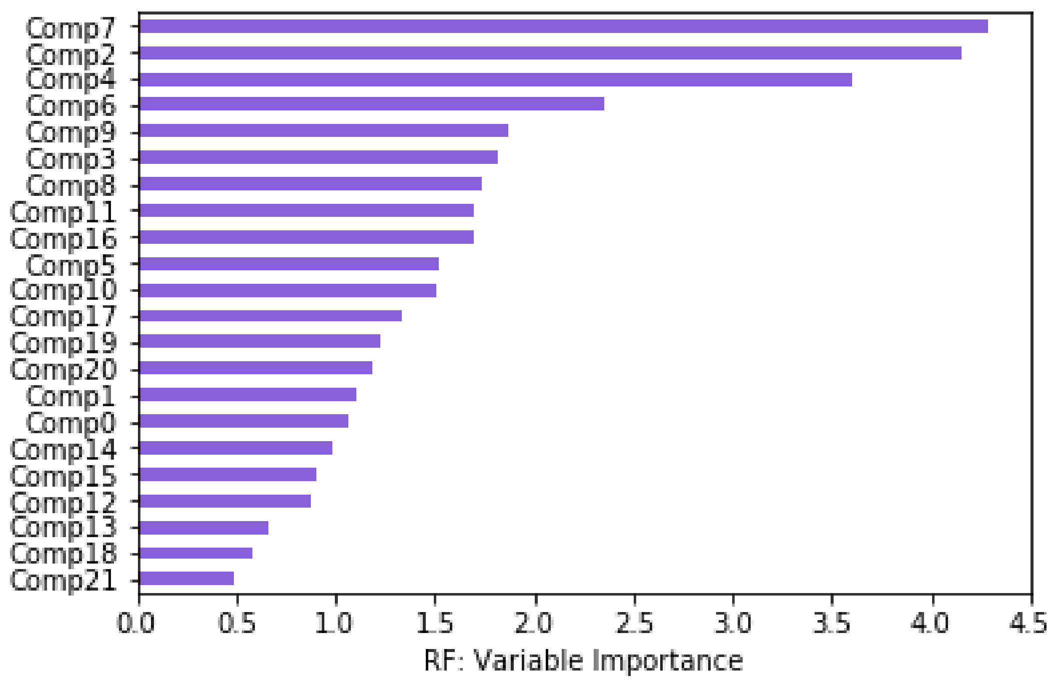
```
In [32]: a=[];
         for i in range(110):
             a.append( "Comp" + str(i))
         comp = pd.DataFrame(pcaname.components_, columns=feature, index=a)
         c7 = comp.loc['Comp7']#
         ind = c7.abs().sort_values(ascending = False).index.tolist()
         c7.loc[ind].to_csv('out1.csv')
```

```
In [33]: a=[];
         for i in range(22):
             a.append( "Comp" + str(i))
         #type(a)
```

```
In [34]: RF = rf_cv.best_estimator_.named_steps['classifier']
```

```
In [35]: plt.figure(figsize=(110,120))
         Importance = pd.DataFrame({'Importance':RF.feature_importances_[0:22]*100},
                                   index = a)
         Importance.sort_values(
             'Importance', axis=0, ascending=True).plot(kind='barh', color='#875FDB')
         plt.xlabel('RF: Variable Importance')
         plt.gca().legend_ = None
         plt.savefig('RF')
```

```
<Figure size 7920x8640 with 0 Axes>
```




```
In [36]: testy_rf_pred_proba = rf_cv.predict_proba(testdata)[:,1]
         pd.DataFrame(testy_rf_pred_proba).to_csv('resultpcarfsmote.csv')
```

12 Building Model 2: Gradient Boosting Machine

I used pipelines to build Gradient Boosting Machine. I first created the preprocessing pipelines for both numerical and categorical data. For numerical data, I imputed the missing data with median and standardized the data. For categorial data, I imputed missing value as the most frequent value and created dummy variable (ignoring the missing data). I split the data into 70% training data and 30% test data. I build Gradient Boosting Machine and used grid search on hyperparameter learning rate, maximum depth of the individual estimators, number of features to consider when looking for the best split, with 5-fold cross-validation.The best performing GBM uses 0.2 learning rate, maximum depth as 8, and the square root of the total number of features as the number of features to consider when looking for the best split. The accuracy score is 0.953, and the ROC AUC score is 0.9834.

