PCA and Deep learning

Algorithm:

Principle Component Analysis is a dimensionality reduction technique. PCA works by projecting input data onto the eigenvectors of the data’s covariance matrix. The covariance matrix quantifies the variance of the data and how much each variable varies with respect to one another.

Eigenvectors are simply vectors that retain their span through a linear transformation, that is, they point in the same direction before and after the transformation. The covariance matrix transforms the original basis vectors to be oriented in the direction of the covariance between each variable. In simpler terms, the eigenvector allows us to re-frame the orientation of the original data to view it at a different angle without actually transforming the data. We are essentially extracting the component of each variable that leads to the most variance when we project the data onto these vectors. We can then select the dominant axes using the eigenvalues of the covariance matrix because they reflect the magnitude of the variance in the direction of their corresponding eigenvector.

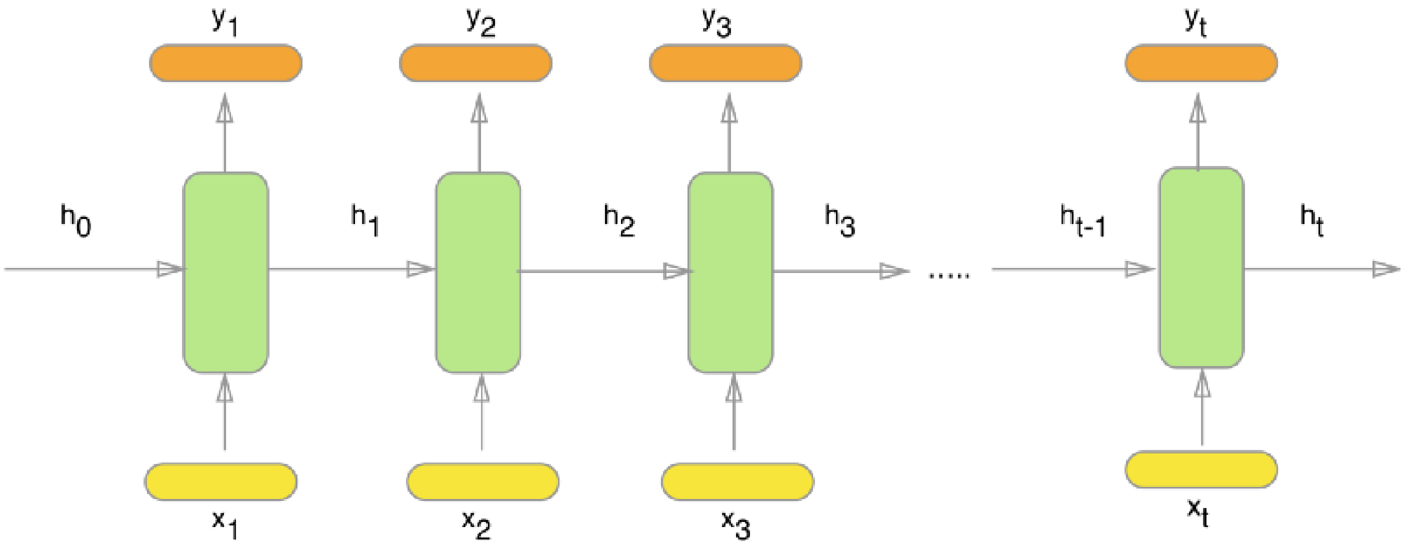
We want principal components to be oriented in the direction of maximum variance because greater variance in attribute values can lead to better forecasting abilities.

Pros of PCA: Reduces dimensionality, Interpretable, Fast run time

Cons of PCA: Incapable of learning non-linear feature representations, Deep Learning tend to overfit the data. By performing PCA, we restrict input data to relevant lower dimensional space which makes it easier to identify outliers

