capstone-credit card fraud detection

May 2, 2019

1 Introduction

The datasets contains transactions made by credit cards in September 2013 by european cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data. Features V1, V2, … V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-senstive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

```
In [1]: ## Using IBM Cloud

        # import types
        # import pandas as pd
        # from botocore.client import Config
        # import ibm_boto3

        # def __iter__(self): return 0

        # # @hidden_cell
        # # The following code accesses a file in your IBM Cloud Object Storage. It includes y
        # # You might want to remove those credentials before you share your notebook.
        # client_dcc662e2032147378976f288b5e93fcc = ibm_boto3.client(service_name='s3',
        #     ibm_api_key_id='pGKI3UAgcg9d2BDY9-eXqh5gO0qv-R-AJryFg9j6-0gJ',
        #     ibm_auth_endpoint="https://iam.bluemix.net/oidc/token",
        #     config=Config(signature_version='oauth'),
        #     endpoint_url='https://s3-api.us-geo.objectstorage.service.networklayer.com')

        # body = client_dcc662e2032147378976f288b5e93fcc.get_object(Bucket='default-donotdelet
        # # add missing __iter__ method, so pandas accepts body as file-like object
        # if not hasattr(body, "__iter__"): body.__iter__ = types.MethodType( __iter__, body )
```

```
# df = pd.read_csv(body)
# df.head()
```

Out[1]:
```
   Time        V1        V2        V3        V4        V5        V6        V7  \
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

         V8        V9  ...         V21       V22       V23       V24  \
0  0.098698  0.363787  ...   -0.018307  0.277838 -0.110474  0.066928
1  0.085102 -0.255425  ...   -0.225775 -0.638672  0.101288 -0.339846
2  0.247676 -1.514654  ...    0.247998  0.771679  0.909412 -0.689281
3  0.377436 -1.387024  ...   -0.108300  0.005274 -0.190321 -1.175575
4 -0.270533  0.817739  ...   -0.009431  0.798278 -0.137458  0.141267

        V25       V26       V27       V28  Amount  Class
0  0.128539 -0.189115  0.133558 -0.021053  149.62      0
1  0.167170  0.125895 -0.008983  0.014724    2.69      0
2 -0.327642 -0.139097 -0.055353 -0.059752  378.66      0
3  0.647376 -0.221929  0.062723  0.061458  123.50      0
4 -0.206010  0.502292  0.219422  0.215153   69.99      0

[5 rows x 31 columns]
```

In [1]:
```python
# running on local machine
import pandas as pd
df = pd.read_csv('creditcardfraud/creditcard.csv')
```

In [2]:
```python
# it has 28 features plus time and amount variables
df.columns
```

Out[2]:
```
Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10',
       'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20',
       'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount',
       'Class'],
      dtype='object')
```

In [3]:
```python
# take a look at the statistics of these features
df.describe()
```

Out[3]:
```
                Time            V1            V2            V3            V4  \
count  284807.000000  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean    94813.859575  3.919560e-15  5.688174e-16 -8.769071e-15  2.782312e-15
std     47488.145955  1.958696e+00  1.651309e+00  1.516255e+00  1.415869e+00
min         0.000000 -5.640751e+01 -7.271573e+01 -4.832559e+01 -5.683171e+00
25%     54201.500000 -9.203734e-01 -5.985499e-01 -8.903648e-01 -8.486401e-01
50%     84692.000000  1.810880e-02  6.548556e-02  1.798463e-01 -1.984653e-02
```

```
75%    139320.500000  1.315642e+00  8.037239e-01  1.027196e+00  7.433413e-01
max    172792.000000  2.454930e+00  2.205773e+01  9.382558e+00  1.687534e+01

                 V5            V6            V7            V8            V9  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean  -1.552563e-15  2.010663e-15 -1.694249e-15 -1.927028e-16 -3.137024e-15
std    1.380247e+00  1.332271e+00  1.237094e+00  1.194353e+00  1.098632e+00
min   -1.137433e+02 -2.616051e+01 -4.355724e+01 -7.321672e+01 -1.343407e+01
25%   -6.915971e-01 -7.682956e-01 -5.540759e-01 -2.086297e-01 -6.430976e-01
50%   -5.433583e-02 -2.741871e-01  4.010308e-02  2.235804e-02 -5.142873e-02
75%    6.119264e-01  3.985649e-01  5.704361e-01  3.273459e-01  5.971390e-01
max    3.480167e+01  7.330163e+01  1.205895e+02  2.000721e+01  1.559499e+01

       ...           V21           V22           V23           V24  \
count  ...  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05
mean   ...  1.537294e-16  7.959909e-16  5.367590e-16  4.458112e-15
std    ...  7.345240e-01  7.257016e-01  6.244603e-01  6.056471e-01
min    ... -3.483038e+01 -1.093314e+01 -4.480774e+01 -2.836627e+00
25%    ... -2.283949e-01 -5.423504e-01 -1.618463e-01 -3.545861e-01
50%    ... -2.945017e-02  6.781943e-03 -1.119293e-02  4.097606e-02
75%    ...  1.863772e-01  5.285536e-01  1.476421e-01  4.395266e-01
max    ...  2.720284e+01  1.050309e+01  2.252841e+01  4.584549e+00

                V25           V26           V27           V28         Amount  \
count  2.848070e+05  2.848070e+05  2.848070e+05  2.848070e+05  284807.000000
mean   1.453003e-15  1.699104e-15 -3.660161e-16 -1.206049e-16      88.349619
std    5.212781e-01  4.822270e-01  4.036325e-01  3.300833e-01     250.120109
min   -1.029540e+01 -2.604551e+00 -2.256568e+01 -1.543008e+01       0.000000
25%   -3.171451e-01 -3.269839e-01 -7.083953e-02 -5.295979e-02       5.600000
50%    1.659350e-02 -5.213911e-02  1.342146e-03  1.124383e-02      22.000000
75%    3.507156e-01  2.409522e-01  9.104512e-02  7.827995e-02      77.165000
max    7.519589e+00  3.517346e+00  3.161220e+01  3.384781e+01   25691.160000

               Class
count  284807.000000
mean        0.001727
std         0.041527
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000

[8 rows x 31 columns]
```