In [37]:

```
# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = train.loc[:, train.dtypes == float].columns.tolist()
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['x34', 'x35', 'x68','x93']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
gbm= GradientBoostingClassifier()
clf = Pipeline(steps=[('preprocessor', preprocessor),
                     # ('sampling', SMOTE()),
                      ('classifier', gbm)])


#clf.fit(X_train, y_train)
param_grid = {
   # "loss":["deviance"],
    "classifier__learning_rate": [0.01, 0.075, 0.2],
    #"min_samples_split": np.linspace(0.1, 0.5, 12),
    #"min_samples_leaf": np.linspace(0.1, 0.5, 12),
    "classifier__max_depth":[3,8],
    "classifier__max_features":["log2","sqrt"],
    #"criterion": ["friedman_mse",  "mae"],
    #"subsample":[0.5, 0.618, 0.8, 0.85, 0.9, 0.95, 1.0],
    #"n_estimators":[10]
    }

rf_cv = GridSearchCV(clf, param_grid, cv = 5)


 rf_cv.fit(x_train, y_train.values.ravel())
```

```
y_pred_proba = rf_cv.predict_proba(x_test)[:,1]
# Evaluate test-set roc_auc_score
rf_roc_auc = roc_auc_score(y_test, y_pred_proba)

print("model score: %.3f" % rf_cv.score(x_test, y_test))
# Print roc_auc_score
print('ROC AUC score: {:.4f}'.format(rf_roc_auc))
print("Tuned Random Forest Parameter: {}".format(rf_cv.best_params_))
print("Tuned Random Forest Accuracy: {}".format(rf_cv.best_score_))
```

```
model score: 0.953
ROC AUC score: 0.9834
Tuned GBM Parameter: {'classifier__learning_rate': 0.2, 'classifier__max_depth': 8,
 'classifier__max_features': 'sqrt'}
Tuned GBM Accuracy: 0.9504285714285714
```

In [38]:
```
predictions_rf = rf_cv.predict(x_test)

# Classification Report of Prediction
print("Classification Report:")
print(classification_report(y_test, predictions_rf))
# Confusion Matrix for predictions made
conf2 = confusion_matrix(y_test,predictions_rf)
print(conf2)
# Plot Confusion Matrix
label = ["0","1"]
sns.heatmap(conf2, annot=True, xticklabels=label, yticklabels=label)
```

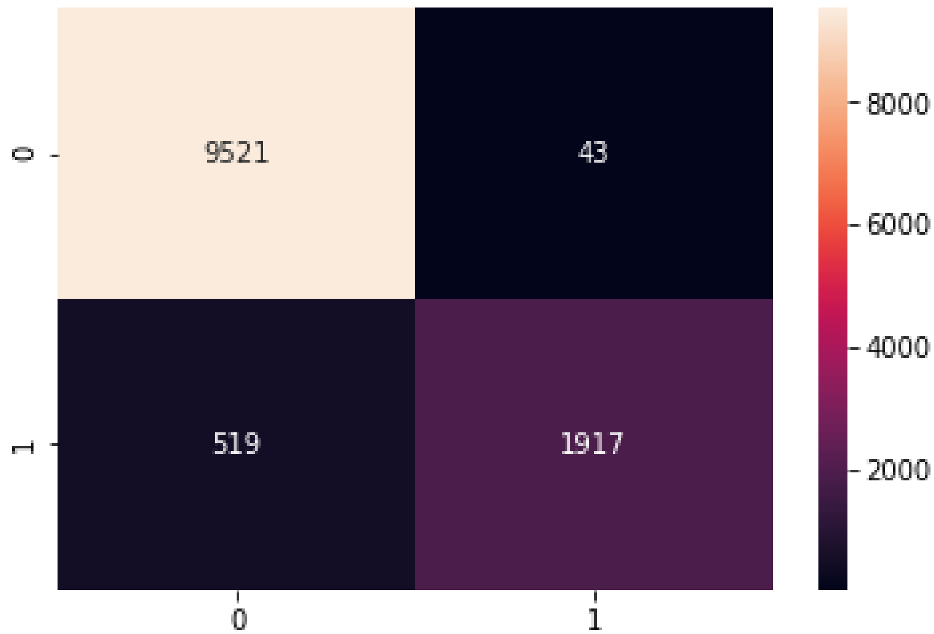
```
Classification Report:
              precision    recall  f1-score   support

           0       0.95      1.00      0.97      9564
           1       0.98      0.79      0.87      2436

   micro avg       0.95      0.95      0.95     12000
   macro avg       0.96      0.89      0.92     12000
weighted avg       0.95      0.95      0.95     12000

[[9521   43]
 [ 519 1917]]
```