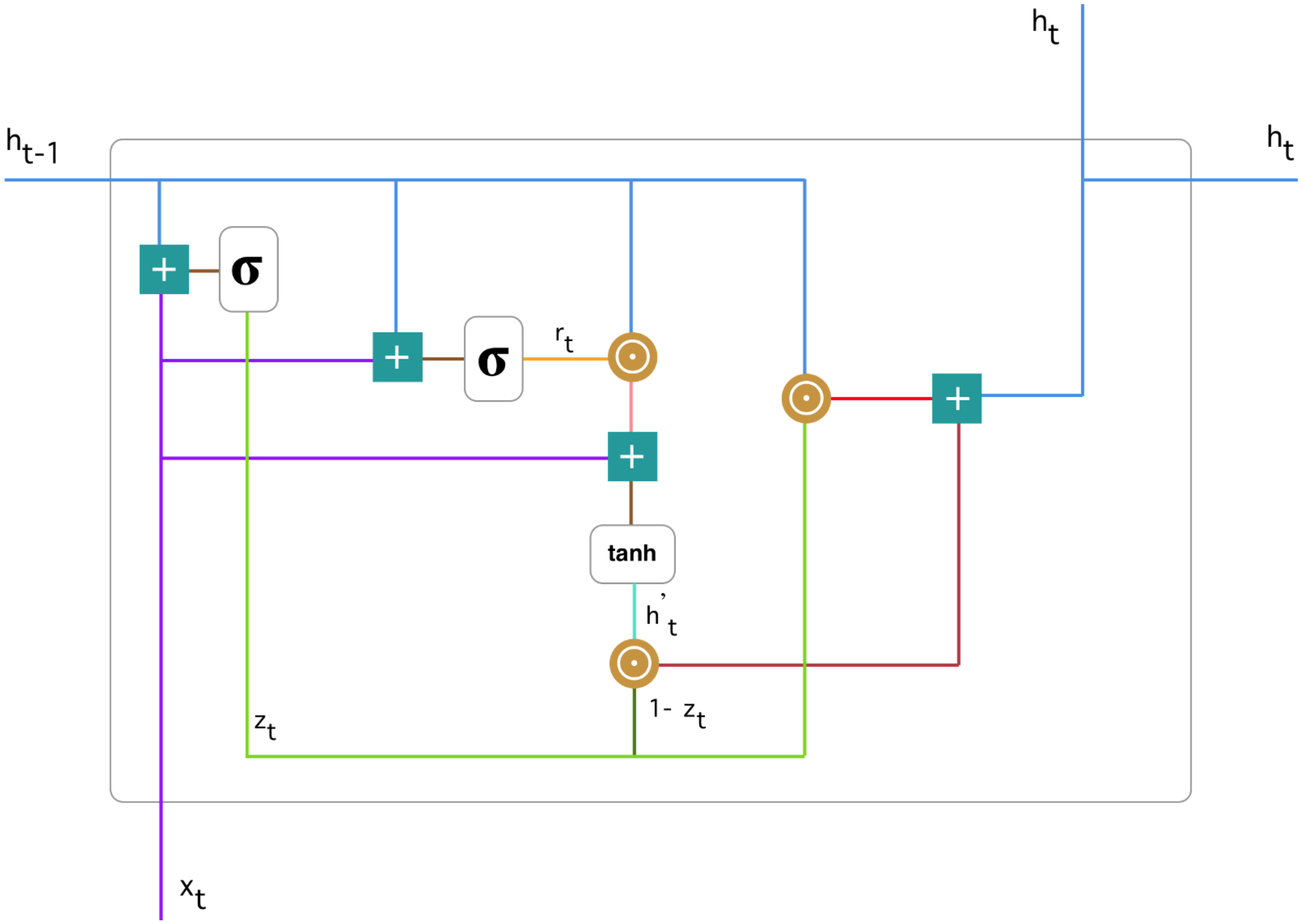
A limitation of Vanilla Neural Networks is their API is too constrained: they accept a fixed-sized vector as input and produce a fixed-sized vector as output (e.g. probabilities of different classes). Not only that: These models perform this mapping using a fixed amount of computational steps (e.g. the number of layers in the model). The core reason that recurrent nets are more exciting is that they allow us to operate over sequences of vectors: Sequences in the input, the output, or in the most general case both. GRU, introduced by Cho, et al. in 2014, GRU (Gated Recurrent Unit) aims to solve the vanishing gradient problem which comes with a standard recurrent neural network. GRU can also be considered as a variation on the LSTM because both are designed similarly and, in some cases, produce equally excellent results.

Simple RNN



To solve the vanishing gradient problem of a standard RNN, GRU uses, so called, update gate and reset gate. Basically, these are two vectors which decide what information should be passed to the output. The special thing about them is that they can be trained to keep information from long ago, without washing it through time or remove information which is irrelevant to the prediction.

