Brito-creditcardfraud

May 14, 2019

1 Implement a simple NN to predict largely imbalanced credit card fraud dataset

In [1]:
```
# largely inspired and guided from
# https://www.kaggle.com/janiobachmann/credit-fraud-dealing-with-imbalanced-datasets

# Imported Libraries

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA, TruncatedSVD
import matplotlib.patches as mpatches
import time

# Classifier Libraries
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import collections
```

In [3]:
```
# Imported Libraries
from imblearn.datasets import fetch_datasets
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from imblearn.pipeline import make_pipeline as imbalanced_make_pipeline
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import NearMiss
from imblearn.metrics import classification_report_imbalanced
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score, ac
from collections import Counter
from sklearn.model_selection import KFold, StratifiedKFold
```

```python
import warnings
warnings.filterwarnings("ignore")
```

```python
df = pd.read_csv('creditcard.csv')
df.head()
```

```
Out[3]:    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       V25  \
        0  0.098698  0.363787  ... -0.018307  0.277838 -0.110474  0.066928  0.128539
        1  0.085102 -0.255425  ... -0.225775 -0.638672  0.101288 -0.339846  0.167170
        2  0.247676 -1.514654  ...  0.247998  0.771679  0.909412 -0.689281 -0.327642
        3  0.377436 -1.387024  ... -0.108300  0.005274 -0.190321 -1.175575  0.647376
        4 -0.270533  0.817739  ... -0.009431  0.798278 -0.137458  0.141267 -0.206010

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

        [5 rows x 31 columns]
```

```python
In [5]: # Very unbalanced
        print('No Frauds', round(df['Class'].value_counts()[0]/len(df) * 100,2), '% of the data
        print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,2), '% of the dataset
```

```
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
```

For this very unbalanced data, need to create a subsample with a 50/50 ratio of fraud and non-fraud transactions. Meaning our sub-sample will have the same amount of fraud and non fraud transactions.

Addtionally, there is the need to scale the other two features: time and amount. Using sklearn for that.

```python
In [6]: # Since most of our data has already been scaled we should scale the columns that are
        from sklearn.preprocessing import StandardScaler, RobustScaler

        # RobustScaler is less prone to outliers.
```

```
std_scaler = StandardScaler()
rob_scaler = RobustScaler()

df['scaled_amount'] = rob_scaler.fit_transform(df['Amount'].values.reshape(-1,1))
df['scaled_time'] = rob_scaler.fit_transform(df['Time'].values.reshape(-1,1))

df.drop(['Time','Amount'], axis=1, inplace=True)

scaled_amount = df['scaled_amount']
scaled_time = df['scaled_time']

df.drop(['scaled_amount', 'scaled_time'], axis=1, inplace=True)
df.insert(0, 'scaled_amount', scaled_amount)
df.insert(1, 'scaled_time', scaled_time)

# Amount and Time are Scaled!

df.head()
```

Out[6]:

```
   scaled_amount  scaled_time        V1        V2        V3        V4  \
0       1.783274    -0.994983 -1.359807 -0.072781  2.536347  1.378155
1      -0.269825    -0.994983  1.191857  0.266151  0.166480  0.448154
2       4.983721    -0.994972 -1.358354 -1.340163  1.773209  0.379780
3       1.418291    -0.994972 -0.966272 -0.185226  1.792993 -0.863291
4       0.670579    -0.994960 -1.158233  0.877737  1.548718  0.403034

         V5        V6        V7        V8  ...       V20       V21       V22  \
0 -0.338321  0.462388  0.239599  0.098698  ...  0.251412 -0.018307  0.277838
1  0.060018 -0.082361 -0.078803  0.085102  ... -0.069083 -0.225775 -0.638672
2 -0.503198  1.800499  0.791461  0.247676  ...  0.524980  0.247998  0.771679
3 -0.010309  1.247203  0.237609  0.377436  ... -0.208038 -0.108300  0.005274
4 -0.407193  0.095921  0.592941 -0.270533  ...  0.408542 -0.009431  0.798278

        V23       V24       V25       V26       V27       V28  Class
0 -0.110474  0.066928  0.128539 -0.189115  0.133558 -0.021053      0
1  0.101288 -0.339846  0.167170  0.125895 -0.008983  0.014724      0
2  0.909412 -0.689281 -0.327642 -0.139097 -0.055353 -0.059752      0
3 -0.190321 -1.175575  0.647376 -0.221929  0.062723  0.061458      0
4 -0.137458  0.141267 -0.206010  0.502292  0.219422  0.215153      0

[5 rows x 31 columns]
```