Out[38]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a0af2a2b0>
```



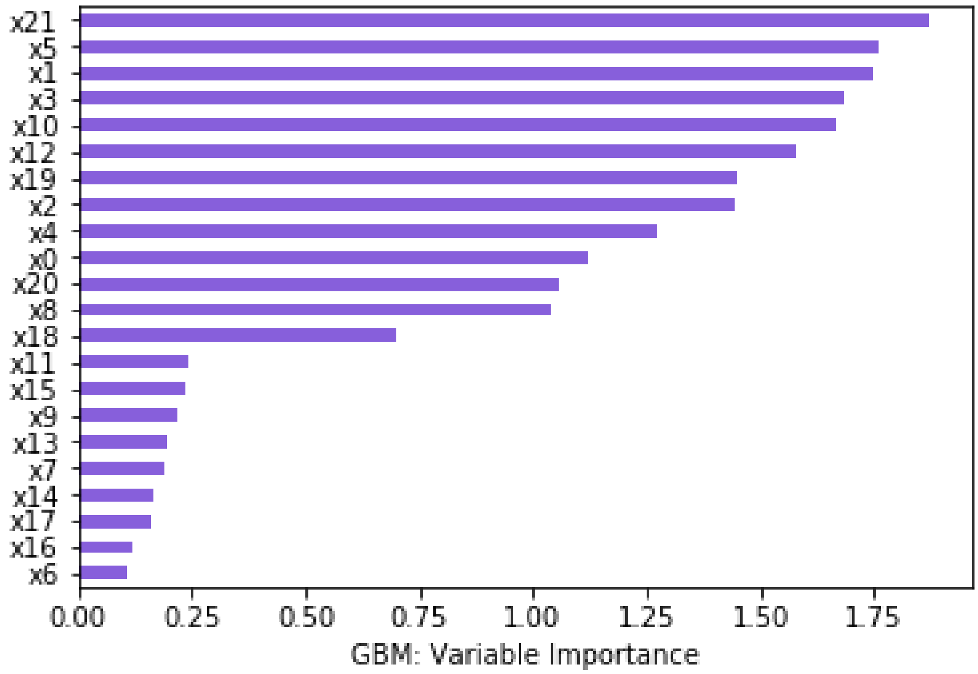

```
In [39]: floatlist = train.loc[:, train.dtypes == float].columns.tolist()
         numcat = rf_cv.best_estimator_.named_steps['preprocessor'].named_transformers_
         bb=numcat['cat'].named_steps["onehot"].get_feature_names()
         feature = np.concatenate((floatlist,bb))
         print(feature.shape)
```

```
(126,)
```

```
In [40]: RF = rf_cv.best_estimator_.named_steps['classifier']
```

```
In [41]: plt.figure(figsize=(110,120))
         Importance = pd.DataFrame({'Importance':RF.feature_importances_[0:22]*100},
                                   index = feature[0:22])
         Importance.sort_values(
             'Importance', axis=0, ascending=True).plot(kind='barh', color='#875FDB')
         plt.xlabel('GBM: Variable Importance')
         plt.gca().legend_ = None
         plt.savefig('RBM')
```

```
<Figure size 7920x8640 with 0 Axes>
```



In [42]:
```
testy_rf_pred_proba = rf_cv.predict_proba(testdata)[:,1]
pd.DataFrame(testy_rf_pred_proba).to_csv('resultrgbm.csv')
```

In [43]:
```
# KS test for training and holdout data
from scipy import stats
numeric_features = train.loc[:, train.dtypes == float].columns.tolist()
trainnum = train[numeric_features]
testnum = testdata[numeric_features]
pv = np.empty([len(trainnum.columns), 1])
t = 0
for column in trainnum:
    pv[t]=stats.ks_2samp(trainnum[column], testnum[column]).pvalue
    t = t+1
```

13 Appendix

13.1 Random Forest

I used pipelines to build the Random Forest (RF) classifier. I first created the preprocessing pipelines for both numerical and categorical data. For numerical data, I imputed the missing data with a median and standardized the data. For categorical data, I imputed missing value with the most frequent values and created dummy variable. I used Synthetic Minority Over-sampling (SMOTE) to create new cases make the classification categories equally represented. I build the RF classifier, tuning hyper-parameters with grid search on the number of trees, quality of split criteria, number of features to consider when looking for the best split, with 5-fold cross-validation. The best performing RF uses 500 trees, entropy as split criteria, and the square root of the total number of features as the number of features to consider when looking for the best split. The accuracy score is 0.925, and the ROC AUC score is 0.9752.

In [31]:
```
from imblearn.pipeline import Pipeline