A single GRU


h t
h t-1
h t
σ
σ
r t
tanh
h' t
z t
1- z t
x t

In [1]:
```
import time
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import tensorflow as tf
from tensorflow import keras
from keras import optimizers
from keras.models import Sequential
from keras.layers import Input, Dense, Dropout, LSTM, GRU
from keras.models import Model, load_model
from keras.callbacks import ModelCheckpoint, TensorBoard
from keras import regularizers
%matplotlib inline

from sklearn.decomposition import PCA
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, precision_recall_curve
from sklearn.metrics import recall_score, classification_report, auc, roc_curv
e
from sklearn.metrics import precision_recall_fscore_support, f1_score
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import roc_auc_score
import itertools
```

```
Using TensorFlow backend.
```

In [2]:
```
epochs=20
batch_size=4096
train_split = 0.80
nrows = 1_000_000
path = 'c:/users/ugy1/abs/'
df=pd.read_csv(path+'datasets/processed_abs_loan_'+str(nrows)+'.csv',
               #usecols=use_list,
               #sep='\t',
               #compression=bz2,
               nrows=nrows,
               low_memory=False,
              index_col=0,
               parse_dates=True
              )
df.shape
```

Out[2]:
```
(913751, 85)
```

In [3]:
```
column_list=df.columns.tolist()
```

In [4]:
```
df.head()
```

Out[4]:

	originalloanamount	originalloanterm	originalinterestratepercentage	graceperiodnuml
0	11940.46	60	0.2149	1
1	17501.22	62	0.0190	2
2	13310.93	72	0.1897	2
3	21427.33	72	0.0651	1
4	6200.00	60	0.1868	1

5 rows × 85 columns

In [5]:
```
Y=df.label.values
Y.shape
```

Out[5]: (913751,)

In [6]:
```
input=df.values
input.shape
```

Out[6]: (913751, 85)

In [7]:
```
# calculate train/test split
len_train = int(len(input)*train_split)
print(len_train)
```

```
731000
```

In [8]:
```
# apply train/test split to labels
y_train = Y[0:len_train]
y_test = Y[len_train:]
x_train = input[0:len_train]
x_test = input[len_train:]
x_train.shape
```

Out[8]: (731000, 85)

In [9]:
```
export_x_test = pd.DataFrame(data=x_test)
```

In [10]:
```
export_x_test.columns=column_list
export_x_test.rename(columns={'label':'True Label'}, inplace=True)
export_x_test.head()
```

Out[10]:

	originalloanamount	originalloanterm	originalinterestratepercentage	graceperiodnumb
0	15634.45	72.0	0.1823	1.0
1	21551.70	60.0	0.1980	2.0
2	66580.36	60.0	1.9000	1.0
3	32845.00	72.0	12.5000	2.0
4	58840.50	72.0	1.0000	0.0

5 rows × 85 columns

In [11]:
```
#from sklearn.preprocessing import MinMaxScaler
# from sklearn.preprocessing import minmax_scale
# from sklearn.preprocessing import MaxAbsScaler
#from sklearn.preprocessing import StandardScaler
# from sklearn.preprocessing import RobustScaler
# from sklearn.preprocessing import Normalizer
# from sklearn.preprocessing import QuantileTransformer
# from sklearn.preprocessing import PowerTransformer
```

In [12]:
```
x_scaler=StandardScaler()
x_train = x_scaler.fit_transform(x_train)
x_test = x_scaler.fit_transform(x_test)
```

In [13]:
```
# x_train = keras.utils.normalize(x_train, axis=-1, order=2)
# x_test = keras.utils.normalize(x_test, axis=-1, order=2)
# x_train.shape
```

Principal Component Analysis

In [14]:
```
x_train = PCA(n_components=30, svd_solver='full').fit_transform(x_train)
x_test = PCA(n_components=30, svd_solver='full').fit_transform(x_test)
```

In [15]:
```
x_train.shape
```

Out[15]: (731000, 30)

In [16]:
```
x_test.shape
```

Out[16]: (182751, 30)

In [17]:
```
# x_scaler=StandardScaler()
# x_train = x_scaler.fit_transform(x_train)
# x_test = x_scaler.fit_transform(x_test)
```