In [7]:

```
# Saving the dataframe with scaled values, just in case
df.to_csv('creditcard-scaled.csv')
```

In [8]:

```
# creating the subsample
# Since our classes are highly skewed we should make them equivalent in order to have
```

```
# Lets shuffle the data before creating the subsamples

df = df.sample(frac=1)

# amount of fraud classes 492 rows.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] == 0][:492]

normal_distributed_df = pd.concat([fraud_df, non_fraud_df])

# Shuffle dataframe rows
subsample_df = normal_distributed_df.sample(frac=1, random_state=42)

subsample_df.head()
```

Out[8]:

```
        scaled_amount  scaled_time        V1        V2        V3        V4  \
220454       0.937609     0.675349 -0.354864 -0.079652  1.268568 -0.771426
151103       0.164326     0.116695  1.707857  0.024881 -0.488140  3.787548
58625        1.174317    -0.425487 -0.969392 -0.425451  1.719085 -1.522503
106679       2.868721    -0.171771 -0.440095  1.137239 -3.227080  3.242293
105178      -0.293440    -0.179725  1.140431  1.134243 -1.429455  2.012226

              V5        V6        V7        V8  ...       V20       V21  \
220454  0.014903  0.006893  0.529193  0.025738  ...  0.071095  0.318964
151103  1.139451  2.914673 -0.743358  0.699136  ... -0.368014  0.010865
58625   0.039450 -1.044677  0.846700 -0.195767  ...  0.180883 -0.028532
106679 -2.033998 -1.618415 -3.028013  0.764555  ...  0.895841  0.764187
105178  0.622800 -1.152923  0.221159  0.037372  ... -0.099712 -0.367136

             V22       V23       V24       V25       V26       V27       V28  \
220454  0.876832  0.031803 -0.404039 -0.791774  0.428448 -0.048719 -0.037618
151103  0.548258  0.091218 -1.007959 -0.082183  0.179709  0.007738 -0.068841
58625  -0.014184  0.228058  0.584870 -0.081384 -1.026947 -0.041269 -0.068795
106679 -0.275578 -0.343572  0.233085  0.606434 -0.315433  0.768291  0.459623
105178 -0.891627 -0.160578 -0.108326  0.668374 -0.352393  0.071993  0.113684

        Class
220454      0
151103      1
58625       0
106679      1
105178      1

[5 rows x 31 columns]
```

In [10]:

```
print('Distribution of the Classes in the subsample dataset')
print(subsample_df['Class'].value_counts()/len(subsample_df))
```

```
colors = ["#0101DF", "#DF0101"]

sns.countplot('Class', data=subsample_df, palette=colors)
plt.title('Equally Distributed Classes', fontsize=14)
plt.show()
```

```
Distribution of the Classes in the subsample dataset
1    0.5
0    0.5
Name: Class, dtype: float64
```

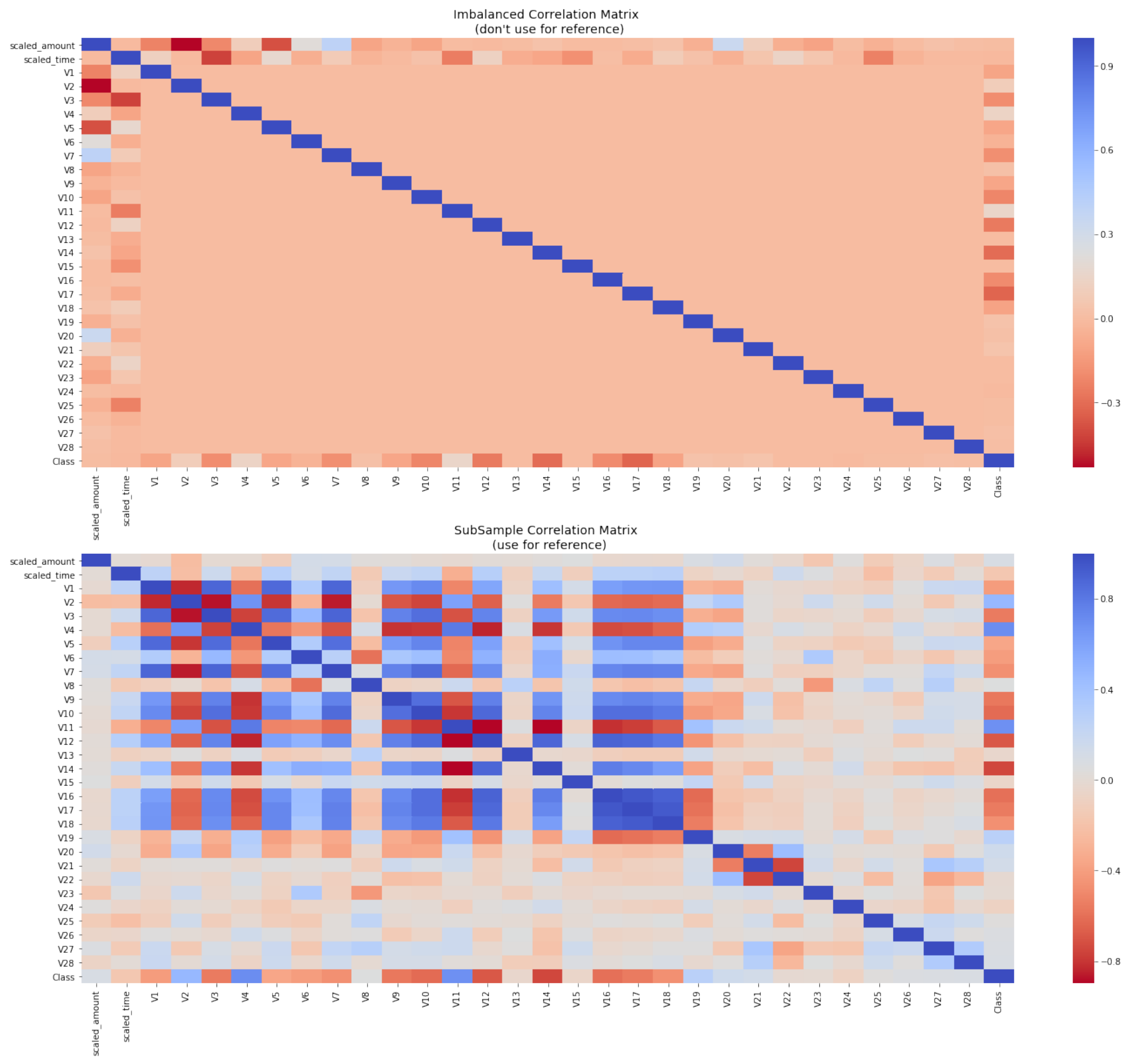

```
In [11]: # Saving the subset dataframe for google AutoML
         subsample_df.to_csv('creditcard-subset-distributed.csv')
```

1.1 Quickly checking the most influential features (aka correlation matrices)

V17, V14, V12 and V10 are negatively correlated. V2, V4, V11, and V19 are positively correlated.

```
In [62]: # Make sure we use the subsample in our correlation

         f, (ax1, ax2) = plt.subplots(2, 1, figsize=(24,20))

         # Entire DataFrame
```

```python
corr = df.corr()
sns.heatmap(corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax1)
ax1.set_title("Imbalanced Correlation Matrix \n (don't use for reference)", fontsize=


sub_sample_corr = subsample_df.corr()
sns.heatmap(sub_sample_corr, cmap='coolwarm_r', annot_kws={'size':20}, ax=ax2)
ax2.set_title('SubSample Correlation Matrix \n (use for reference)', fontsize=14)
plt.show()
```



Let's check if a simple neural network behaves in both the random undersample (subsample_df) and oversample dataframes and see whether they can predict accurately both non-fraud and fraud cases. Let's leverage a confusion matrix.

Confusion matrix:

- Upper Left Square: The amount of correctly classified by our model of no fraud transactions.

- Upper Right Square: The amount of incorrectly classified transactions as fraud cases, but the actual label is no fraud .
- Lower Left Square: The amount of incorrectly classified transactions as no fraud cases, but the actual label is fraud .
- Lower Right Square: The amount of correctly classified by our model of fraud transactions.

In [13]:
```
# subset_df is from the random undersample data (fewer instances) aka subset
X = subsample_df.drop('Class', axis=1)
y = subsample_df['Class']

# Our data is already scaled we should split our training and test sets
from sklearn.model_selection import train_test_split

# This is explicitly used for undersampling.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=
```

In [14]:
```
import keras
from keras import backend as K
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense
from keras.optimizers import Adam
from keras.metrics import categorical_crossentropy

n_inputs = X_train.shape[1]

undersample_model = Sequential([
    Dense(n_inputs, input_shape=(n_inputs, ), activation='relu'),
    Dense(32, activation='relu'),
    Dense(2, activation='softmax')
])
```

```
Using TensorFlow backend.
```

In [15]:
```
undersample_model.summary()
```

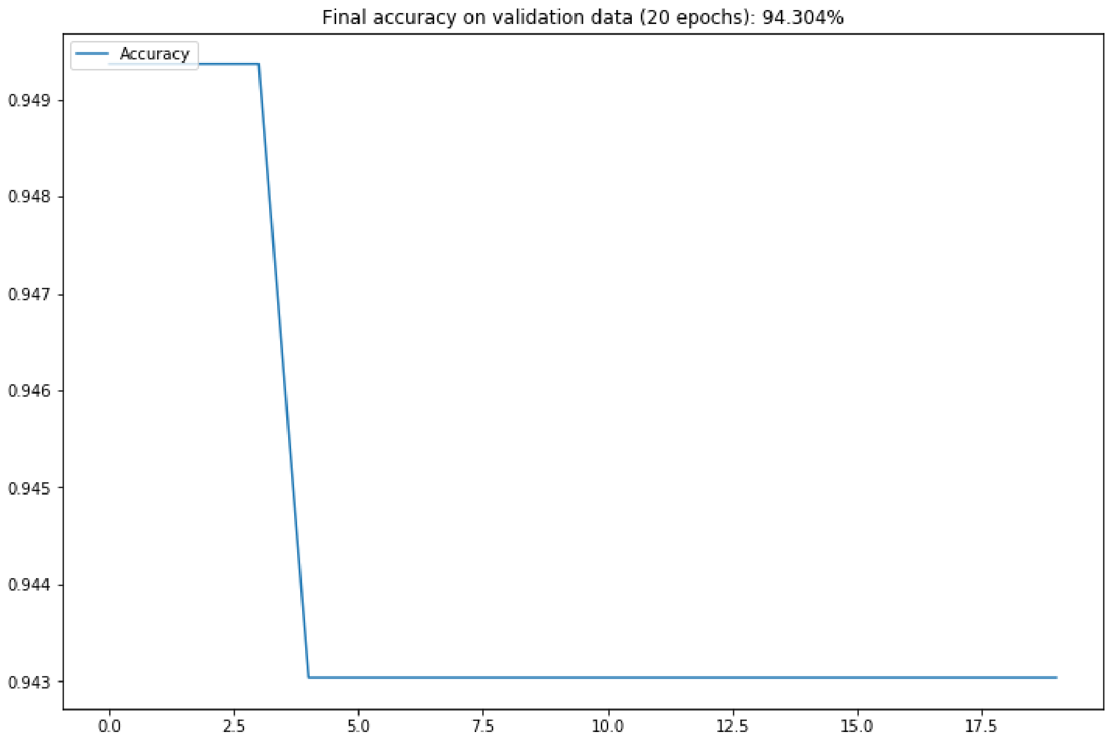
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 30)                930
_________________________________________________________________
dense_2 (Dense)              (None, 32)                992
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 66
=================================================================
Total params: 1,988
Trainable params: 1,988
Non-trainable params: 0
```

```
---------------------------------------------------------------
```

```
In [16]: undersample_model.compile(Adam(lr=0.001), loss='sparse_categorical_crossentropy', met

In [54]: hist = undersample_model.fit(X_train, y_train, validation_split=0.2, batch_size=25, ep
```

```
Train on 629 samples, validate on 158 samples
Epoch 1/20
 - 0s - loss: 0.0232 - acc: 0.9936 - val_loss: 0.3088 - val_acc: 0.9430
Epoch 2/20
 - 0s - loss: 0.0219 - acc: 0.9936 - val_loss: 0.3172 - val_acc: 0.9430
Epoch 3/20
 - 0s - loss: 0.0206 - acc: 0.9952 - val_loss: 0.3195 - val_acc: 0.9430
Epoch 4/20
 - 0s - loss: 0.0195 - acc: 0.9952 - val_loss: 0.3314 - val_acc: 0.9430
Epoch 5/20
 - 0s - loss: 0.0186 - acc: 0.9952 - val_loss: 0.3384 - val_acc: 0.9430
Epoch 6/20
 - 0s - loss: 0.0177 - acc: 0.9952 - val_loss: 0.3441 - val_acc: 0.9430
Epoch 7/20
 - 0s - loss: 0.0167 - acc: 0.9968 - val_loss: 0.3517 - val_acc: 0.9430
Epoch 8/20
 - 0s - loss: 0.0172 - acc: 0.9952 - val_loss: 0.3509 - val_acc: 0.9430
Epoch 9/20
 - 0s - loss: 0.0159 - acc: 0.9984 - val_loss: 0.3564 - val_acc: 0.9430
Epoch 10/20
 - 0s - loss: 0.0143 - acc: 0.9984 - val_loss: 0.3628 - val_acc: 0.9430
Epoch 11/20
 - 0s - loss: 0.0155 - acc: 0.9968 - val_loss: 0.3537 - val_acc: 0.9430
Epoch 12/20
 - 0s - loss: 0.0136 - acc: 0.9984 - val_loss: 0.3647 - val_acc: 0.9430
Epoch 13/20
 - 0s - loss: 0.0127 - acc: 1.0000 - val_loss: 0.3719 - val_acc: 0.9430
Epoch 14/20
 - 0s - loss: 0.0123 - acc: 0.9984 - val_loss: 0.3854 - val_acc: 0.9367
Epoch 15/20
 - 0s - loss: 0.0114 - acc: 0.9984 - val_loss: 0.3925 - val_acc: 0.9367
Epoch 16/20
 - 0s - loss: 0.0108 - acc: 1.0000 - val_loss: 0.3945 - val_acc: 0.9430
Epoch 17/20
 - 0s - loss: 0.0102 - acc: 1.0000 - val_loss: 0.3996 - val_acc: 0.9430
Epoch 18/20
 - 0s - loss: 0.0100 - acc: 1.0000 - val_loss: 0.4036 - val_acc: 0.9430
Epoch 19/20
 - 0s - loss: 0.0095 - acc: 1.0000 - val_loss: 0.4099 - val_acc: 0.9430
Epoch 20/20
 - 0s - loss: 0.0092 - acc: 1.0000 - val_loss: 0.4159 - val_acc: 0.9430
```

In [37]:
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=(12,8))
plt.plot(hist.history['val_acc'], label='Accuracy')
plt.title("Final accuracy on validation data (20 epochs): {:.3%}".format(hist.history
plt.legend(loc=2)
_ = plt.show()
```



1.2 Setting up parameters for the prediction - split the oversample using StratifiedK-Fold

In [33]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedShuffleSplit

print('No Frauds', round(df['Class'].value_counts()[0]/len(df) * 100,2), '% of the da
print('Frauds', round(df['Class'].value_counts()[1]/len(df) * 100,2), '% of the datas

X = df.drop('Class', axis=1)
y = df['Class']

sss = StratifiedKFold(n_splits=5, random_state=None, shuffle=False)
```

```
for train_index, test_index in sss.split(X, y):
    print("Train:", train_index, "Test:", test_index)
    original_Xtrain, original_Xtest = X.iloc[train_index], X.iloc[test_index]
    original_ytrain, original_ytest = y.iloc[train_index], y.iloc[test_index]

# We already have X_train and y_train for undersample data thats why I am using origi
# original_Xtrain, original_Xtest, original_ytrain, original_ytest = train_test_split

# Check the Distribution of the labels


# Turn into an array
original_Xtrain = original_Xtrain.values
original_Xtest = original_Xtest.values
original_ytrain = original_ytrain.values
original_ytest = original_ytest.values

# See if both the train and test label distribution are similarly distributed
train_unique_label, train_counts_label = np.unique(original_ytrain, return_counts=True
test_unique_label, test_counts_label = np.unique(original_ytest, return_counts=True)
print('-' * 100)

print('Label Distributions: \n')
print(train_counts_label/ len(original_ytrain))
print(test_counts_label/ len(original_ytest))
```

```
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
Train: [ 54677  55187  56165 ... 284804 284805 284806] Test: [     0      1      2 ... 56962 5696
Train: [     0      1      2 ... 284804 284805 284806] Test: [ 54677  55187  56165 ... 113921 
Train: [     0      1      2 ... 284804 284805 284806] Test: [113922 113923 113924 ... 170893 
Train: [     0      1      2 ... 284804 284805 284806] Test: [163864 164807 164989 ... 227853 
Train: [     0      1      2 ... 227853 227854 227855] Test: [223435 224368 225054 ... 284804 
----------------------------------------------------------------------------------------------------
Label Distributions:

[0.99827076 0.00172924]
[0.99827952 0.00172048]
```

1.3 Getting the predictions with undersample

In [38]:
```
undersample_predictions = undersample_model.predict(original_Xtest, batch_size=400, v
```

In [39]:
```
undersample_fraud_predictions = undersample_model.predict_classes(original_Xtest, bat
```

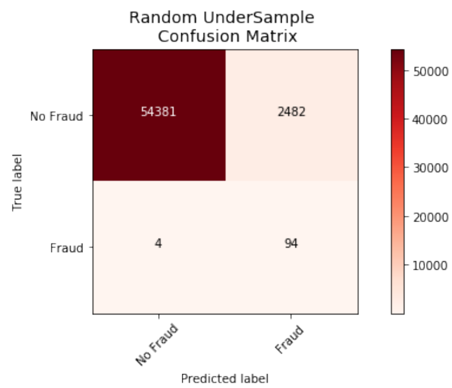
1.4 Creating the confusion matrix

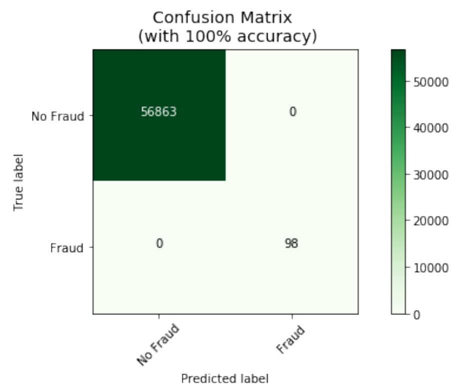
```
In [41]: import itertools

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

             print(cm)

             plt.imshow(cm, interpolation='nearest', cmap=cmap)
             plt.title(title, fontsize=14)
             plt.colorbar()
             tick_marks = np.arange(len(classes))
             plt.xticks(tick_marks, classes, rotation=45)
             plt.yticks(tick_marks, classes)

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

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

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

         undersample_cm = confusion_matrix(original_ytest, undersample_fraud_predictions)
         actual_cm = confusion_matrix(original_ytest, original_ytest)
         labels = ['No Fraud', 'Fraud']

         fig = plt.figure(figsize=(16,8))

         fig.add_subplot(221)
         plot_confusion_matrix(undersample_cm, labels, title="Random UnderSample \n Confusion I
```

```
        fig.add_subplot(222)
        plot_confusion_matrix(actual_cm, labels, title="Confusion Matrix \n (with 100% accura
```

```
Confusion matrix, without normalization
[[54381  2482]
 [    4    94]]
Confusion matrix, without normalization
[[56863     0]
 [    0    98]]
```





1.5 SMOTE Calculation (not used for comparison with Google AutoML)

```
In [63]: from imblearn.over_sampling import SMOTE

         # SMOTE Technique (OverSampling) After splitting and Cross Validating
         sm = SMOTE(ratio='minority', random_state=42)
         # Xsm_train, ysm_train = sm.fit_sample(X_train, y_train)


         # This will be the data were we are going to
         Xsm_train, ysm_train = sm.fit_sample(original_Xtrain, original_ytrain)
```

```
In [64]: n_inputs = Xsm_train.shape[1]

         oversample_model = Sequential([
             Dense(n_inputs, input_shape=(n_inputs, ), activation='relu'),
             Dense(32, activation='relu'),
             Dense(2, activation='softmax')
         ])

         oversample_model.compile(Adam(lr=0.001), loss='sparse_categorical_crossentropy', metr

         oversample_model.fit(Xsm_train, ysm_train, validation_split=0.2, batch_size=300, epoc
```

```
Train on 363923 samples, validate on 90981 samples
Epoch 1/20
 - 3s - loss: 0.0645 - acc: 0.9758 - val_loss: 0.0218 - val_acc: 0.9985
Epoch 2/20
 - 2s - loss: 0.0127 - acc: 0.9971 - val_loss: 0.0123 - val_acc: 0.9985
Epoch 3/20
 - 2s - loss: 0.0072 - acc: 0.9986 - val_loss: 0.0052 - val_acc: 0.9998
Epoch 4/20
 - 2s - loss: 0.0051 - acc: 0.9991 - val_loss: 0.0059 - val_acc: 0.9997
Epoch 5/20
 - 2s - loss: 0.0042 - acc: 0.9993 - val_loss: 0.0059 - val_acc: 1.0000
Epoch 6/20
 - 2s - loss: 0.0039 - acc: 0.9993 - val_loss: 0.0010 - val_acc: 1.0000
Epoch 7/20
 - 3s - loss: 0.0038 - acc: 0.9994 - val_loss: 0.0053 - val_acc: 0.9991
Epoch 8/20
 - 2s - loss: 0.0030 - acc: 0.9995 - val_loss: 0.0016 - val_acc: 1.0000
Epoch 9/20
 - 2s - loss: 0.0031 - acc: 0.9995 - val_loss: 2.1941e-04 - val_acc: 1.0000
Epoch 10/20
 - 2s - loss: 0.0025 - acc: 0.9996 - val_loss: 6.8718e-04 - val_acc: 1.0000
Epoch 11/20
 - 2s - loss: 0.0021 - acc: 0.9996 - val_loss: 9.9775e-04 - val_acc: 1.0000
Epoch 12/20
 - 2s - loss: 0.0020 - acc: 0.9996 - val_loss: 5.3142e-04 - val_acc: 1.0000
Epoch 13/20
 - 2s - loss: 0.0015 - acc: 0.9997 - val_loss: 3.2394e-04 - val_acc: 1.0000
Epoch 14/20
 - 2s - loss: 0.0016 - acc: 0.9997 - val_loss: 0.0016 - val_acc: 1.0000
Epoch 15/20
 - 2s - loss: 0.0016 - acc: 0.9997 - val_loss: 7.5209e-04 - val_acc: 1.0000
Epoch 16/20
 - 2s - loss: 0.0012 - acc: 0.9997 - val_loss: 4.3707e-04 - val_acc: 1.0000
Epoch 17/20
 - 2s - loss: 0.0014 - acc: 0.9997 - val_loss: 0.0048 - val_acc: 0.9985
Epoch 18/20
 - 2s - loss: 0.0014 - acc: 0.9997 - val_loss: 3.3824e-04 - val_acc: 1.0000
Epoch 19/20
 - 2s - loss: 0.0010 - acc: 0.9998 - val_loss: 0.0090 - val_acc: 0.9971
Epoch 20/20
 - 2s - loss: 0.0014 - acc: 0.9997 - val_loss: 2.4220e-04 - val_acc: 1.0000
```

Out[64]: <keras.callbacks.History at 0x135af7128>

In [65]:
```
oversample_predictions = oversample_model.predict(original_Xtest, batch_size=200, verl

oversample_fraud_predictions = oversample_model.predict_classes(original_Xtest, batch_
```

```
oversample_smote = confusion_matrix(original_ytest, oversample_fraud_predictions)
actual_cm = confusion_matrix(original_ytest, original_ytest)
labels = ['No Fraud', 'Fraud']

fig = plt.figure(figsize=(16,8))

fig.add_subplot(221)
plot_confusion_matrix(oversample_smote, labels, title="OverSample (SMOTE) \n Confusion

fig.add_subplot(222)
plot_confusion_matrix(actual_cm, labels, title="Confusion Matrix \n (with 100% accurac
```

```
Confusion matrix, without normalization
[[56816    47]
 [   19    79]]
Confusion matrix, without normalization
[[56863     0]
 [    0    98]]
```