# We create the preprocessing pipelines for both numeric and categorical data.
numeric_features = train.loc[:, train.dtypes == float].columns.tolist()
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['x34', 'x35', 'x68','x93']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
os = SMOTENC(categorical_features=[34,35,68,93],random_state=0)
rf = RandomForestClassifier()
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('sampling', SMOTE()),
                      ('classifier', rf)])

param_grid = {
    'classifier__n_estimators': [200, 500],
   'classifier__max_features': ['sqrt', 'log2'],
    'classifier__criterion' :['gini', 'entropy']
}

rf_cv = GridSearchCV(clf, param_grid, cv = 5)
```

```
rf_cv.fit(x_train, y_train.values.ravel())



y_pred_proba = rf_cv.predict_proba(x_test)[:,1]
# Evaluate test-set roc_auc_score
rf_roc_auc = roc_auc_score(y_test, y_pred_proba)

print("model score: %.3f" % rf_cv.score(x_test, y_test))
# Print roc_auc_score
print('ROC AUC score: {:.4f}'.format(rf_roc_auc))
print("Tuned Random Forest Parameter: {}".format(rf_cv.best_params_))
print("Tuned Random Forest Accuracy: {}".format(rf_cv.best_score_))
```

```
model score: 0.925
ROC AUC score: 0.9752
Tuned Random Forest Parameter: {'classifier__criterion': 'entropy', 'classifier__max_features': 'sqrt',
'classifier__n_estimators': 500}
Tuned Random Forest Accuracy: 0.9211785714285714
```

In [32]:
```
predictions_rf = rf_cv.predict(x_test)

# Classification Report of Prediction
print("Classification Report:")
print(classification_report(y_test, predictions_rf))
# Confusion Matrix for predictions made
conf2 = confusion_matrix(y_test,predictions_rf)
print(conf2)
# Plot Confusion Matrix
label = ["0","1"]
sns.heatmap(conf2, annot=True, xticklabels=label, yticklabels=label)
```

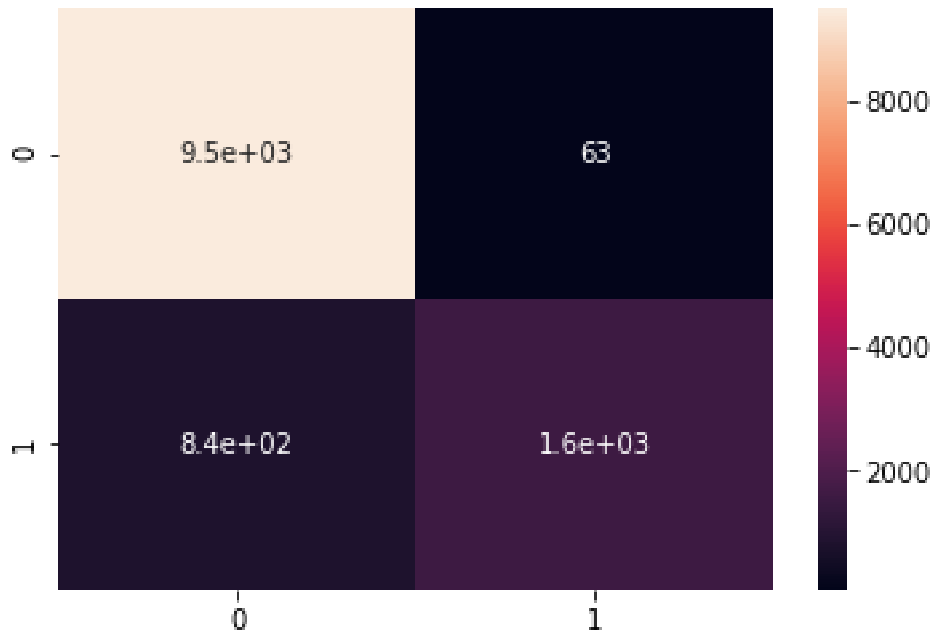
```
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.99      0.95      9564
           1       0.96      0.66      0.78      2436

   micro avg       0.93      0.93      0.93     12000
   macro avg       0.94      0.83      0.87     12000
weighted avg       0.93      0.93      0.92     12000

[[9501   63]
 [ 836 1600]]
```

Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x10490d518>



In [33]:
```
floatlist = train.loc[:, train.dtypes == float].columns.tolist()
numcat = rf_cv.best_estimator_.named_steps['preprocessor'].named_transformers_
bb=numcat['cat'].named_steps["onehot"].get_feature_names()
feature = np.concatenate((floatlist,bb))
print(feature.shape)
feature
```

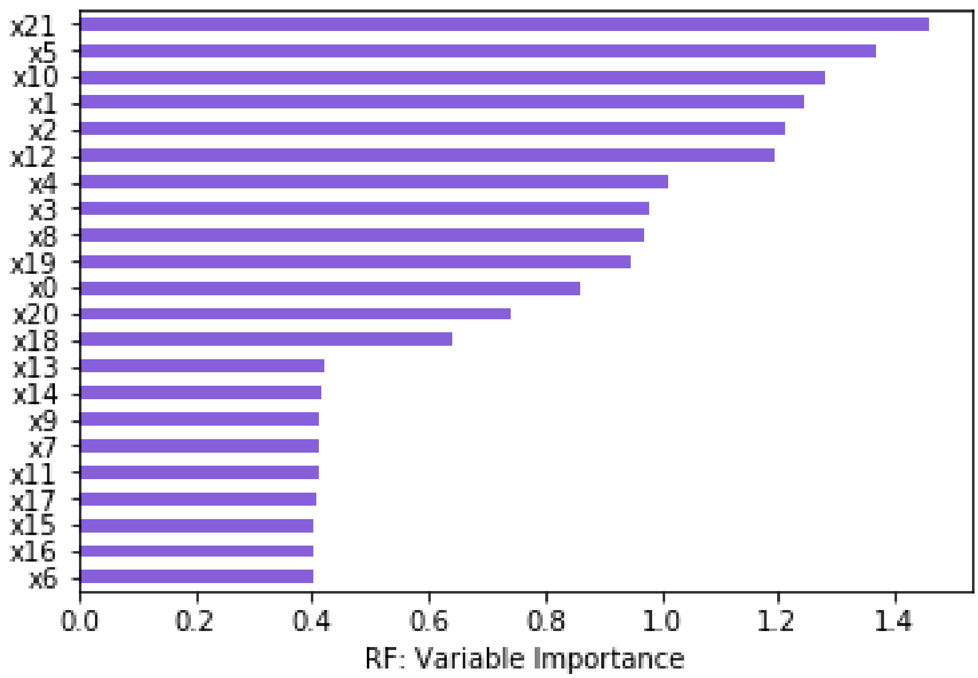
(126,)

Out[33]:
```
array(['x0', 'x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10',
       'x11', 'x12', 'x13', 'x14', 'x15', 'x16', 'x17', 'x18', 'x19',
       'x20', 'x21', 'x22', 'x23', 'x24', 'x25', 'x26', 'x27', 'x28',
       'x29', 'x30', 'x31', 'x32', 'x33', 'x36', 'x37', 'x38', 'x39',
       'x40', 'x41', 'x42', 'x43', 'x44', 'x45', 'x46', 'x47', 'x48',
       'x49', 'x50', 'x51', 'x52', 'x53', 'x54', 'x55', 'x56', 'x57',
       'x58', 'x59', 'x60', 'x61', 'x62', 'x63', 'x64', 'x65', 'x66',
       'x67', 'x69', 'x70', 'x71', 'x72', 'x73', 'x74', 'x75', 'x76',
       'x77', 'x78', 'x79', 'x80', 'x81', 'x82', 'x83', 'x84', 'x85',
       'x86', 'x87', 'x88', 'x89', 'x90', 'x91', 'x92', 'x94', 'x95',
       'x96', 'x97', 'x98', 'x99', 'x0_Honda', 'x0_Toyota', 'x0_bmw',
       'x0_chevrolet', 'x0_chrystler', 'x0_ford', 'x0_mercades',
       'x0_nissan', 'x0_tesla', 'x0_volkswagon', 'x1_friday', 'x1_monday',
       'x1_thursday', 'x1_tuesday', 'x1_wednesday', 'x2_Apr', 'x2_Aug',
       'x2_Dec', 'x2_Feb', 'x2_Jan', 'x2_Jul', 'x2_Jun', 'x2_Mar',
       'x2_May', 'x2_Nov', 'x2_Oct', 'x2_Sep', 'x3_america', 'x3_asia',
       'x3_euorpe'], dtype=object)
```

```
In [34]: RF = rf_cv.best_estimator_.named_steps['classifier']

In [35]: plt.figure(figsize=(110,120))
         Importance = pd.DataFrame({'Importance':RF.feature_importances_[0:22]*100},
                                   index = feature[0:22])
         Importance.sort_values(
             'Importance', axis=0, ascending=True).plot(kind='barh', color='#875FDB')
         plt.xlabel('RF: Variable Importance')
         plt.gca().legend_ = None
         plt.savefig('RF')
```