In [18]:
```
# reshape for deep learning
#input=input.reshape(input.shape[0], input.shape[1], 1)
x_train=x_train.reshape(x_train.shape[0], x_train.shape[1], 1)
x_test=x_test.reshape(x_test.shape[0], x_test.shape[1], 1)
y_train=y_train.reshape(y_train.shape[0],1)
y_test=y_test.reshape(y_test.shape[0],1)
```

In [19]:
```
model = keras.Sequential()
model.add(keras.layers.GRU(254, activation='relu',
                           kernel_regularizer=regularizers.l2(0.01),
                           input_shape=(x_train.shape[1:]),
                           return_sequences=True))
model.add(keras.layers.Dropout(0.1))

model.add(keras.layers.GRU(128, activation='relu', kernel_regularizer=regulari
zers.l2(0.01)))
model.add(keras.layers.Dropout(0.2))

model.add(keras.layers.Dense(32, activation='relu',  kernel_regularizer=regula
rizers.l2(0.01)))
model.add(keras.layers.Dropout(0.1))

model.add(keras.layers.Dense(16, activation='relu',  kernel_regularizer=regula
rizers.l2(0.01)))
model.add(keras.layers.Dropout(0.1))

model.add(keras.layers.Dense(16, activation='relu',  kernel_regularizer=regula
rizers.l2(0.01)))


model.add(keras.layers.Dense(2, activation='softmax'))

optimizer = tf.keras.optimizers.Adam(lr=1e-3, decay=1e-6)
model.summary()
```

Layer (type)	Output Shape	Param #
gru (GRU)	(None, 30, 254)	195072
dropout (Dropout)	(None, 30, 254)	0
gru_1 (GRU)	(None, 128)	147072
dropout_1 (Dropout)	(None, 128)	0
dense (Dense)	(None, 32)	4128
dropout_2 (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
dropout_3 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 16)	272
dense_3 (Dense)	(None, 2)	34

```
Total params: 347,106
Trainable params: 347,106
Non-trainable params: 0
```

In [20]:
```
checkpoint = ModelCheckpoint(filepath="./model/PCA_GRU_abs_loan.h5",
                               save_best_only=True,
                               verbose=0)
tensorboard = TensorBoard(log_dir='./logs',
                histogram_freq=0,
                write_graph=True,
                write_images=True)

model.compile(loss='sparse_categorical_crossentropy', optimizer=optimizer, met
rics=['accuracy'])

model.fit(x_train, y_train, epochs=epochs, validation_data=(x_test, y_test), b
atch_size=batch_size,
        callbacks = [
            checkpoint,
   # baseLogger,
    #history,
    #tensorboard,
    #LearningRateScheduler,
    #reduceLROnPlateau
         ],
          shuffle=False
)
```

```
Train on 731000 samples, validate on 182751 samples
Epoch 1/20
731000/731000 [==============================] - 48s 66us/step - loss: 1.1856
- acc: 0.9126 - val_loss: 0.3349 - val_acc: 0.9114
Epoch 2/20
731000/731000 [==============================] - 43s 58us/step - loss: 0.2000
- acc: 0.9324 - val_loss: 0.1441 - val_acc: 0.9824
Epoch 3/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.2513
- acc: 0.9422 - val_loss: 0.3171 - val_acc: 0.9114
Epoch 4/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.2971
- acc: 0.9127 - val_loss: 0.1649 - val_acc: 0.9114
Epoch 5/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0704
- acc: 0.9756 - val_loss: 0.0688 - val_acc: 0.9771
Epoch 6/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0686
- acc: 0.9762 - val_loss: 0.0989 - val_acc: 0.9825
Epoch 7/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0258
- acc: 0.9957 - val_loss: 0.0371 - val_acc: 0.9870
Epoch 8/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0132
- acc: 0.9977 - val_loss: 0.0279 - val_acc: 0.9897
Epoch 9/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0174
- acc: 0.9965 - val_loss: 0.0446 - val_acc: 0.9778
Epoch 10/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0092
- acc: 0.9987 - val_loss: 0.0380 - val_acc: 0.9825
Epoch 11/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0062
- acc: 0.9991 - val_loss: 0.0250 - val_acc: 0.9908
Epoch 12/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0770
- acc: 0.9718 - val_loss: 0.1050 - val_acc: 0.9114
Epoch 13/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0601
- acc: 0.9831 - val_loss: 0.0439 - val_acc: 0.9874
Epoch 14/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0082
- acc: 0.9989 - val_loss: 0.0161 - val_acc: 0.9944
Epoch 15/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0060
- acc: 0.9992 - val_loss: 0.0146 - val_acc: 0.9967
Epoch 16/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0059
- acc: 0.9991 - val_loss: 0.0099 - val_acc: 0.9973
Epoch 17/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0048
- acc: 0.9994 - val_loss: 0.0079 - val_acc: 0.9977
Epoch 18/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0048
- acc: 0.9994 - val_loss: 0.0093 - val_acc: 0.9973
Epoch 19/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0046
```

```
- acc: 0.9994 - val_loss: 0.0116 - val_acc: 0.9968
Epoch 20/20
731000/731000 [==============================] - 43s 59us/step - loss: 0.0043
- acc: 0.9995 - val_loss: 0.0172 - val_acc: 0.9964
```