In [4]:
```
# see if there is missing values
df.isnull().sum()
```

Out[4]:
```
Time      0
```

```
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

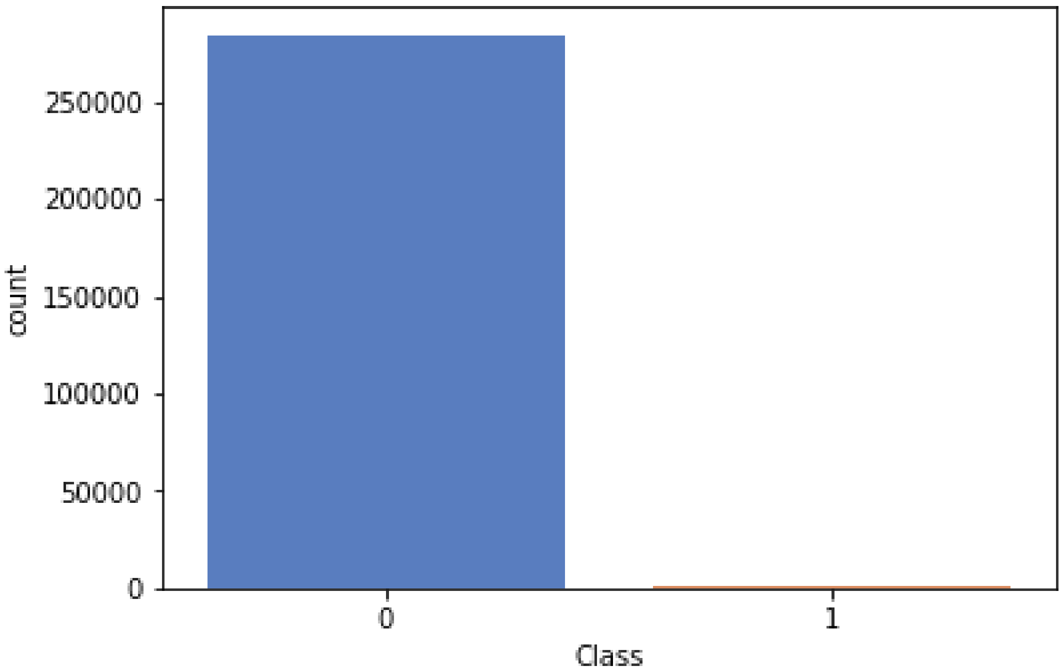
In [5]:
```
tmp = df.Class.value_counts()
print('No frauds: ', tmp[0])
print('Frauds: ', tmp[1])
```

```
No frauds:  284315
Frauds:  492
```

In [6]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
%matplotlib inline
```

In [7]:
```
# visualize the classes distribution
sns.countplot('Class', data=df, palette="muted")
```

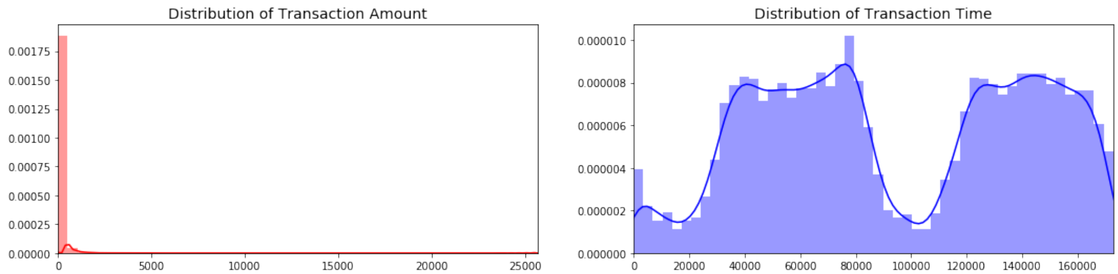
```
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x1a13f5cd3c8>
```



As we can see from the above plot, our original dataset is extremely imbalanced. Most of the transactions are non-fraud. If we use this dataset without any modification to train our classifiers, then we will most certainly overfit.

```
In [8]: # lets also examine the distributions of time and amount features
        fig, ax = plt.subplots(1, 2, figsize=(18,4))

        amount_val = df['Amount'].values
        time_val = df['Time'].values

        sns.distplot(amount_val, ax=ax[0], color='r')
        ax[0].set_title('Distribution of Transaction Amount', fontsize=14)
        ax[0].set_xlim([min(amount_val), max(amount_val)])

        sns.distplot(time_val, ax=ax[1], color='b')
        ax[1].set_title('Distribution of Transaction Time', fontsize=14)
        ax[1].set_xlim([min(time_val), max(time_val)])
```

```
Out[8]: (0.0, 172792.0)
```



As shown in the above plots, both Time and Transaction Amount are needed to be scaled in order to improve model performances.

1.1 Scaling Time and Amount

```
In [9]: from sklearn.preprocessing import RobustScaler
        scaler = RobustScaler() # robust scaler is less prone to outliers
        df['scaled_time'] = scaler.fit_transform(df.Time.values.reshape(-1,1))
        df['scaled_amount'] = scaler.fit_transform(df.Amount.values.reshape(-1,1))
        df.drop(['Time', 'Amount'], axis=1, inplace=True)
```

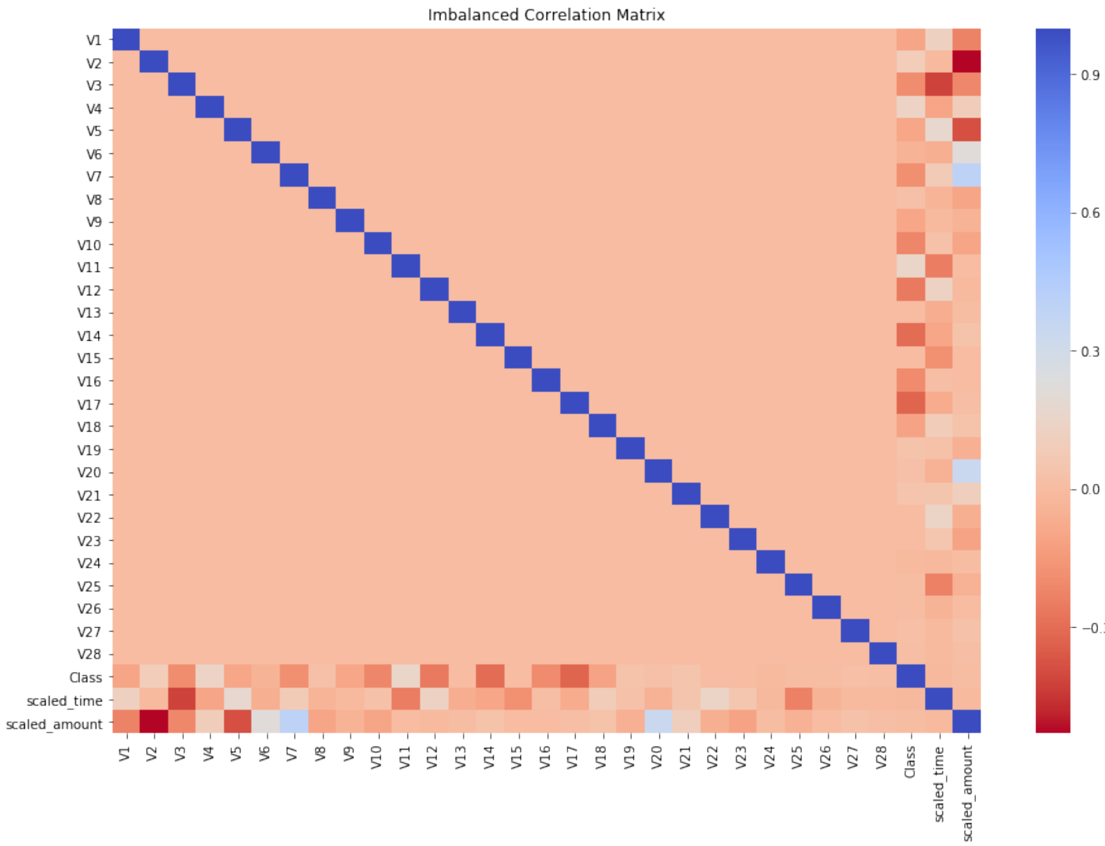
```
In [10]: df.head()
```

```
Out[10]:          V1        V2        V3        V4        V5        V6        V7  \
         0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
         1  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
         2 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
         3 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
         4 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

                  V8        V9       V10  ...       V22       V23       V24       V25  \
         0  0.098698  0.363787  0.090794  ...  0.277838 -0.110474  0.066928  0.128539
         1  0.085102 -0.255425 -0.166974  ... -0.638672  0.101288 -0.339846  0.167170
         2  0.247676 -1.514654  0.207643  ...  0.771679  0.909412 -0.689281 -0.327642
         3  0.377436 -1.387024 -0.054952  ...  0.005274 -0.190321 -1.175575  0.647376
         4 -0.270533  0.817739  0.753074  ...  0.798278 -0.137458  0.141267 -0.206010

                 V26       V27       V28  Class  scaled_time  scaled_amount
         0 -0.189115  0.133558 -0.021053      0    -0.994983       1.783274
         1  0.125895 -0.008983  0.014724      0    -0.994983      -0.269825
         2 -0.139097 -0.055353 -0.059752      0    -0.994972       4.983721
         3 -0.221929  0.062723  0.061458      0    -0.994972       1.418291
         4  0.502292  0.219422  0.215153      0    -0.994960       0.670579

         [5 rows x 31 columns]
```

```
In [11]: # lets take a look at the correlation plot to examine features relation to classes
         plt.figure(figsize=(15,10))
```

```
plt.title("Imbalanced Correlation Matrix")
sns.heatmap(df.corr(), cmap='coolwarm_r', annot_kws={'size':20})
```

Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x1a13f65d630>



1.2 subsample data with equal observation from both classes and explore correlation matrix again

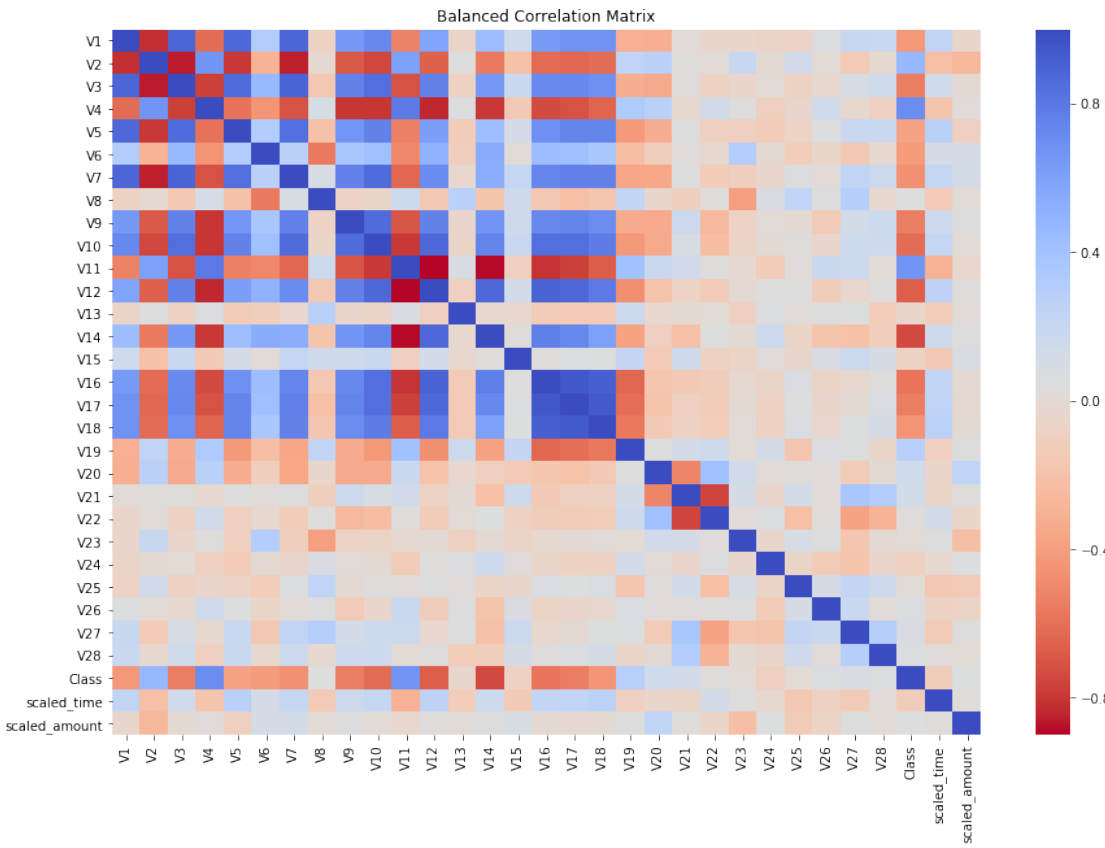
In [12]:
```
# shuffle the data
df = df.sample(frac=1)

frauds = df.loc[df.Class==1]
non_frauds = df.loc[df.Class==0][:429]
new_df = pd.concat([frauds, non_frauds])
new_df = new_df.sample(frac=1, random_state=2019)
new_df.head()
```

Out[12]:
```
              V1        V2        V3        V4        V5        V6        V7  \
140394 -6.961676  3.882633 -2.892660 -0.057403 -2.866139 -0.721680 -2.084298
6472    1.023874  2.001485 -4.769752  3.819195 -1.271754 -1.734662 -3.059245
259002  2.029039 -0.928822  0.048696 -0.724177 -1.359196 -0.319260 -1.304253
```

```
77099  -0.075483  1.812355 -2.566981  4.127549 -1.628532 -0.805895 -3.390135
105980  1.524484 -1.053841  0.497467 -1.442412 -1.524713 -0.646696 -1.110334

              V8        V9       V10  ...       V22       V23       V24  \
140394  3.463034  0.648002  1.615223  ... -0.418317  0.422675  0.056979
6472    0.889805  0.415382 -3.955812  ... -0.054196  0.709654 -0.372216
259002  0.183088  2.312593 -0.422458  ...  0.667772  0.218061 -0.387726
77099   1.019353 -2.451251 -3.555835  ...  0.270471 -0.143624  0.013566
105980 -0.120683 -1.635291  1.425733  ... -0.843161  0.117493 -0.185511

             V25       V26       V27       V28  Class  scaled_time  \
140394  0.820109  0.273678  0.204325 -0.354226      0    -0.011666
6472   -2.032068  0.366778  0.395171  0.020206      1    -0.904052
259002 -0.563044  0.713271 -0.003171 -0.047048      0     0.871967
77099   0.634203  0.213693  0.773625  0.387434      1    -0.326660
105980  0.179495 -0.377980  0.038188  0.024832      0    -0.175319

        scaled_amount
140394       0.156082
6472        -0.293440
259002      -0.293440
77099       -0.237546
105980      -0.033536

[5 rows x 31 columns]
```

In [13]:
```
plt.figure(figsize=(15,10))
plt.title("Balanced Correlation Matrix")
sns.heatmap(new_df.corr(), cmap='coolwarm_r', annot_kws={'size':20})
```

Out[13]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1a141c6fac8>
```



By comparing the two correlation matrix heatmaps, we can see that there is a noticible differences in correlations using balanced and imbalanced datasets, and we can observe more correled relations when using balanced datasets. Therefore, it suggests that we might want to use a more balanced datasets to train robust classifier.

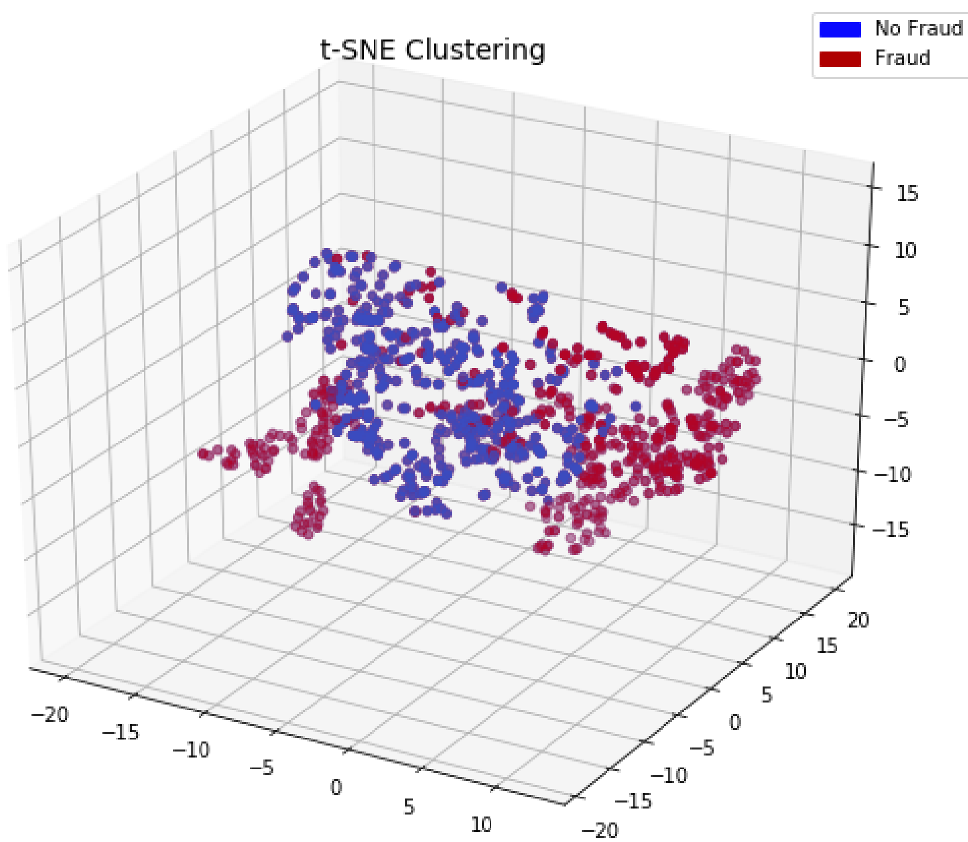
2 TSNE Algorithm to cluster our new balance sample

In [14]:
```
from sklearn.manifold import TSNE
import matplotlib.patches as mpatches
from mpl_toolkits.mplot3d import Axes3D

def plot_tsne(dim=2):
    # New_df is from the random undersample data (fewer instances)
    X = new_df.drop('Class', axis=1)
    y = new_df['Class']
    blue_patch = mpatches.Patch(color='#0A0AFF', label='No Fraud')
    red_patch = mpatches.Patch(color='#AF0000', label='Fraud')

    # T-SNE Implementation
    X_reduced_tsne = TSNE(n_components=dim, random_state=2019).fit_transform(X.values)
    if dim == 3:
```

```
        fig = plt.figure(figsize=(10, 8))
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], X_reduced_tsne[:,2], c=(y
        ax.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], X_reduced_tsne[:,2], c=(y
        ax.set_title('t-SNE Clustering', fontsize=14)
        ax.grid(True)
        ax.legend(handles=[blue_patch, red_patch])
    elif dim == 2:
        plt.figure(figsize=(12,8))
        plt.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], c=(y == 0), cmap='coolwa
        plt.scatter(X_reduced_tsne[:,0], X_reduced_tsne[:,1], c=(y == 1), cmap='coolwa
        plt.title('t-SNE Clustring', fontsize=14)
        plt.grid(True)
        plt.legend(handles=[blue_patch, red_patch])
    else:
        return print('invalid input dimension')
```

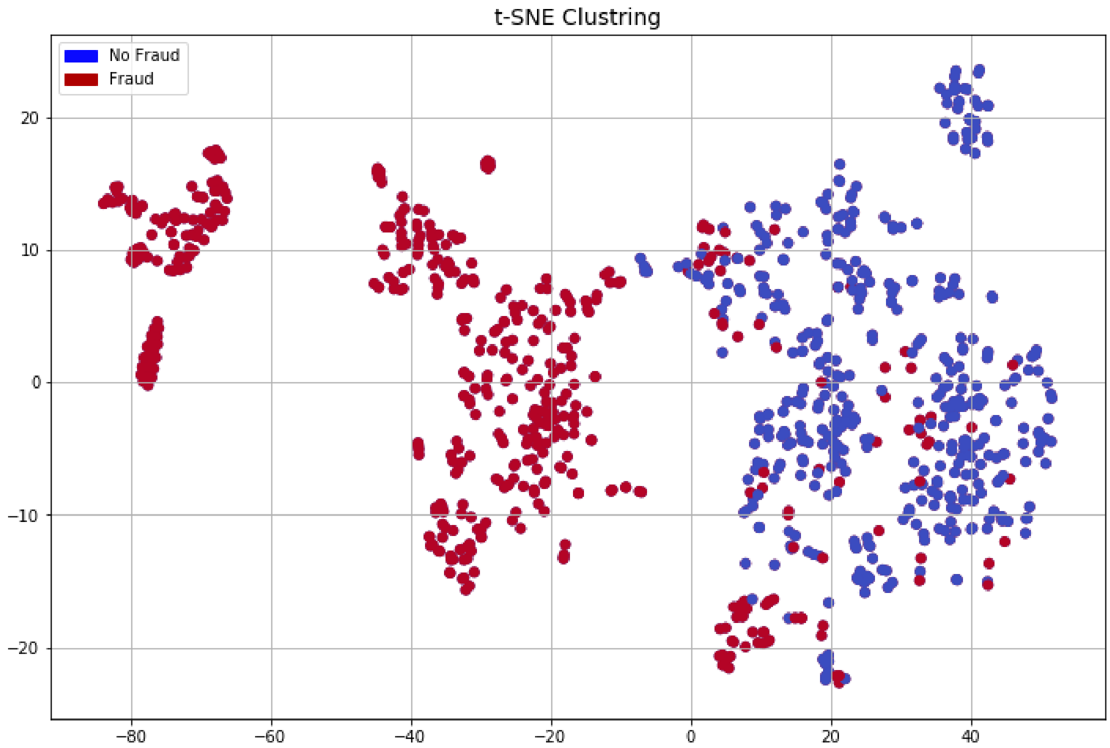
In [15]:
```
%time plot_tsne(3)
```

```
Wall time: 16.2 s
```



In [16]: %time plot_tsne(2)

```
Wall time: 5.23 s
```



3 SMOTE (Synthetic Minority Over-sampling Technique)

We will use SMOTE to address to imbalanced datasets

- SMOTE will generate synthetic points from the minority class in order to make our training set more balance
- SMOTE will picks the distance between the closest neighbors of the minority class, in between these distances it creates synthetic points
- More information is obtained

In [17]: !pip install imbalanced-learn

```
Collecting imbalanced-learn
  Downloading https://files.pythonhosted.org/packages/e5/4c/7557e1c2e791bd43878f8c82065bddc5798
Requirement already satisfied: scipy>=0.13.3 in c:\users\michael\anaconda3\lib\site-packages (f
Requirement already satisfied: scikit-learn>=0.20 in c:\users\michael\anaconda3\lib\site-packag
Requirement already satisfied: numpy>=1.8.2 in c:\users\michael\anaconda3\lib\site-packages (fr
Installing collected packages: imbalanced-learn
Successfully installed imbalanced-learn-0.4.3
```

4 Train and Evaluate

In [18]:
```
def plot_cm(cm):
    cm = pd.DataFrame(cm)
    sns.heatmap(cm, annot=True, fmt='.1f', cmap='YlGnBu', linewidths=5, vmax=500)
    plt.yticks(rotation=0)
    plt.show()
```

In [19]:
```
from sklearn.model_selection import StratifiedShuffleSplit
from imblearn.over_sampling import SMOTE
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score, ac
from sklearn.model_selection import GridSearchCV
import time

# split original datasets into features and label data
X = df.drop(['Class'], axis=1).values
y = df.Class.values


def cross_val(clf, grid_param=None, k=5, test=0.5, cv_test=0.2, plot=True):

    sss = StratifiedShuffleSplit(n_splits=1, test_size=test, random_state=2019)
    for trn_idx, tes_idx in sss.split(X,y):
        time1 = time.time()
        x_trn, x_tes = X[trn_idx], X[tes_idx]
        y_trn, y_tes = y[trn_idx], y[tes_idx]
        x_res, y_res = SMOTE(sampling_strategy='minority').fit_resample(x_trn, y_trn)
        auc = []

        k = 1 if grid_param else k # if use GridSearchCV, use one fold becuase GridSe

        for fold, (tr_idx, te_idx) in enumerate(StratifiedShuffleSplit(n_splits=k, tes
            Xtrain, Xtest = x_res[tr_idx], x_res[te_idx]
            ytrain, ytest = y_res[tr_idx], y_res[te_idx]
            if grid_param:
                cv = GridSearchCV(clf, grid_param, scoring='roc_auc', verbose=1, n_jol
                cv.fit(Xtrain, ytrain)
                score = roc_auc_score(ytest, cv.predict(Xtest))
                print(fold, ':', score)
                auc.append(score)
            else:
                clf.fit(Xtrain, ytrain)
                score = roc_auc_score(ytest, clf.predict(Xtest))
                print(fold, ':', score)
                auc.append(score)
        print(k, 'cv score: ', sum(auc)/len(auc))

        if grid_param:
```

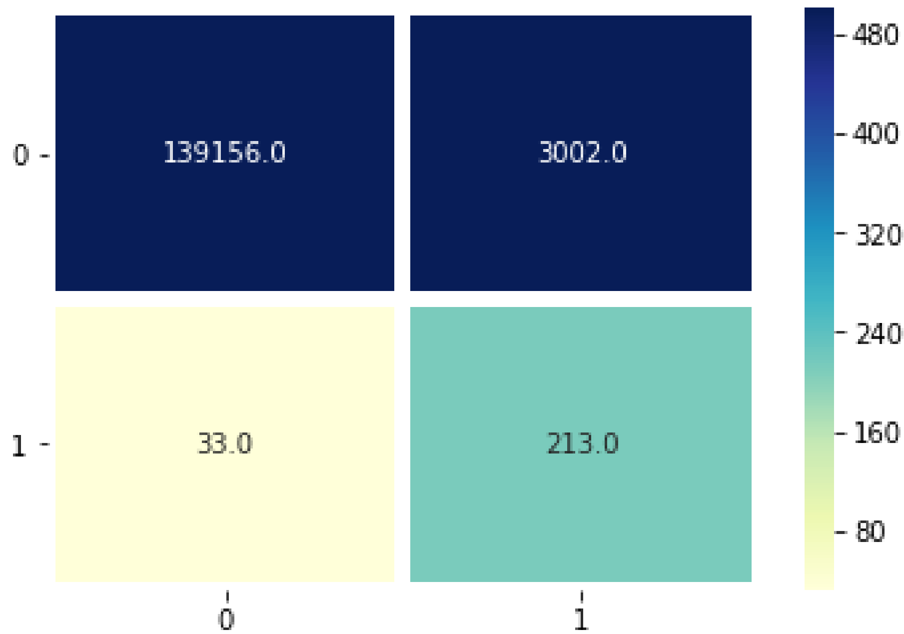
```
            preds = cv.predict(x_tes)
            print('test score: ', roc_auc_score(y_tes, preds))
            print('best parameters: ', cv.best_params_)
        else:
            preds = clf.predict(x_tes)
            print('test score: ', roc_auc_score(y_tes, preds))

        print('Completion time:', time.time()-time1, 'seconds')

        if plot:
            cm = confusion_matrix(y_tes, preds)
            plot_cm(cm)
            print(classification_report(y_tes, preds))

    return cv if grid_param else clf
```

5 Model Selection

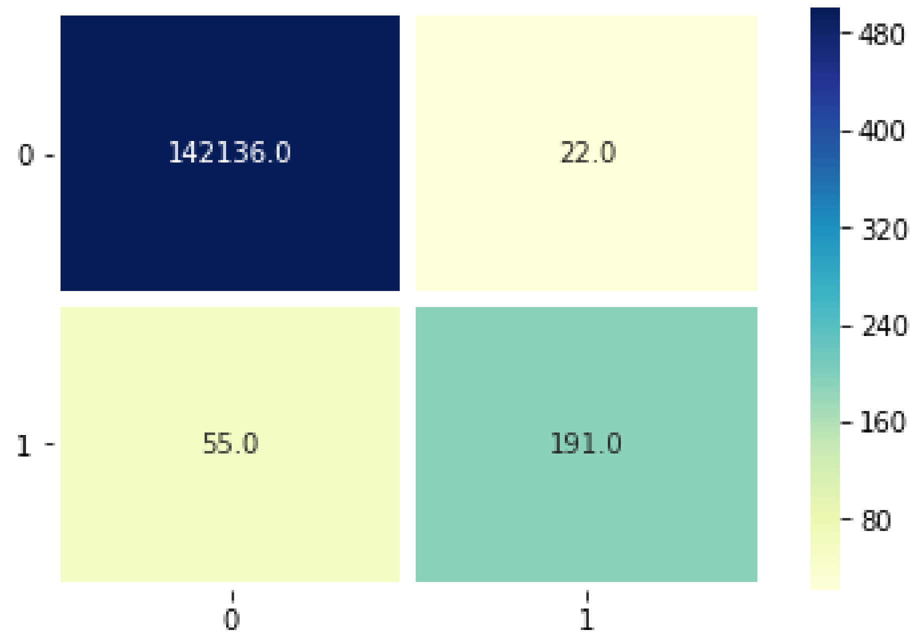
We will define a number of classifiers and train them on just one fold training dataset, and we will select just a small subset of models for tuning based on the ROC_AUC_SCORE and time efficiency.

```
In [20]: from sklearn.linear_model import LogisticRegression
         from sklearn.neighbors import KNeighborsClassifier
         from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
         from sklearn.svm import SVC

         classifiers = {
             "LogisticRegression": LogisticRegression(),
             "RandomForestClassifier": RandomForestClassifier(),
             "kNieghborsClassifier": KNeighborsClassifier(),
             "SVC": SVC(),
             "GradientBoostingClassifier": GradientBoostingClassifier()
         }

         clfs = {}
         for clf in classifiers:
             print(clf)
             clfs.setdefault(clf, cross_val(classifiers[clf], k=3))
```

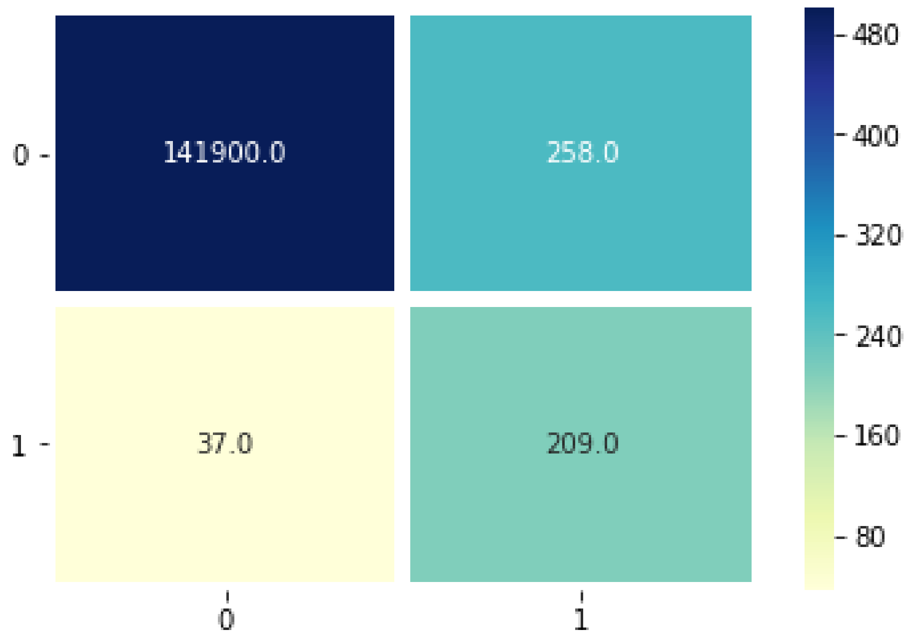
```
LogisticRegression
0 : 0.9573004919877597
1 : 0.9576002290353881
2 : 0.9565618770830947
3 cv score:  0.9571541993687475
test score:  0.9223681551169962
Completion time: 13.247580289840698 seconds
```



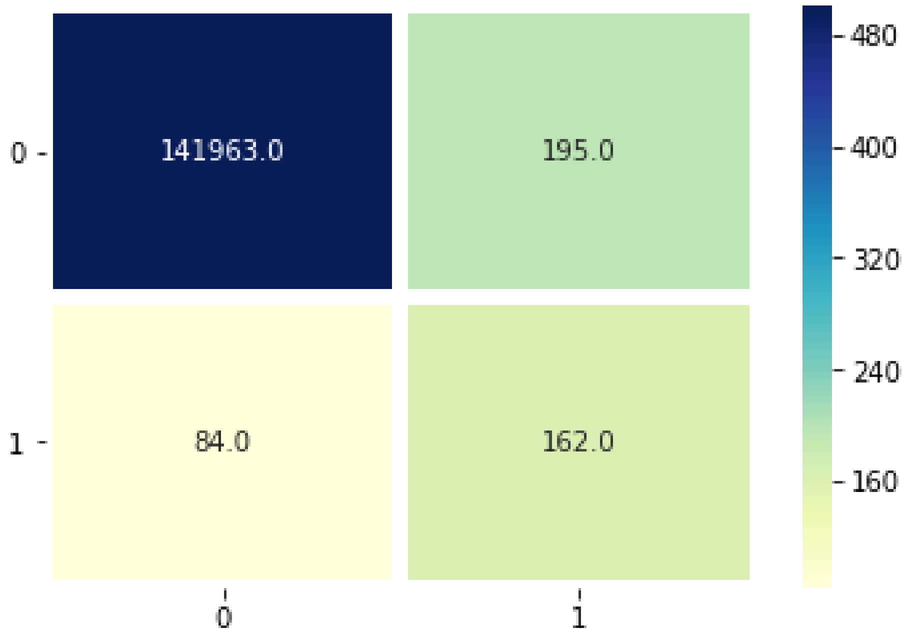

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99    142158
           1       0.07      0.87      0.12       246

   micro avg       0.98      0.98      0.98    142404
   macro avg       0.53      0.92      0.56    142404
weighted avg       1.00      0.98      0.99    142404

RandomForestClassifier
0 : 0.9998768992684299
1 : 0.9999120678133024
2 : 0.9999472425436129
3 cv score:  0.999912069875115
test score:  0.8881340034225058
Completion time: 31.9435932636261 seconds
```



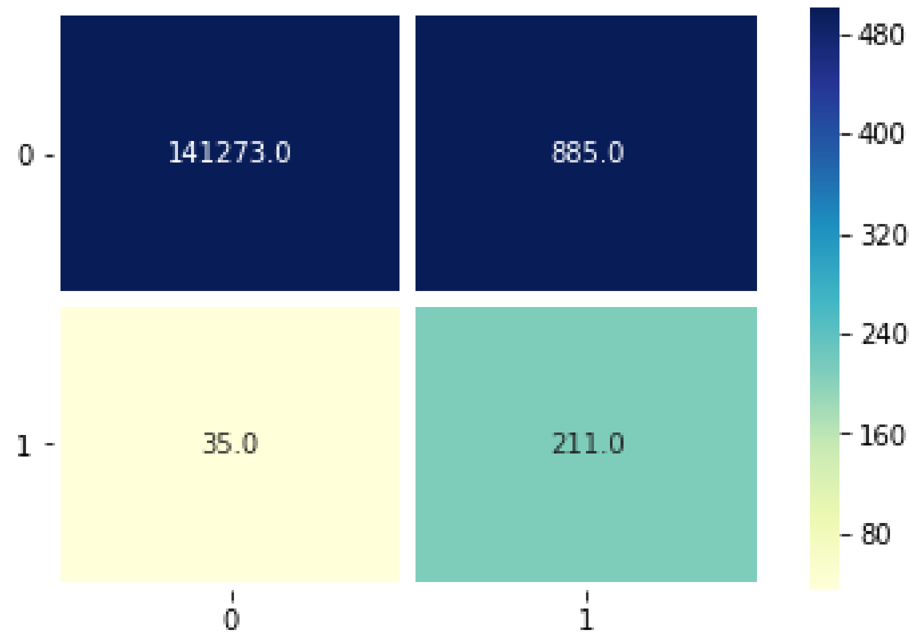

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    142158
           1       0.90      0.78      0.83       246

   micro avg       1.00      1.00      1.00    142404
   macro avg       0.95      0.89      0.92    142404
weighted avg       1.00      1.00      1.00    142404

kNieghborsClassifier
0 : 0.9991207090602138
1 : 0.9990327459463262
2 : 0.9990855374226224
3 cv score:  0.999079664143054
test score:  0.9238893069511457
Completion time: 520.0365943908691 seconds
```




```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    142158
           1       0.45      0.85      0.59       246

   micro avg       1.00      1.00      1.00    142404
   macro avg       0.72      0.92      0.79    142404
weighted avg       1.00      1.00      1.00    142404

SVC
0 : 0.9994196679797411
1 : 0.9992437831944005
2 : 0.9992789814293754
3 cv score:  0.9993141442011724
test score:  0.8285824361008142
Completion time: 768.6259486675262 seconds
```




```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    142158
           1       0.45      0.66      0.54       246

   micro avg       1.00      1.00      1.00    142404
   macro avg       0.73      0.83      0.77    142404
weighted avg       1.00      1.00      1.00    142404

GradientBoostingClassifier
0 : 0.9926841466006605
1 : 0.9926490188796788
2 : 0.992561034735302
3 cv score:  0.9926314000718804
test score:  0.9257490548990662
Completion time: 222.79033279418945 seconds
```




```
              precision    recall  f1-score   support

           0       1.00      0.99      1.00    142158
           1       0.19      0.86      0.31       246

   micro avg       0.99      0.99      0.99    142404
   macro avg       0.60      0.93      0.66    142404
weighted avg       1.00      0.99      1.00    142404
```

We should look at our trained models, and determine how to set values for parameters tuning

```
In [21]: import pickle
         for clf in clfs:
             pickle.dump(clfs[clf], open('models/'+clf+'_intial.sav', 'wb'))
```

```
In [29]: !dir models
```

```
 Volume in drive C has no label.
 Volume Serial Number is 244D-FC98

 Directory of C:\Users\michael\Desktop\coursera\Advanced Data Science with IBM Specialization\I

05/02/2019  05:17 PM    <DIR>          .
05/02/2019  05:17 PM    <DIR>          ..
```

```
05/02/2019  04:16 PM           135,848 GradientBoostingClassifier_intial.sav
05/02/2019  05:17 PM           243,511 GradientBoostingClassifier_tuned.sav
05/02/2019  04:16 PM        61,512,819 kNieghborsClassifier_intial.sav
05/02/2019  04:16 PM             1,042 LogisticRegression_intial.sav
05/02/2019  05:17 PM             5,617 LogisticRegression_tuned.sav
05/02/2019  04:16 PM           721,425 RandomForestClassifier_intial.sav
05/02/2019  05:17 PM         1,496,013 RandomForestClassifier_tuned.sav
05/02/2019  04:16 PM         1,164,080 SVC_intial.sav
               8 File(s)     65,280,355 bytes
               2 Dir(s)  865,668,763,648 bytes free
```

From the above results, we can see that GradientBoostingClassifer, RandomForestClassifier, and LogisticsClassifier did yield a relatively good results with a very fast completeion time, therefore, we can further fine tune these 3 models, and compare how they perform.

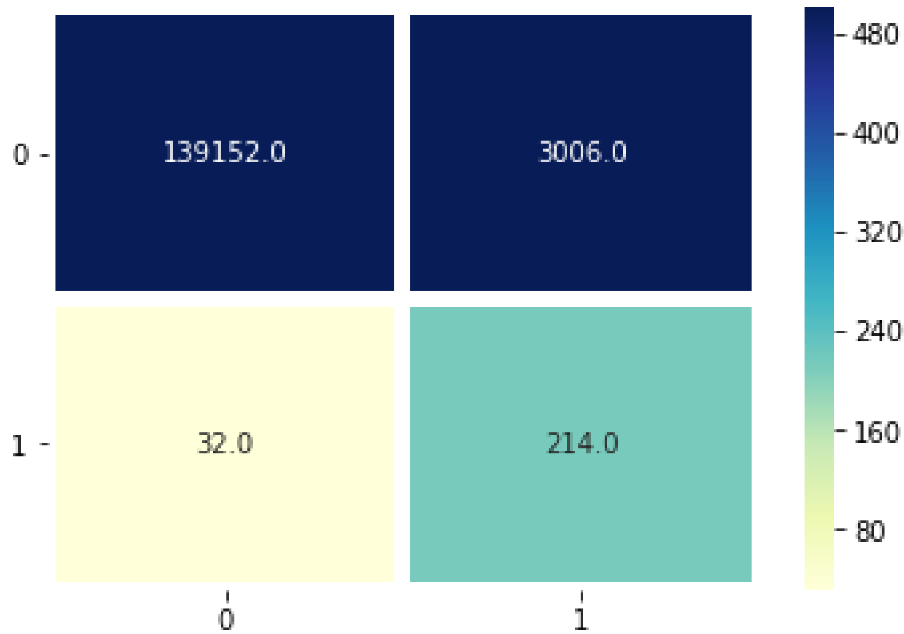
6 Model Tuning

```
In [23]: # define parameters values

         log_params = {"penalty": ['l1', 'l2'], 'C': [0.001, 0.01, 0.1, 1, 10, 100], "n_jobs":
         rf_params = {"min_samples_leaf": [1,3,5,10,25], "max_features": [1, 0.5, 'log2', 'sqr
         gb_params = {"min_samples_leaf": [100, 500], "max_depth": [5, 8], "subsample": [0.3, 

         params = {
             "LogisticRegression": log_params,
             "RandomForestClassifier": rf_params,
             "GradientBoostingClassifier": gb_params
         }

         classifiers = {
             "LogisticRegression": LogisticRegression(),
             "RandomForestClassifier": RandomForestClassifier(),
             "GradientBoostingClassifier": GradientBoostingClassifier()
         }
```

```
In [25]: clfs_tuned = {}
         for clf in classifiers:
             print('Tuning', clf)
             clfs_tuned.setdefault(clf, cross_val(classifiers[clf], grid_param=params[clf]))
```

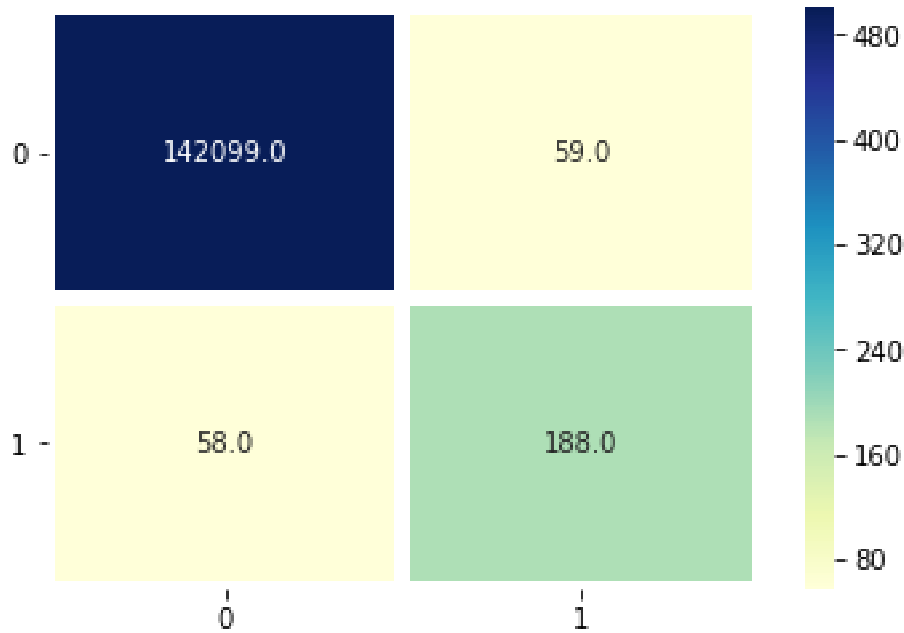
```
Tuning LogisticRegression
Fitting 2 folds for each of 12 candidates, totalling 24 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  24 out of  24 | elapsed:   19.2s finished
```

```
0 : 0.9559990956246349
1 cv score:  0.9559990956246349
test score:  0.924386606589233
best parameters:  {'C': 100, 'n_jobs': -1, 'penalty': 'l2'}
Completion time: 25.183666944503784 seconds
```



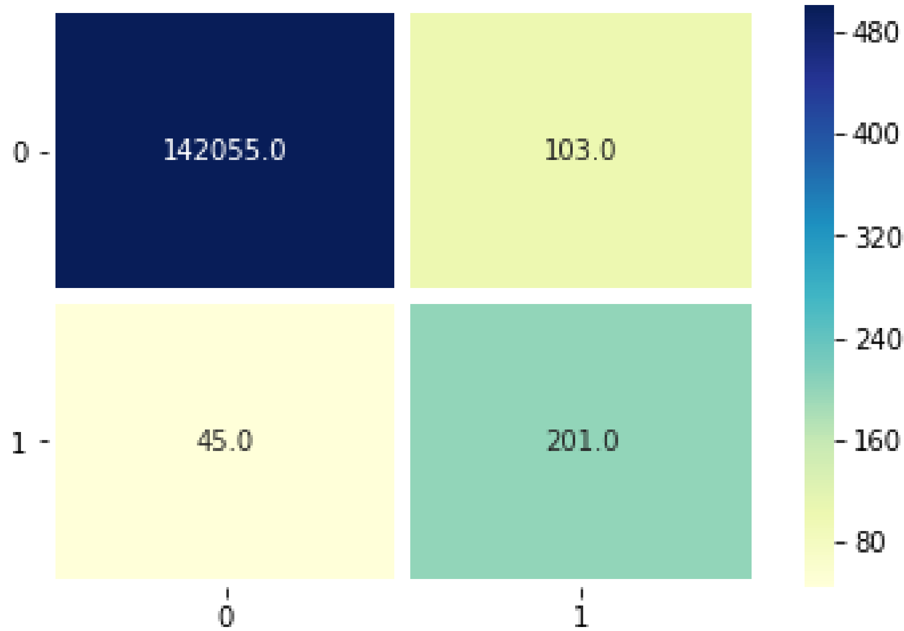

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99    142158
           1       0.07      0.87      0.12       246

   micro avg       0.98      0.98      0.98    142404
   macro avg       0.53      0.92      0.56    142404
weighted avg       1.00      0.98      0.99    142404

Tuning RandomForestClassifier
Fitting 2 folds for each of 20 candidates, totalling 40 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:   51.6s finished


0 : 0.9997537972997723
1 cv score:  0.9997537972997723
```

```
test score:  0.8819063055569567
best parameters:  {'max_features': 1, 'min_samples_leaf': 5, 'n_jobs': -1}
Completion time: 53.95374536514282 seconds
```




```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    142158
           1       0.76      0.76      0.76       246

   micro avg       1.00      1.00      1.00    142404
   macro avg       0.88      0.88      0.88    142404
weighted avg       1.00      1.00      1.00    142404

Tuning GradientBoostingClassifier
Fitting 2 folds for each of 48 candidates, totalling 96 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed: 12.7min
[Parallel(n_jobs=-1)]: Done  96 out of  96 | elapsed: 36.5min finished


0 : 0.9997362127180642
1 cv score:  0.9997362127180642
test score:  0.9081743124019683
```

```
best parameters:  {'learning_rate': 0.5, 'max_depth': 5, 'min_samples_leaf': 500, 'n_estimator
Completion time: 2246.9703459739685 seconds
```




```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    142158
           1       0.66      0.82      0.73       246

   micro avg       1.00      1.00      1.00    142404
   macro avg       0.83      0.91      0.87    142404
weighted avg       1.00      1.00      1.00    142404
```

As a result of tuning our models' parameters, we can see that all of our models have a small improvement in terms of ROC_AUC score, and the LogisticClassifier has the best performance, therefore, we can decise to use LogisticClassifier as our final model for deployment.

```
In [27]: # saving models
         for clf in clfs_tuned:
             pickle.dump(clfs_tuned[clf], open('models/'+clf+'_tuned.sav', 'wb'))
```

```
In [28]: !dir models
```

```
 Volume in drive C has no label.
 Volume Serial Number is 244D-FC98
```

```
 Directory of C:\Users\michael\Desktop\coursera\Advanced Data Science with IBM Specialization\

05/02/2019  05:17 PM    <DIR>          .
05/02/2019  05:17 PM    <DIR>          ..
05/02/2019  04:16 PM           135,848 GradientBoostingClassifier_intial.sav
05/02/2019  05:17 PM           243,511 GradientBoostingClassifier_tuned.sav
05/02/2019  04:16 PM        61,512,819 kNieghborsClassifier_intial.sav
05/02/2019  04:16 PM             1,042 LogisticRegression_intial.sav
05/02/2019  05:17 PM             5,617 LogisticRegression_tuned.sav
05/02/2019  04:16 PM           721,425 RandomForestClassifier_intial.sav
05/02/2019  05:17 PM         1,496,013 RandomForestClassifier_tuned.sav
05/02/2019  04:16 PM         1,164,080 SVC_intial.sav
               8 File(s)     65,280,355 bytes
               2 Dir(s)  865,669,206,016 bytes free
```