```
<Figure size 7920x8640 with 0 Axes>
```




```
In [36]: testy_rf_pred_proba = rf_cv.predict_proba(testdata)[:,1]
         pd.DataFrame(testy_rf_pred_proba).to_csv('resultrfsmote.csv')
```

13.2 K-Nearest Neighbors

I used pipelines to build the K-Nearest Neighbors (KNN) classifier. I first created the preprocessing pipelines for both numerical and categorical data. For numerical data, I imputed the missing data with a median and standardized the data. For categorical data, I imputed missing value with the most frequent values and created dummy variable. I build the KNN classifier, tuning hyper-parameters with grid search on the number of neighbors and metric type with 5-fold cross-validation. The best performing KNN uses five neighbors and Euclidean distance. The accuracy score is 0.9263, and the ROC AUC score is 0.9597. When using SMOTE, the accuracy score is , and the ROC AUC score is .

In [28]:
```
# Create the preprocessing pipelines for both numeric and categorical data.
numeric_features = floatlist
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['x34', 'x35', 'x68','x93']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
knn = KNeighborsClassifier()
 clf = Pipeline(steps=[('preprocessor',    preprocessor),
                      #('sampling', SMOTE()), #use SMOTE or not
                      ('classifier', knn)])


param_grid = [{'classifier__n_neighbors': [5,10, 15, 20, 25],
               "classifier__metric": ["euclidean", "cityblock"]
                   }]
knn_cv = GridSearchCV(clf, param_grid, cv = 5)

knn_cv.fit(x_train, y_train)

predictions_knn = knn_cv.predict(x_test)
y_pred_proba = knn_cv.predict_proba(x_test)[:,1]
# Evaluate test-set roc_auc_score
knn_roc_auc = roc_auc_score(y_test, y_pred_proba)

# Print roc_auc_score
print('ROC AUC score: {:.4f}'.format(knn_roc_auc))
print("Tuned KNN Parameter: {}".format(knn_cv.best_params_))
print("Tuned KNN Accuracy: {}".format(knn_cv.best_score_))
```

```
ROC AUC score: 0.9611
Tuned KNN Parameter: {'classifier__metric': 'euclidean', 'classifier__n_neighbors': 5}
Tuned KNN Accuracy: 0.9216785714285715
```

In [29]:
```
predictions_knn = knn_cv.predict(X_test)

# Classification Report of Prediction
print("Classification Report:")
print(classification_report(y_test, predictions_knn))
# Confusion Matrix for predictions made
conf2 = confusion_matrix(y_test,predictions_knn)
print(conf2)
# Plot Confusion Matrix
label = ["0","1"]
sns.heatmap(conf2, annot=True, xticklabels=label, yticklabels=label)
```

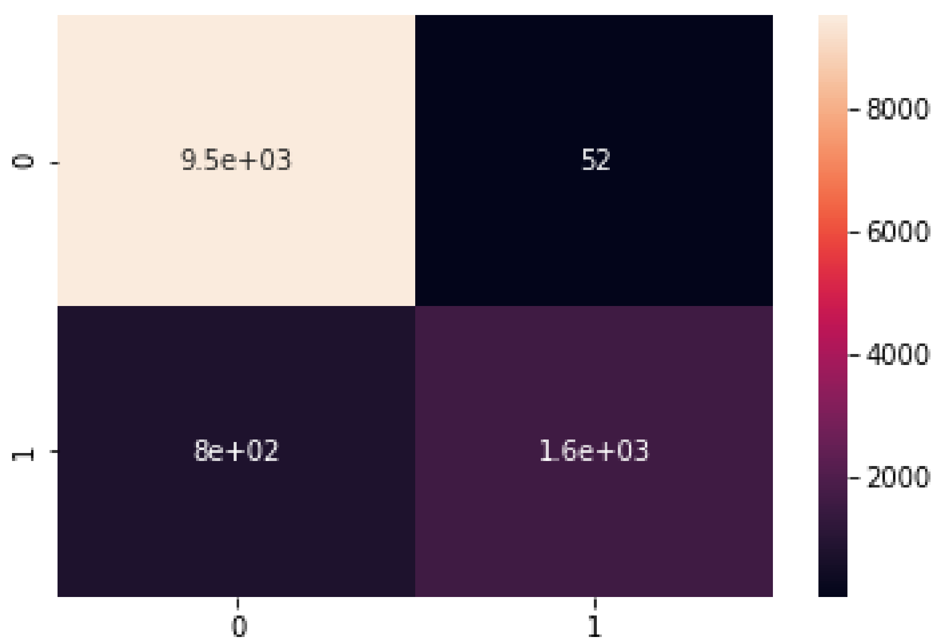
```
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.99      0.96      9564
           1       0.97      0.67      0.79      2436

   micro avg       0.93      0.93      0.93     12000
   macro avg       0.95      0.83      0.88     12000
weighted avg       0.93      0.93      0.92     12000

[[9512   52]
 [ 801 1635]]
```

Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x1131db080>



```
In [30]: testy_logreg_pred_proba = knn_cv.predict_proba(testdata)[:,1]
         pd.DataFrame(testy_logreg_pred_proba).to_csv('resultknnfinal.csv')
```

13.3 KNN with PCA

I used pipelines to build the logistic regression. I build KNN after applying PCA and used grid search on hyperparameter number of neighbors for KNN and number of components for PCA with 5-fold cross-validation.

```
In [8]: # Create the preprocessing pipelines for both numeric and categorical data.
        numeric_features = floatlist
        numeric_transformer = Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='median')),
            ('scaler', StandardScaler())])

        categorical_features = ['x34', 'x35', 'x68','x93']
        categorical_transformer = Pipeline(steps=[
            ('imputer', SimpleImputer(strategy='most_frequent')),
            ('onehot', OneHotEncoder(handle_unknown='ignore'))])

        preprocessor = ColumnTransformer(
            transformers=[
                ('num', numeric_transformer, numeric_features),
                ('cat', categorical_transformer, categorical_features)])

        # Append classifier to preprocessing pipeline.
        # Now we have a full prediction pipeline.
        knn = KNeighborsClassifier()
        pca = PCA()
        clf = Pipeline(steps=[('preprocessor', preprocessor),
                              ('pca', pca),
                              ('classifier', knn)])
        X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3, stratify = y, random_state = 42)

        param_grid = [{'pca__n_components': [ 10, 50, 80,100,110, 124],
            'classifier__n_neighbors': [5, 10, 20]}]
        knn_cv = GridSearchCV(clf, param_grid, cv = 2)

        knn_cv.fit(X_train, y_train)

        predictions_knn = knn_cv.predict(X_test)
        y_pred_proba = knn_cv.predict_proba(X_test)[:,1]
        # Evaluate test-set roc_auc_score
        knn_roc_auc = roc_auc_score(y_test, y_pred_proba)

        # Print roc_auc_score
        print('ROC AUC score: {:.4f}'.format(knn_roc_auc))
        #print("model score: %.3f" % clf.score(X_test, y_test))
```

```
#lg = logreg_cv.best_estimator_.named_steps['classifier']
#print("The Number of Active Features is : {}".format(sum(lg.coef_ == 0).sum()))
print("Tuned Logistic Regression Parameter: {}".format(knn_cv.best_params_))
print("Tuned Logistic Regression Accuracy: {}".format(knn_cv.best_score_))
```

```
ROC AUC score: 0.9611
Tuned Logistic Regression Parameter: {'classifier__n_neighbors': 5, 'pca__n_components': 124}
Tuned Logistic Regression Accuracy: 0.9147857142857143
```

13.4 Logistic Regression with L1

I used pipelines to build the logistic regression. I build logistic regression with L1 regularization and used grid search on hyperparameter C with 5-fold cross-validation.

In [46]:
```
# Create the preprocessing pipelines for both numeric and categorical data.
numeric_features = train.loc[:, train.dtypes == float].columns.tolist()
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    #('imputer',interpolate(method='linear', inplace=True, limit_direction="both"))
    ('scaler', StandardScaler())])

categorical_features = ['x34', 'x35', 'x68','x93']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')), #,
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
logreg = LogisticRegression(penalty = 'l1', solver = 'liblinear', max_iter = 500)
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('classifier', logreg)])
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
                                                    stratify = y, random_state = 42)

#clf.fit(X_train, y_train)

c_space = np.logspace(-5, 8, 15)
param_grid = {'classifier__C': c_space} #{'classifier__C': c_space}
logreg_cv = GridSearchCV(clf, param_grid, cv = 5)

logreg_cv.fit(X_train, y_train)

y_pred_proba = logreg_cv.predict_proba(X_test)[:,1]
```

```
# Evaluate test-set roc_auc_score
logreg_roc_auc = roc_auc_score(y_test, y_pred_proba)

# Print roc_auc_score
print('ROC AUC score: {:.4f}'.format(logreg_roc_auc))
print("model score: %.3f" % logreg_cv.score(X_test, y_test))
lg = logreg_cv.best_estimator_.named_steps['classifier']
print("The Number of Active Features is : {}".format(sum(lg.coef_ != 0).sum()))
print("Tuned Logistic Regression Parameter: {}".format(logreg_cv.best_params_))
print("Tuned Logistic Regression Accuracy: {}".format(logreg_cv.best_score_))
```

```
ROC AUC score: 0.9088
model score: 0.891
The Number of Active Features is : 106
Tuned Logistic Regression Parameter: {'classifier__C': 0.4393970560760795}
Tuned Logistic Regression Accuracy: 0.8904285714285715
```

13.5 Logistic Regression with Recursive Feature Elimination

I used pipelines to build the logistic regression. I first created the preprocessing pipelines for both numerical and categorical data. For numerical data, I imputed the missing data with median and standardized the data. For categorial data, I imputed missing value as missing and created dummy variable (ignoring the missing data). I split the data into 70% training data and 30% test data. I build logistic regression with Recursive Feature Elimination (RFE) and used grid search on hyperparameter C for logistic regression and number of features for RFE with 5-fold cross-validation.

In [ ]:
```
# Create the preprocessing pipelines for both numeric and categorical data.
numeric_features = train.loc[:, train.dtypes == float].columns.tolist()
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())])

categorical_features = ['x34', 'x35', 'x68','x93']
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant', fill_value='missing')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
logreg = LogisticRegression(solver = 'lbfgs',max_iter = 300)
rfe = RFE(logreg)
```

```
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      ('rfe', rfe),
                      ('classifier', logreg)
                     ])
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
                                   stratify = y, random_state = 42)


c_space = np.logspace(-5, 8, 15)
param_grid = {'classifier__C': [.01, .5],
          'rfe__n_features_to_select':[ 30, 50]}
logreg_cv = GridSearchCV(clf, param_grid, cv = 5)

logreg_cv.fit(X_train, y_train)

y_pred_proba = logreg_cv.predict_proba(X_test)[:,1]
# Evaluate test-set roc_auc_score
logreg_roc_auc = roc_auc_score(y_test, y_pred_proba)

# Print roc_auc_score
print('ROC AUC score: {:.4f}'.format(logreg_roc_auc))
#print("model score: %.3f" % logreg_cv.score(X_test, y_test))
lg = logreg_cv.best_estimator_.named_steps['classifier']
print("The Number of Active Features is : {}".format(sum(lg.coef_ != 0).sum()))
print("Tuned Logistic Regression Parameter: {}".format(logreg_cv.best_params_))
print("Tuned Logistic Regression Accuracy: {}".format(logreg_cv.best_score_))
```

```
ROC AUC score: 0.9069
The Number of Active Features is : 50
Tuned Logistic Regression Parameter: {'classifier__C': 0.5, 'rfe__n_features_to_select': 50}
Tuned Logistic Regression Accuracy: 0.8896832773289484
```

13.6 PCA + Logistic Regression

I first applied PCA to the data and build logistic regression on the transformed features from PCA, and used grid search on hyperparameter C for logistic regression and number of components for PCA with 5-fold cross-validation. Uncomment the SMOTE commands in the pipeline will use SMOTE in the analysis.

```
In [11]: # Create the preprocessing pipelines for both numeric and categorical data.
         numeric_features = floatlist
         numeric_transformer = Pipeline(steps=[
             ('imputer', SimpleImputer(strategy='median')),
             ('scaler', StandardScaler())])

         categorical_features = ['x34', 'x35', 'x68','x93']
         categorical_transformer = Pipeline(steps=[
```

```
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))])

preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features)])

# Append classifier to preprocessing pipeline.
# Now we have a full prediction pipeline.
logistic = SGDClassifier(loss='log', penalty='l2', early_stopping=True,
                         max_iter=10000, tol=1e-5, random_state=0)

pca = PCA()
clf = Pipeline(steps=[('preprocessor', preprocessor),
                      #('sampling', SMOTE()), #use SMOTE or not
                      ('pca', pca),
                      ('classifier', logistic)])
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.3,
                                                    stratify = y, random_state = 42)

param_grid = [{'pca__n_components': [50, 80, 90, 100],
    'classifier__alpha': np.logspace(-4, 4, 5)}]#n
knn_cv = GridSearchCV(clf, param_grid, cv = 2)

knn_cv.fit(X_train, y_train)

predictions_knn = knn_cv.predict(X_test)
y_pred_proba = knn_cv.predict_proba(X_test)[:,1]
# Evaluate test-set roc_auc_score
knn_roc_auc = roc_auc_score(y_test, y_pred_proba)

# Print roc_auc_score
print('ROC AUC score: {:.4f}'.format(knn_roc_auc))
#print("model score: %.3f" % clf.score(X_test, y_test))
#lg = logreg_cv.best_estimator_.named_steps['classifier']
#print("The Number of Active Features is : {}".format(sum(lg.coef_ == 0).sum()))
print("Tuned Logistic Regression Parameter: {}".format(knn_cv.best_params_))
print("Tuned Logistic Regression Accuracy: {}".format(knn_cv.best_score_))
```

```
ROC AUC score: 0.9036
Tuned Logistic Regression Parameter: {'classifier__alpha': 0.01, 'pca__n_components': 100}
Tuned Logistic Regression Accuracy: 0.8848214285714285
```