Out[20]: <tensorflow.python.keras.callbacks.History at 0x17738291cf8>

In [21]:
```
model=load_model("./model/PCA_GRU_abs_loan.h5")
```

In [22]:
```
x_pred = x_test
```

In [23]:
```
prediction_gru = model.predict_classes(x_pred)
```

In [24]:
```
prediction_gru.shape
```

Out[24]: (182751,)

In [25]:
```
export_x_test['Predicted Label']=prediction_gru
```

In [26]:
```
export_x_test.head()
```

Out[26]:

	originalloanamount	originalloanterm	originalinterestratepercentage	graceperiodnuml
0	15634.45	72.0	0.1823	1.0
1	21551.70	60.0	0.1980	2.0
2	66580.36	60.0	1.9000	1.0
3	32845.00	72.0	12.5000	2.0
4	58840.50	72.0	1.0000	0.0

5 rows × 86 columns

In [27]:
```
export_x_test.shape
```

Out[27]: (182751, 86)

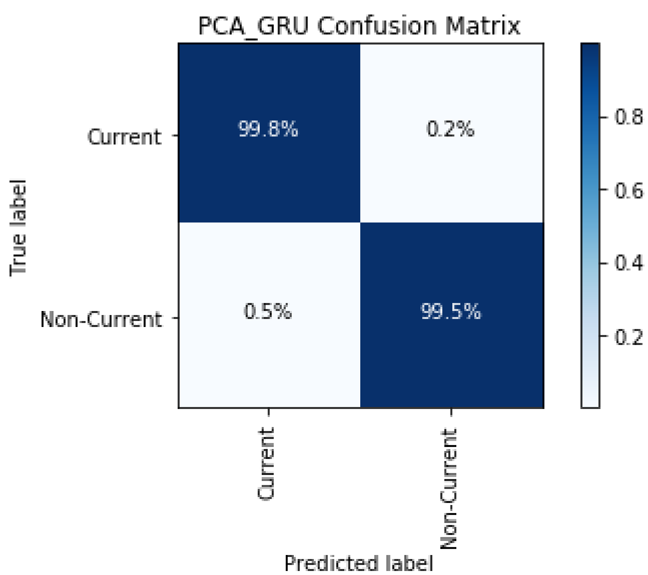
In [28]:
```
export_x_test.to_csv(path+"prediction/gru/predicated_PCA_gru_abs_loans_"+str(n
rows)+".csv", chunksize=10000)
```

In [29]:
```
def plot_confusion_matrix(cm, title, classes=['Current', 'Non-Current'],
                          cmap=plt.cm.Blues, save=False, saveas="MyFigure.png"
):

    # print Confusion matrix with blue gradient colours

    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=90)
    plt.yticks(tick_marks, classes)

    fmt = '.1%'
    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')

    if save:
        plt.savefig(saveas, dpi=100)
```

In [30]:
```
def plot_gridsearch_cv(results, estimator, x_min, x_max, y_min, y_max,save=False, saveas="MyFigure.png"):

    # print GridSearch cross-validation for parameters

    plt.figure(figsize=(10,8))
    plt.title("GridSearchCV for "+estimator, fontsize=24)

    plt.xlabel(estimator)
    plt.ylabel("Score")
    plt.grid()

    ax = plt.axes()
    ax.set_xlim(x_min, x_max)
    ax.set_ylim(y_min, y_max)

    pad = 0.005
    X_axis = np.array(results["param_"+estimator].data, dtype=float)

    for scorer, color in zip(sorted(scoring), ['b', 'k']):
        for sample, style in (('train', '--'), ('test', '-')):
            sample_score_mean = results['mean_%s_%s' % (sample, scorer)]
            sample_score_std = results['std_%s_%s' % (sample, scorer)]
            ax.fill_between(X_axis, sample_score_mean - sample_score_std,
                            sample_score_mean + sample_score_std,
                            alpha=0.1 if sample == 'test' else 0, color=color)
            ax.plot(X_axis, sample_score_mean, style, color=color,
                alpha=1 if sample == 'test' else 0.7,
                label="%s (%s)" % (scorer, sample))

        best_index = np.nonzero(results['rank_test_%s' % scorer] == 1)[0][0]
        best_score = results['mean_test_%s' % scorer][best_index]

        # Plot a dotted vertical line at the best score for that scorer marked by x
        ax.plot([X_axis[best_index], ] * 2, [0, best_score],
            linestyle='-.', color=color, marker='x', markeredgewidth=3, ms=8)

        # Annotate the best score for that scorer
        ax.annotate("%0.2f" % best_score,
                (X_axis[best_index], best_score+pad))

    plt.legend(loc="best")
    plt.grid('off')
    plt.tight_layout()
    if save:
        plt.savefig(saveas, dpi=100)

    plt.show()
```

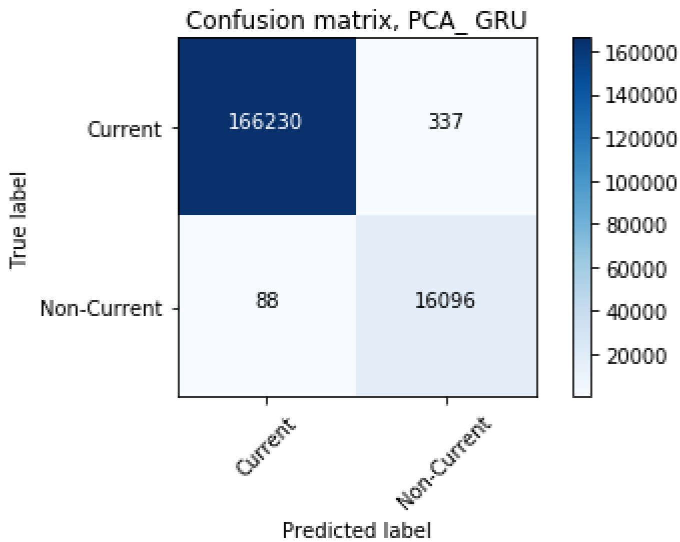
In [31]:
```
print(classification_report(y_test, prediction_gru, target_names=['Current',
'Non_Current']))
print ("AUC: ", "{:.1%}".format(roc_auc_score(y_test, prediction_gru)))
cm = confusion_matrix(y_test, prediction_gru)
plot_confusion_matrix(cm, title="PCA_GRU Confusion Matrix",save=True,
                      saveas='prediction/gru/cm'+str(' PCA_GRU Accuracy-')+str
(nrows)+'.png')
```

```
             precision    recall  f1-score   support

    Current       1.00      1.00      1.00    166567
Non_Current       0.98      0.99      0.99     16184

avg / total       1.00      1.00      1.00    182751

AUC:  99.6%
```



In [32]:
```
class_names = ['Current', 'Non-Current']

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

    print(cm)

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

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

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


print('ROC_AUC_SCORE ; ', roc_auc_score(y_test, prediction_gru))
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, prediction_gru)
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, title= 'Confusion matri
x, PCA_ GRU')
plt.savefig('prediction/gru/cm'+str(' PCA_GRU Prediction-')+str(nrows)+'.png')
plt.show()
```

```
ROC_AUC_SCORE ;  0.9962696605076014
Confusion matrix, without normalization
[[166230    337]
 [    88  16096]]
```



In [ ]: