The purpose of this notebook is to demonstrate the use of autoencoding to extract relevant data from a signal.

Training usually consists of taking data that has been manually classified, and using it to train an artificial neural network (ann). When each example contains large amounts of data, we sometimes extract features before training (transformation), or use multi-layer ann to automatically extract features.

The former has issues because we may discard important information that takes place at a tiny detailed scale of the data, in favor of dimensionality reduction.

The latter can be cumbersome and time consuming for two reasons; first due to the large size of input layers - because we may need information from the small scale, and second due to the large depth of the ann - because large scale info may be important to correctly classify the data.

Instead, this notebook proposes to train a artificial neural network to simply compress the data, without regard to the manual classification - thereby learning the patterns that are common to all of the training signals. We then subtract this "common data" from the individual training signal, and only use the remaining portion (the residual) to perform classification.

Theory:

You are probably familiar with existing methods to extract basic time-varying signal information through the use of transformation. **Fourier** transformation is often used for periodic signals, where the convolution is with various period sine and cosine functions. **Wavelet** transformation is often used for time limited signals, where convolution is with a wavelet shape that has been selected by hand. Note that with wavelet decomposition, the scale of the wavelet shape is changed, similar to the way the period of the Fourier functions are changed. **Convolutional** networks seek to learn a set of optimal wavelets. Unlike traditional wavelet transformation, convolutional networks do not change the scale of the wavelet shape, and instead rely on layers of the network to learn patterns of various scale. In all of these cases, a compressed "lossy" representation of the original signal is created. Dense networks can also be used to learn patterns within time varying signals, and also create a compressed version of the original signal.

A **Dense** neural network layer is used to compress the original signal of nearly 1MB down to a few floating point numbers (less than 1kB). Using this compressed information, a lossy version of the original signal can be reconstructed. The difference between the original signal, and the reconstructed signal is called the **Residual**. The process can be repeated in an attempt to compress the residual, similar to the method of successive wavelet decomposition.

Along the way, we can also see interesting patterns emerge. By using visualization, we can compare the original signal to successive de-compressed versions, and see what is common across all signals, and also what is different between them.

Revision Info:

PAN Dec. 21, 2018

Forked from Panchajanya Banerjee (Pancham) - First Steps EDA https://www.kaggle.com/delayedkarma/first-steps-eda (https://www.kaggle.com/delayedkarma/first-steps-eda)

V09 fixed typo in feature extraction code

V08 - tuned dense nn

V07 switched from random forest as final classifier to Dense NN

V05 and V06 fixing memory overflow errors in Kaggle, added comments to make it more "educational" (hopefully)

V04 - Improved residual compression MSE by using multi-layer, added better feature extraction of 2nd residual using example code from VSB Power LSTM attention https://www.kaggle.com/braquino/vsb-power-lstm-attention (https://www.kaggle.com/braquino/vsb-power-lstm-attention) by Bruno Aquino

V02 and V03 - Added 2nd residual features

V01 - Intended for release to the public as an educational tool Jan 2019

Training and Testing Data Info:

The data being used - comes from the VSB Power contest on kaggle.com (January, 2019)

Each id_measurement [train 0-2903, test 2904-9682] consists of three signal_id's [train 0-8711, test 8712-29048] where each signal_id is associated with one of three phase's [0-2] also, the training set also provides a target [0-1] for each signal_id indicating a fault or not

There are 800,000 int8 samples for each signal_id [train 0-8711, test 8712-29048]

Possible next steps:

Sloppy "cut and paste" coding should be replaced by function calls

The final residual data can be further analyzed for feature extraction

If enough folks are interested, I can make the documentation more rigorous with citations and formulas

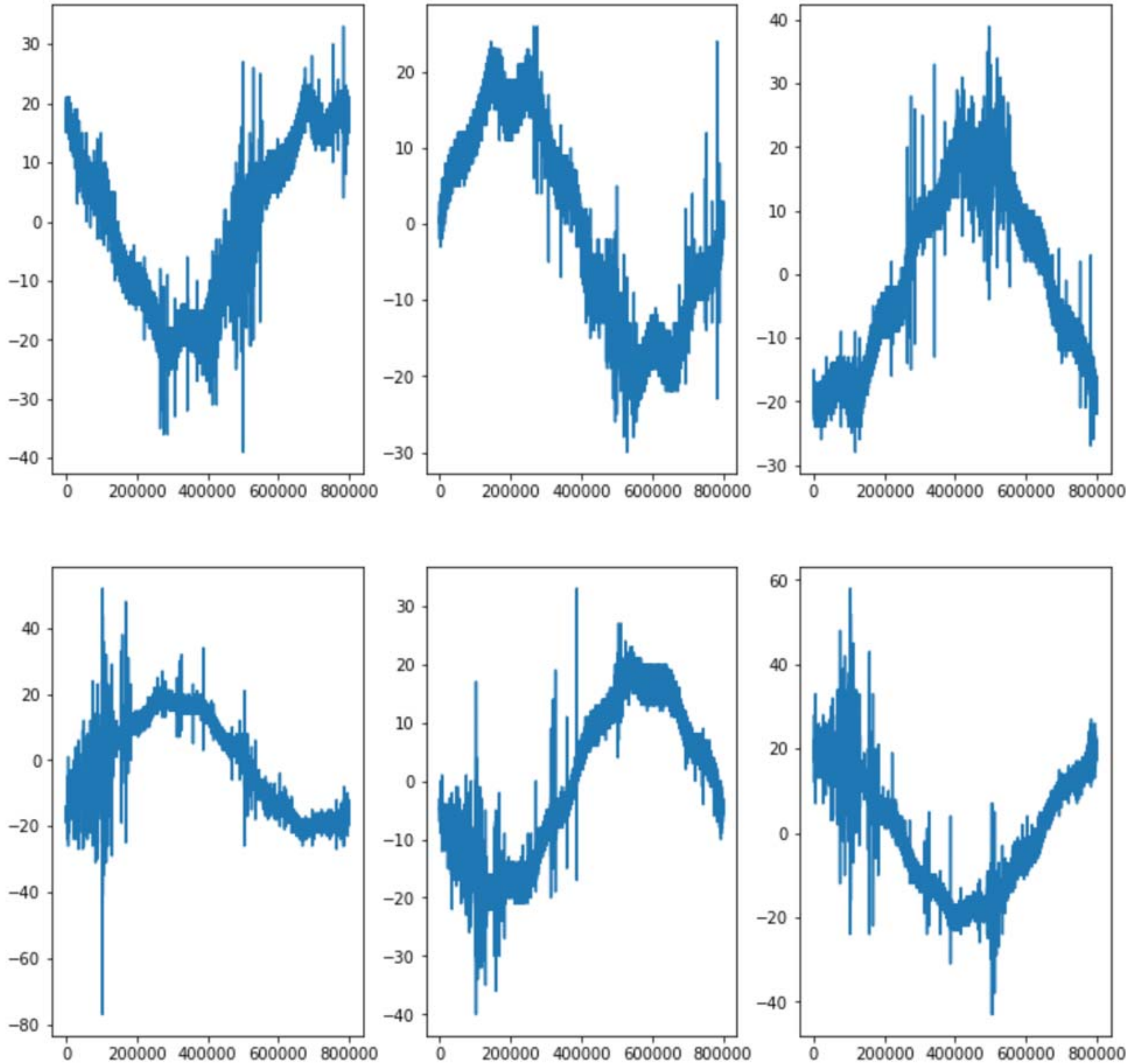
In [1]:

```
# LOADING UP PYTHON COMPONENTS
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will
 list the files in the input directory
import pyarrow.parquet as pq
import os
print(os.listdir("../input"))

import matplotlib.pyplot as plt
import seaborn as sns

import tensorflow as tf
from tensorflow.python.keras.models import Sequential, Model, load_model
from tensorflow.python.keras.layers import Dense
```

```
['test.parquet', 'sample_submission.csv', 'metadata_train.csv', 'trai
n.parquet', 'metadata_test.csv']
```

In [2]:

```
# VIEWING THE FIRST SIX TRAINING SIGNALS
# The first row of three shows the three phases of a power line that has been hand-classified as not having a fault
# The second row of three shows the three phases of a power line that has a fault
meta_train = pd.read_csv('../input/metadata_train.csv')
# %%time
# Read in the first two signals (three phases each) for display.
# Each column contains one signal
subset_train = pq.read_pandas('../input/train.parquet', columns=[str(i) for i in range(6)]).to_pandas()
# Comparing a good signal (forst row, all three phases)
# with a bad signal (second row, all three phases)
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2,3, figsize=(12,12))
ax1.plot(subset_train['0']) ;
ax2.plot(subset_train['1']) ;
ax3.plot(subset_train['2']) ;
ax4.plot(subset_train['3']) ;
ax5.plot(subset_train['4']) ;
ax6.plot(subset_train['5']) ;
```


30
20
10
0
−10
−20
−30
−40
0 200000 400000 600000 800000
20
10
0
−10
−20
−30
0 200000 400000 600000 800000
40
30
20
10
0
−10
−20
−30
0 200000 400000 600000 800000
40
20
0
−20
−40
−60
−80
0 200000 400000 600000 800000
30
20
10
0
−10
−20
−30
−40
0 200000 400000 600000 800000
60
40
20
0
−20
−40
0 200000 400000 600000 800000

In [3]:

```
# HAND-CLASSIFIED INFORMATION ABOUT THE ABOVE SIGNALS
print("Classifications of above")
meta_train[0:6]
```

```
Classifications of above
```

Out[3]:

	signal_id	id_measurement	phase	target
0	0	0	0	0
1	1	0	1	0
2	2	0	2	0
3	3	1	0	1
4	4	1	1	1
5	5	1	2	1

In [4]:

```
%%time
### Load Raw Training Data
begin_col = 0
# number of test examples to use
# num_to_use = 8712
# use a smaller subset because of Kaggle RAM limitations
num_to_use = 2001
# num_to_use = 30

filename = '../input/train.parquet'

X = pq.read_pandas(filename, columns=[str(j + begin_col) for j in range(n
um_to_use)]).to_pandas().values.transpose()
```

```
CPU times: user 9.95 s, sys: 1.88 s, total: 11.8 s
Wall time: 11.9 s
```

In [5]:

```
# the data we will use to train the autoencoder
X.shape
```

Out[5]:

```
(2001, 800000)
```

In [6]:

```
# The autoencoder network
compression_1 = 5

model = Sequential()
model.add(Dense(compression_1, activation='relu', input_shape=(800000,),
name='compress'))
# this output layer has to have 800,000 neurons, and needs to be linearly a
ctivated
model.add(Dense(800000, activation='linear'))
model.compile(loss='mean_squared_error', optimizer='adam')
# Diplay the model summary
print("model summary")
model.summary()
```

```
model summary
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
compress (Dense)             (None, 5)                 4000005
_________________________________________________________________
dense (Dense)                (None, 800000)            4800000
=================================================================
Total params: 8,800,005
Trainable params: 8,800,005
Non-trainable params: 0
_________________________________________________________________
```

In [7]:

```
# Train the autoencoder to minimize mean square error
from keras.callbacks import EarlyStopping, ModelCheckpoint

earlystopper = EarlyStopping(patience=2, verbose=1)
checkpointer = ModelCheckpoint('VSBautoassoc', verbose=1, save_best_only=
True)
results = model.fit(X, X, validation_split=0.1, batch_size=50, epochs=300
,
                    callbacks=[earlystopper, checkpointer])
```

```
Using TensorFlow backend.


Train on 1800 samples, validate on 201 samples
Epoch 1/300
1750/1800 [============================>.] - ETA: 0s - loss: 120.4851
Epoch 00001: val_loss improved from inf to 34.66722, saving model to V
SBautoassoc
1800/1800 [==============================] - 25s 14ms/step - loss: 11
8.0996 - val_loss: 34.6672
Epoch 2/300
1750/1800 [============================>.] - ETA: 0s - loss: 18.3891
Epoch 00002: val_loss improved from 34.66722 to 6.89925, saving model
to VSBautoassoc
1800/1800 [==============================] - 23s 13ms/step - loss: 18.
0858 - val_loss: 6.8993
Epoch 3/300
1750/1800 [============================>.] - ETA: 0s - loss: 6.1549
Epoch 00003: val_loss improved from 6.89925 to 5.56623, saving model t
o VSBautoassoc
1800/1800 [==============================] - 23s 13ms/step - loss: 6.1
274 - val_loss: 5.5662
Epoch 4/300
1750/1800 [============================>.] - ETA: 0s - loss: 5.4621
Epoch 00004: val_loss improved from 5.56623 to 5.26071, saving model t
o VSBautoassoc
1800/1800 [==============================] - 23s 13ms/step - loss: 5.4
751 - val_loss: 5.2607
Epoch 5/300
1750/1800 [============================>.] - ETA: 0s - loss: 5.3269
Epoch 00005: val_loss did not improve from 5.26071
1800/1800 [==============================] - 23s 13ms/step - loss: 5.3
166 - val_loss: 5.3872
Epoch 6/300
1750/1800 [============================>.] - ETA: 0s - loss: 5.3719
Epoch 00006: val_loss did not improve from 5.26071
1800/1800 [==============================] - 23s 13ms/step - loss: 5.3
550 - val_loss: 5.3411
Epoch 00006: early stopping
```

In [8]:
```
model = load_model('VSBautoassoc')
```

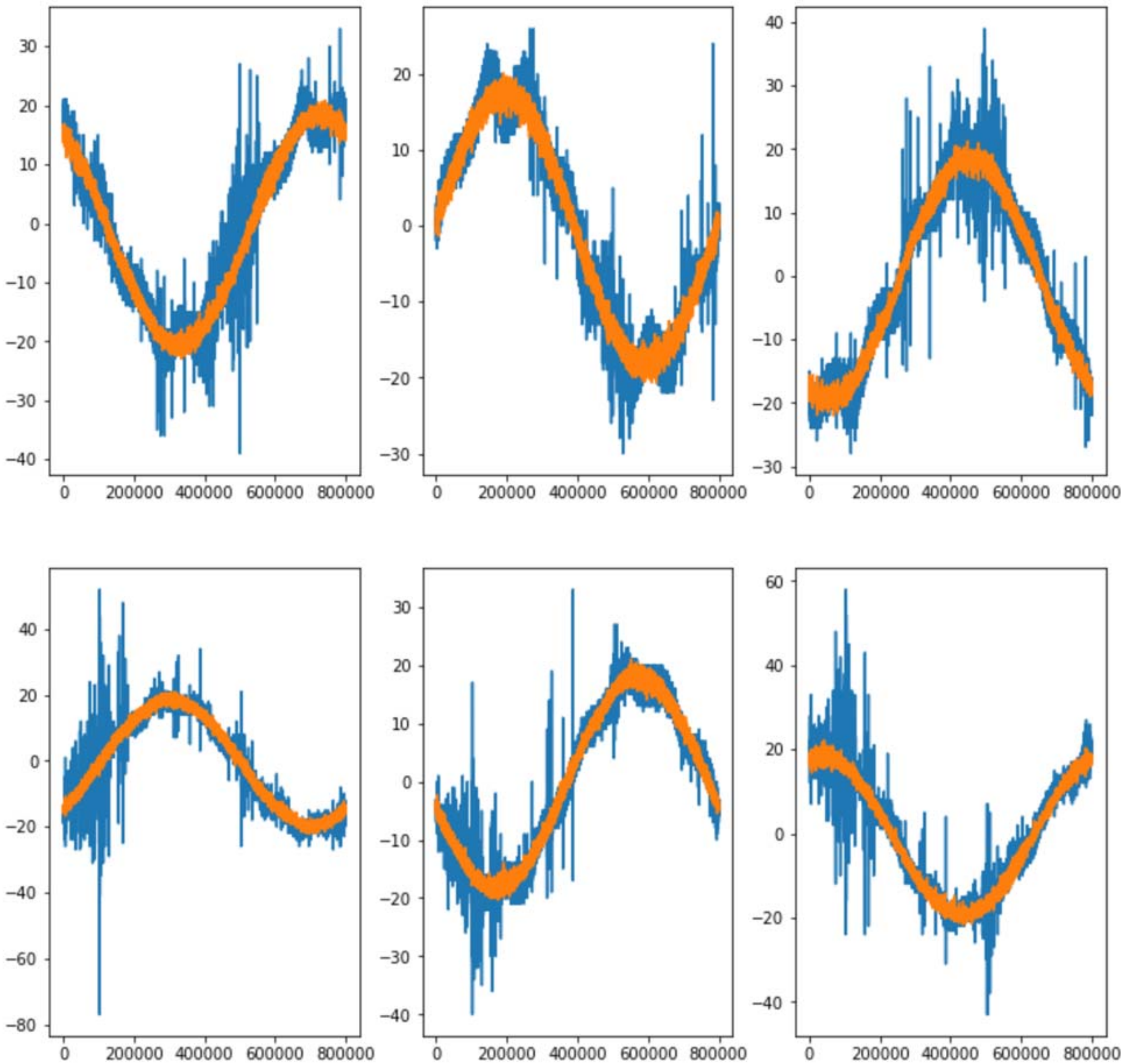
In [9]:
```
# The front half of the autoencoder is the "coder" part of this CODEC pair
# We can use the coder to convert the very large signal data (800000 intege
rs) into a much smaller compressed version
# Load just the compression model
c_model = Sequential()
c_model.add(Dense(compression_1, activation='relu', input_shape=(800000
,), name='compress'))

c_model.load_weights('VSBautoassoc', by_name=True)
```

In [10]:
```
# Since the autoencoder both codes and decodes, we can use it to create the
lossy approximation of the original signal
# Looking at those first six signals again, overlaying their compressed ver
sion on top of the original signal
X_decompressed = model.predict(X[0:6])
X_decompressed.shape
```

Out[10]:
```
(6, 800000)
```

In [11]:

```
# Here we are comparing the original signal to its lossy reconstruction
# Notice how only the elements most common to ALL the signals in the traini
ng set are represented
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2,3, figsize=(12,1
2))
ax1.plot(subset_train['0']) ;
ax2.plot(subset_train['1']) ;
ax3.plot(subset_train['2']) ;
ax4.plot(subset_train['3']) ;
ax5.plot(subset_train['4']) ;
ax6.plot(subset_train['5']) ;
ax1.plot(X_decompressed[0]) ;
ax2.plot(X_decompressed[1]) ;
ax3.plot(X_decompressed[2]) ;
ax4.plot(X_decompressed[3]) ;
ax5.plot(X_decompressed[4]) ;
ax6.plot(X_decompressed[5]) ;
```



In [12]:

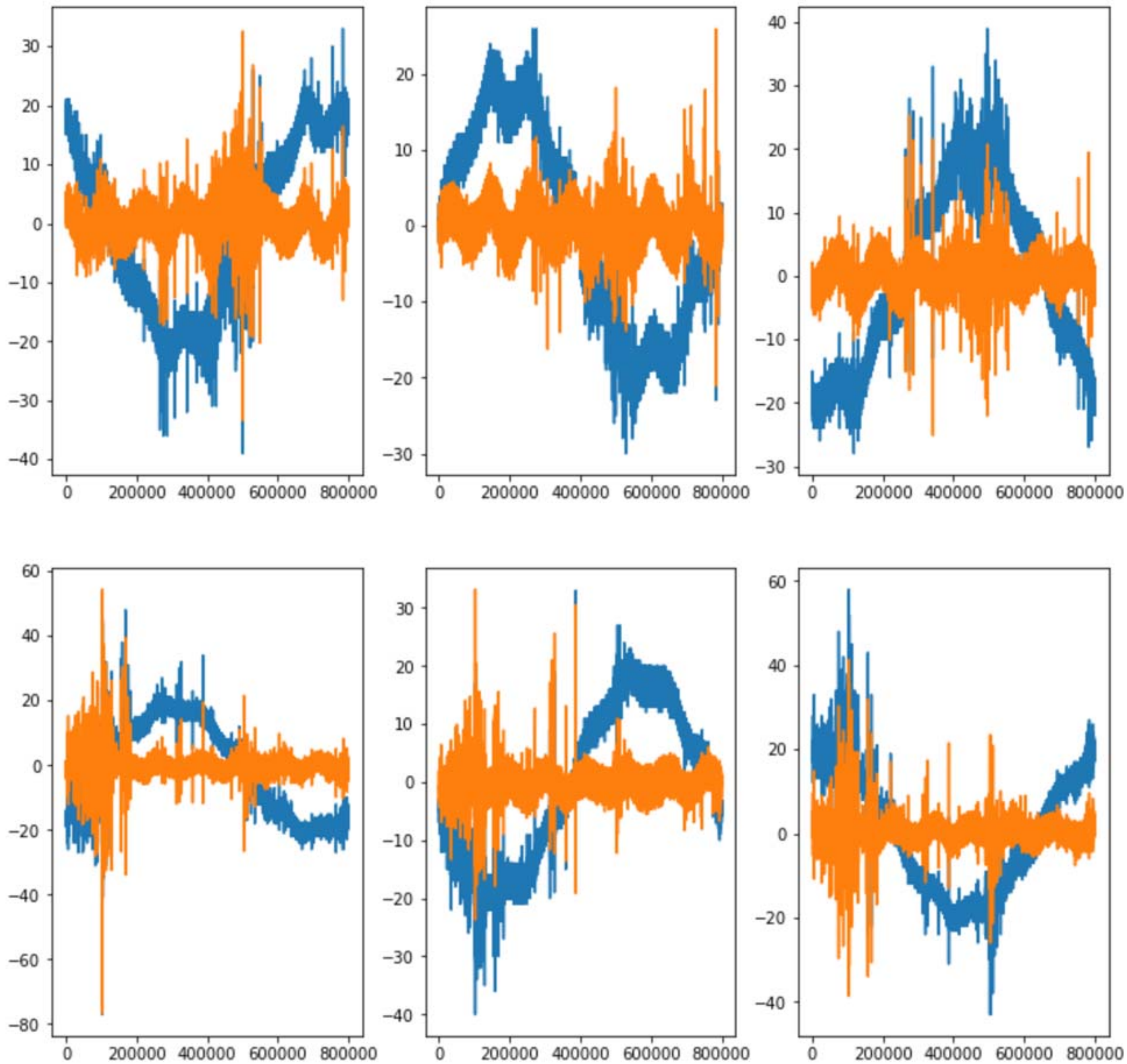
```
# the residual is what remains after subtracting the recreated signal from
 the original.
# In theorythis should contain the information that makes the training sign
als DIFFERENT from each other
# we will be repeating this iteratively, but for now, let's look at the res
idual
X_residual = X[0:6] - X_decompressed[0:6]
X_residual.shape
```

Out[12]:

```
(6, 800000)
```

In [13]:

```
# Here we see the residual superimposed on the original signal
# As expected, the residual removes the unimportant information, and leaves
the possibly important noise spikes
# Nevertheless, a strange seventh harmonic (ripple with seven peaks) shows
 up on these first two sets of 3-phase samples (?)
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2,3, figsize=(12,1
2))
ax1.plot(subset_train['0']) ;
ax2.plot(subset_train['1']) ;
ax3.plot(subset_train['2']) ;
ax4.plot(subset_train['3']) ;
ax5.plot(subset_train['4']) ;
ax6.plot(subset_train['5']) ;
ax1.plot(X_residual[0]) ;
ax2.plot(X_residual[1]) ;
ax3.plot(X_residual[2]) ;
ax4.plot(X_residual[3]) ;
ax5.plot(X_residual[4]) ;
ax6.plot(X_residual[5]) ;
```



In [14]:

```
# we will now create a new autoencoder, to compress those residual signals
# We are repeating this a second time in case there is still useless inform
ation that is very common between all signals
# There is little risk of "throwing away the baby with the bath water" beca
use the "codes" or compressed version of the info
# will still be kept and used for classification training.
#
# so we begin by gathering all the residuals
### CONVERT X into FIRST Residuals (to save memory)
X.shape
begin_col = 0
batch = 600
num_batches = int((num_to_use) / batch)
remainder = int((num_to_use) % batch)
X_decompressed = np.zeros([batch, 800000])
if (num_batches > 0) :
    for ix in range (num_batches) :
        X_decompressed[int(0):int(batch)] = model.predict(X[int(ix*batch
):int((ix+1)*batch)])
        X[int(ix*batch):int((ix+1)*batch)] = X[int(ix*batch):int((ix+1)*b
atch)] - X_decompressed[int(0):int(batch)]
if (remainder > 0) :
    ix = num_batches
    X_decompressed[int(0):int(remainder)] = model.predict(X[int(ix*batch
):int((ix*batch)+remainder)])
    X[int(ix*batch):int((ix*batch)+remainder)] = X[int(ix*batch):int((ix*
batch)+remainder)] - X_decompressed[int(0):int(remainder)]

X_decompressed = 5 # quick and dirty data=cleanup
X.shape
```

Out[14]:

```
(2001, 800000)
```

In [15]:
```
# The residuals are a bit more complex than the original signals, and so w
e'll use a multilayer network for that compression
### As with the original signal, we compress the residuals using an autoenc
oder
compression_2 = 20

r_model = Sequential()
r_model.add(Dense(compression_2 * 2, activation='relu', input_shape=(8000
00,), name='layer1'))
r_model.add(Dense(compression_2 * 4, activation='relu', name='layer2'))
r_model.add(Dense(compression_2, activation='relu', name='compressed'))
r_model.add(Dense(compression_2 * 4, activation='relu', name='layer4'))
r_model.add(Dense(compression_2 * 2, activation='relu', name='layer5'))
r_model.add(Dense(800000, activation='linear', name='output'))

r_model.compile(loss='mean_squared_error', optimizer='adam')
# Diplay the model summary
print("model summary")
r_model.summary()
```

```
model summary
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
layer1 (Dense)               (None, 40)                32000040  
_________________________________________________________________
layer2 (Dense)               (None, 80)                3280      
_________________________________________________________________
compressed (Dense)           (None, 20)                1620      
_________________________________________________________________
layer4 (Dense)               (None, 80)                1680      
_________________________________________________________________
layer5 (Dense)               (None, 40)                3240      
_________________________________________________________________
output (Dense)               (None, 800000)            32800000  
=================================================================
Total params: 64,809,860
Trainable params: 64,809,860
Non-trainable params: 0
_________________________________________________________________
```

In [16]:

```
### Train the residual compressor
earlystopper = EarlyStopping(patience=2, verbose=1)
checkpointer = ModelCheckpoint('VSB_r_autoassoc', verbose=1, save_best_on
ly=True)
results = r_model.fit(X, X, validation_split=0.1, batch_size=50, epochs=3
00,
                    callbacks=[earlystopper, checkpointer])
```

```
Train on 1800 samples, validate on 201 samples
Epoch 1/300
1750/1800 [============================>.] - ETA: 0s - loss: 3.4836
Epoch 00001: val_loss improved from inf to 3.16149, saving model to VS
B_r_autoassoc
1800/1800 [==============================] - 19s 10ms/step - loss: 3.4
892 - val_loss: 3.1615
Epoch 2/300
1750/1800 [============================>.] - ETA: 0s - loss: 3.3301
Epoch 00002: val_loss improved from 3.16149 to 3.09321, saving model t
o VSB_r_autoassoc
1800/1800 [==============================] - 17s 9ms/step - loss: 3.31
99 - val_loss: 3.0932
Epoch 3/300
1750/1800 [============================>.] - ETA: 0s - loss: 3.2874
Epoch 00003: val_loss did not improve from 3.09321
1800/1800 [==============================] - 16s 9ms/step - loss: 3.28
12 - val_loss: 3.1068
Epoch 4/300
1750/1800 [============================>.] - ETA: 0s - loss: 3.2494
Epoch 00004: val_loss did not improve from 3.09321
1800/1800 [==============================] - 15s 9ms/step - loss: 3.27
02 - val_loss: 3.0963
Epoch 00004: early stopping
```

In [17]:

```
r_model = load_model('VSB_r_autoassoc')
```

In [18]:

```
# Load just the compression portion of the autoencoder
c_r_model = Sequential()
c_r_model.add(Dense(compression_2 * 2, activation='relu', input_shape=(80
0000,), name='layer1'))
c_r_model.add(Dense(compression_2 * 4, activation='relu', name='layer2'))
c_r_model.add(Dense(compression_2, activation='relu', name='compressed'))

c_r_model.load_weights('VSB_r_autoassoc', by_name=True)
```

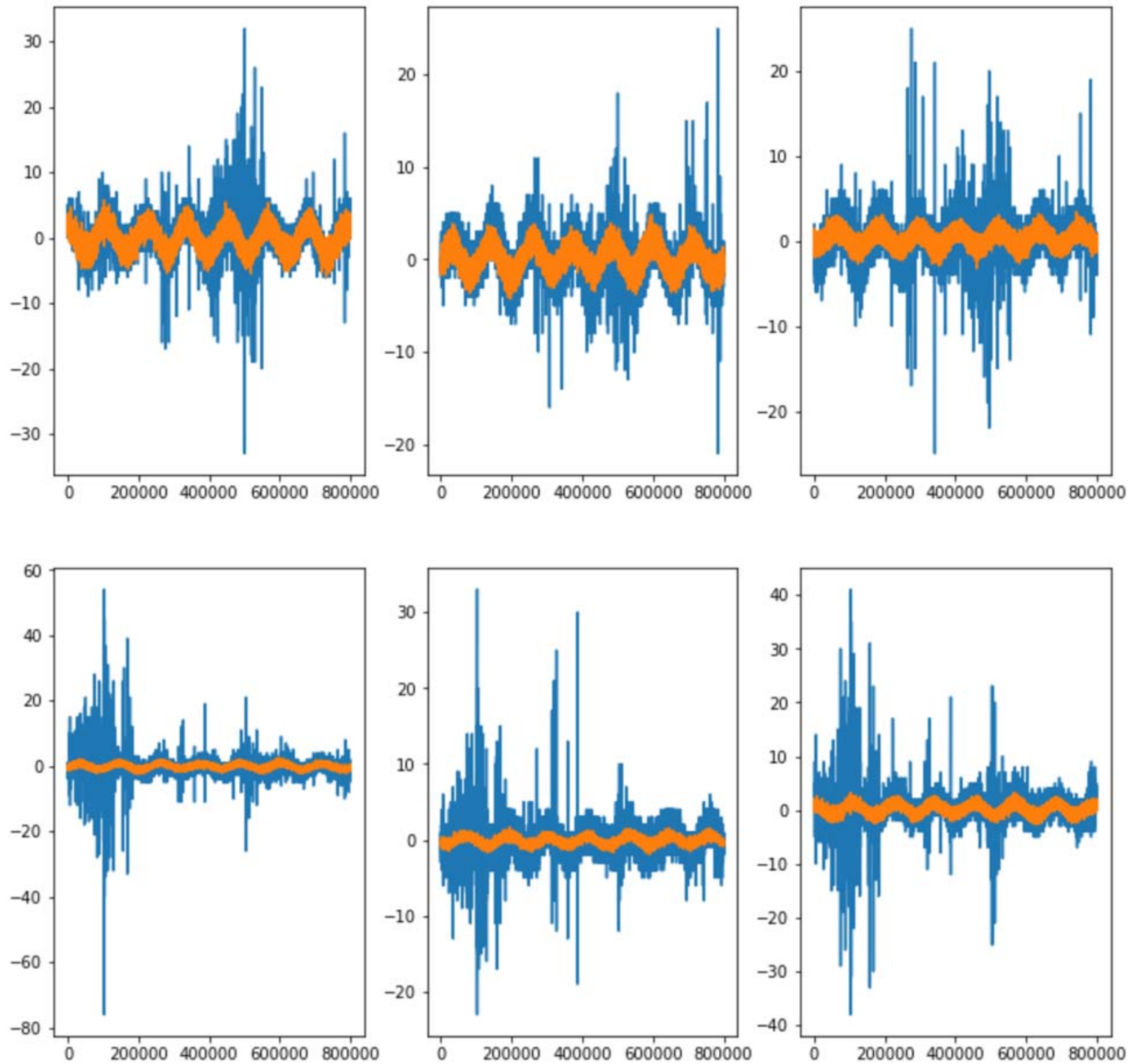
In [19]:

```
# Use the complete autoencoder to create lossy compressed versions of the r
esiduals
X_decompressed = r_model.predict(X[0:6])
X_decompressed.shape
```

Out[19]:

```
(6, 800000)
```

In [20]:
```
# Display these decompressed residuals, showing difference from the actual
 residuals
# Here we should see any remaining common elements represented by the resid
ual approximation
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2,3, figsize=(12,1
2))
ax1.plot(X[0]) ;
ax2.plot(X[1]) ;
ax3.plot(X[2]) ;
ax4.plot(X[3]) ;
ax5.plot(X[4]) ;
ax6.plot(X[5]) ;
ax1.plot(X_decompressed[0]) ;
ax2.plot(X_decompressed[1]) ;
ax3.plot(X_decompressed[2]) ;
ax4.plot(X_decompressed[3]) ;
ax5.plot(X_decompressed[4]) ;
ax6.plot(X_decompressed[5]) ;
```



In [21]:

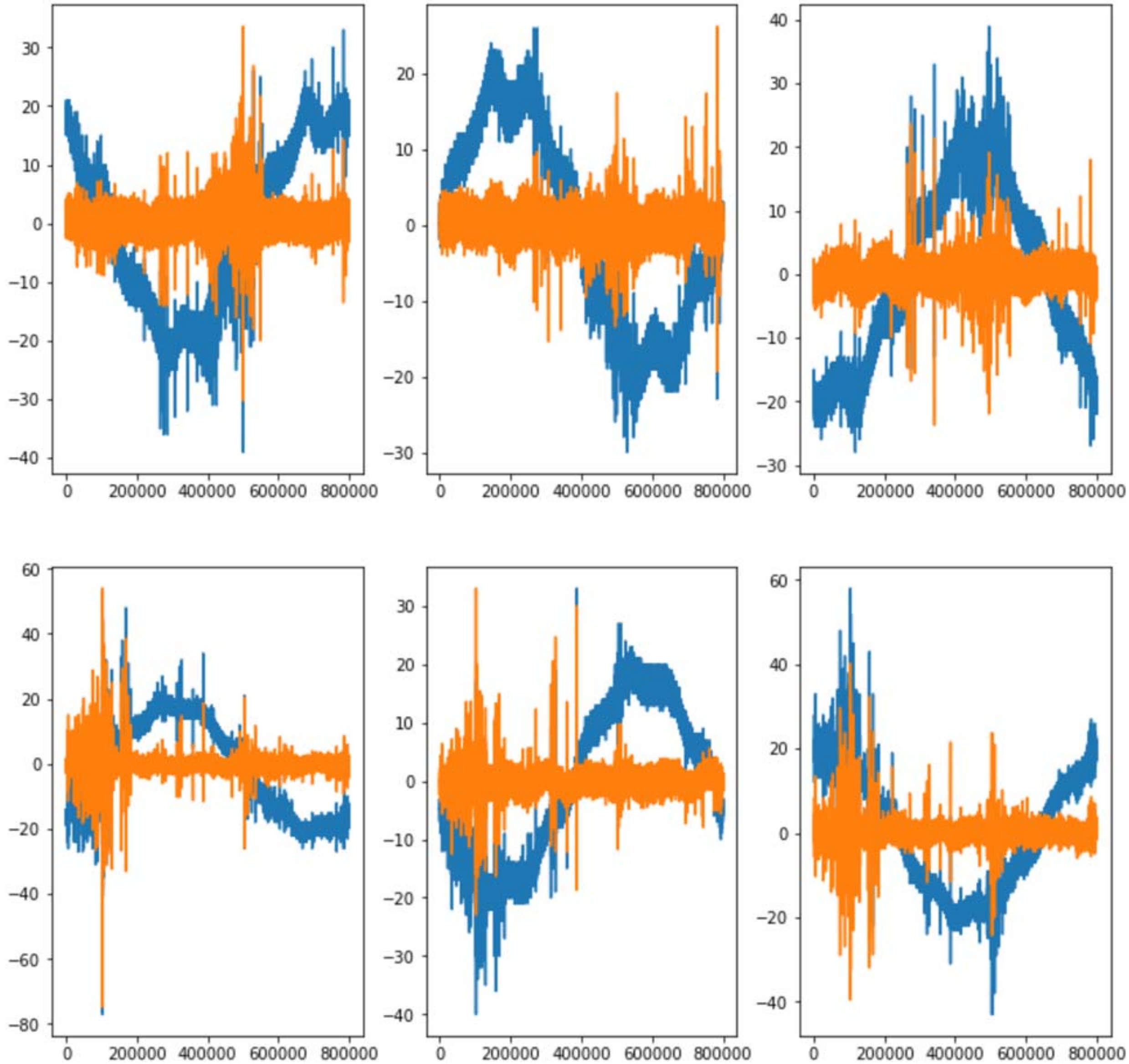
```
# What about the residual of the residual? This is also called the 2nd deco
mposition, in wavelet transformation lingo
# This is (hopefully) the distilled information that makes every signal dif
ferent from all the others
# becuase it was not captured by the "approximations" learned by the previo
us two autoencoders
X_residual = X[0:6] - X_decompressed[0:6]
X_residual.shape
```

Out[21]:

```
(6, 800000)
```

In [22]:

```
# This 2nd residual is now displayed, superimposed on the original signal
# Visually / qualitatively, it should look like the "important" informatio
n, if our process is working correctly
fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2,3, figsize=(12,1
2))
ax1.plot(subset_train['0']) ;
ax2.plot(subset_train['1']) ;
ax3.plot(subset_train['2']) ;
ax4.plot(subset_train['3']) ;
ax5.plot(subset_train['4']) ;
ax6.plot(subset_train['5']) ;
ax1.plot(X_residual[0]) ;
ax2.plot(X_residual[1]) ;
ax3.plot(X_residual[2]) ;
ax4.plot(X_residual[3]) ;
ax5.plot(X_residual[4]) ;
ax6.plot(X_residual[5]) ;
```


30
20
10
0
−10
−20
−30
−40
0 200000 400000 600000 800000
20
10
0
−10
−20
−30
0 200000 400000 600000 800000
40
30
20
10
0
−10
−20
−30
0 200000 400000 600000 800000
60
40
20
0
−20
−40
−60
−80
0 200000 400000 600000 800000
30
20
10
0
−10
−20
−30
−40
0 200000 400000 600000 800000
60
40
20
0
−20
−40
0 200000 400000 600000 800000

In [23]:

```
# Some global parameters
# number of non-overlapping time slices within the signal that will be used
to extract feature information
windows = 80
# number of features per time slice
win_feat = 7
# total number of features extracted from the signal
number_of_features = int(compression_1 + compression_2 + (windows * win_f
eat))

# Now batch convert original signals into training feature vectors
# the feature vectors consist of three parts:
# * compressed signal (a handful of floating point numbers)
# * compressed first residual (some more floating point numbers)
# * N*F features of the N windows of 800,000 / N samples of the 2nd residua
l
#     put another way, this last part is F floating point numbers containin
g statistical info about the 2nd residual
#     after we chop up that 2nd residual into N different non-overlapping t
ime windows
def extract_features(begin_col, loc_num_to_use, filename) :
    batch = 300
    num_batches = int((loc_num_to_use) / batch)
    remainder = int((loc_num_to_use) % batch)
    win_size = int(800000 / windows)
    ### Create a pandas data frame
    loc_X = np.zeros((int(loc_num_to_use), number_of_features))
    if num_batches > 0:
        for ix in range (num_batches) :
            # load a batch of signals
            x1 = pq.read_pandas(filename, columns=[str(ix * batch + j + b
egin_col) for j in range(batch)]).to_pandas().values.transpose()
            # compress them into 20 data values
            c_x1 = c_model.predict(x1)
            # recreate them from the compressed data
            x1_r = model.predict(x1)
            # residual is the original signal minus the recreated one
            res_x1 = x1 - x1_r
            # compress the residual
            c_res_x1 = c_r_model.predict(res_x1)
            # recreate the residuals from the compressed data
```

```
            res_x1_r = r_model.predict(res_x1)
            # second residual is the residual minus the recreated residual
            res_2_x1 = res_x1 - res_x1_r
            for j in range (0,batch) :
                i = ix * batch + j
                loc_X[i,0:compression_1] = c_x1[j]
                loc_X[i,compression_1:compression_1 + compression_2] = c_
res_x1[j]
                for win in range (windows) :
                    # start and end of window in signal data
                    win_start = win * win_size
                    win_end = win_start + win_size
                    # start of windows features in feature array (loc_X)
                    win_fs = win * win_feat
                    ### V04 ----------
                    ### Using example code from VSB Power LSTM attention h
ttps://www.kaggle.com/braquino/vsb-power-lstm-attention by Bruno Aquino
                    # Mean
                    loc_X[i,compression_1 + compression_2 + win_fs + 0] =
mean = res_2_x1[j,win_start:win_end].mean()
                    # Standard Deviation = sqrt(variance)
                    loc_X[i,compression_1 + compression_2 + win_fs + 1] =
std  = res_2_x1[j,win_start:win_end].std()
                    # top of standard deviation range
                    loc_X[i,compression_1 + compression_2 + win_fs + 2] =
mean + std
                    # bottom of standard deviation range
                    loc_X[i,compression_1 + compression_2 + win_fs + 3] =
mean - std
                    # calculate a handful of percentiles
                    pct_calc = np.percentile(res_2_x1[j,win_start:win_end
], [0, 1, 25, 50, 75, 99, 100])
                    # max range of percentiles
                    loc_X[i,compression_1 + compression_2 + win_fs + 4] =
pct_calc[-1] - pct_calc[0]
                    # coefficient of variation (standard deviation divided
by mean)
                    loc_X[i,compression_1 + compression_2 + win_fs + 5] =
std / mean
                    # A measure of asymmetry (75th percentile subtracted f
rom mean)
                    loc_X[i,compression_1 + compression_2 + win_fs + 6] =
```

```
mean - pct_calc[4]
                    # the seven percentile values calcuated earlier
                    ### end of example code for suggested features to extr
act
    ix = num_batches
    # load a batch of signals
    x1 = pq.read_pandas(filename, columns=[str(ix * batch + j + begin_col
) for j in range(batch)]).to_pandas().values.transpose()
    # compress them into 20 data values
    c_x1 = c_model.predict(x1)
    # recreate them from the compressed data
    x1_r = model.predict(x1)
    # residual is the original signal minus the recreated one
    res_x1 = x1 - x1_r
    # compress the residual
    c_res_x1 = c_r_model.predict(res_x1)
    # recreate the residuals from the compressed data
    res_x1_r = r_model.predict(res_x1)
    # second residual is the resodual minus the recreated residual
    res_2_x1 = res_x1 - res_x1_r

    for j in range (0,remainder) :
        i = ix * batch + j
        loc_X[i,0:compression_1] = c_x1[j]
        loc_X[i,compression_1:compression_1 + compression_2] = c_res_x1[j
]
        for win in range (windows) :
            win_start = win * win_size
            win_end = win_start + win_size
            # start of windows features in feature array (loc_X)
            win_fs = win * win_feat
            ### V04 ----------
            ### Using example code from VSB Power LSTM attention https://ww
w.kaggle.com/braquino/vsb-power-lstm-attention by Bruno Aquino
            # Mean
            loc_X[i,compression_1 + compression_2 + win_fs + 0] = mean =
res_2_x1[j,win_start:win_end].mean()
            # Standard Deviation = sqrt(variance)
            loc_X[i,compression_1 + compression_2 + win_fs + 1] = std  =
res_2_x1[j,win_start:win_end].std()
            # top of standard deviation range
            loc_X[i,compression_1 + compression_2 + win_fs + 2] = mean +
```

```
std
            # bottom of standard deviation range
            loc_X[i,compression_1 + compression_2 + win_fs + 3] = mean -
std
            # calculate a handful of percentiles
            pct_calc = np.percentile(res_2_x1[j,win_start:win_end], [0, 1
, 25, 50, 75, 99, 100])
            # max range of percentiles
            loc_X[i,compression_1 + compression_2 + win_fs + 4] = pct_cal
c[-1] - pct_calc[0]
            # coefficient of variation (standard deviation divided by mean)
            loc_X[i,compression_1 + compression_2 + win_fs + 5] = std / m
ean
            # A measure of asymmetry (75th percentile subtracted from mean)
            loc_X[i,compression_1 + compression_2 + win_fs + 6] = mean -
pct_calc[4]
            # the seven percentile values calcuated earlier
            ### end of example code for suggested features to extract

    return loc_X
```

In [24]:

```
%%time
# Now we are recreating what we did above for the first six samples, but
 for the entire training set
# encoding the signals, encoding the residuals, and gathering statistical
info about each 2nd residual
num_to_use = 8712
# num_to_use = 30

from sklearn import preprocessing

# extract features
X_unscaled = extract_features(0, num_to_use, '../input/train.parquet')

scaler = preprocessing.StandardScaler().fit(X_unscaled)
X = scaler.transform(X_unscaled)

# correct classifications from training set
y = np.zeros((num_to_use))
for i in range(0, int(num_to_use)):
    y[i] = meta_train.target[i]

print(y.shape)
```

```
(8712,)
CPU times: user 8min 22s, sys: 1min 7s, total: 9min 30s
Wall time: 9min 27s
```

In [25]:

```
# This is not really necessary, but it gives me a good hint if the problem
 is being solved correctly
# How often do power line faults occur on all three phases simultaneously,
 versus on fewer than all 3?
triples = 0
doubles = 0
singles = 0
for i in range(0,int(num_to_use),3) :
    if (meta_train.target[i] and meta_train.target[i+1] and meta_train.ta
rget[i+2] ):
#        print('triple',meta_train.signal_id[i], meta_train.phase[i] )
        triples = triples + 1
    elif (meta_train.target[i] + meta_train.target[i+1] + meta_train.targ
et[i+2] == 2):
#        print('double',meta_train.signal_id[i], meta_train.phase[i])
        doubles = doubles + 1
    elif (meta_train.target[i] + meta_train.target[i+1] + meta_train.targ
et[i+2] == 1):
#        print('single',meta_train.signal_id[i], meta_train.phase[i])
        singles = singles + 1

print('triples', triples, 'doubles', doubles, 'singles', singles)
print('sanity check: ', 'total faults', meta_train.target[0:int(num_to_us
e)].sum(), ' sum of above ', 3 * triples + 2 * doubles + singles)
# plt.plot(meta_train.target)
```

```
triples 156 doubles 19 singles 19
sanity check:  total faults 525  sum of above  525
```

In [26]:

```
"""%%time



from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

first_model = RandomForestRegressor(n_estimators=30, min_samples_leaf=30,
                                    random_state=1).fit(X, y)
# Environment Set-Up for feedback system.
from learntools.core import binder
binder.bind(globals())
from learntools.ml_insights.ex2 import *
print("Training Complete")
predicted_y = first_model.predict(X)
"""
```

Out[26]:

```
'%%time\n\n\nfrom sklearn.ensemble import RandomForestRegressor\nfrom
sklearn.linear_model import LinearRegression\nfrom sklearn.model_selec
tion import train_test_split\n\nfirst_model = RandomForestRegressor(n_
estimators=30, min_samples_leaf=30, \n
random_state=1).fit(X, y)\n# Environment Set-Up for feedback system.\n
from learntools.core import binder\nbinder.bind(globals())\nfrom learn
tools.ml_insights.ex2 import *\nprint("Training Complete")\npredicted_
y = first_model.predict(X)\n'
```

In [27]:

```
# Now we train a multi-layer ann to learn the correct classifications
class_model = Sequential()
class_model.add(Dense(number_of_features, activation='relu', input_shape=
(number_of_features,)))
class_model.add(Dense(number_of_features * 2, activation='relu'))
class_model.add(Dense(number_of_features * 4, activation='relu'))
class_model.add(Dense(number_of_features * 2, activation='relu'))
class_model.add(Dense(number_of_features, activation='relu'))
class_model.add(Dense(1, activation='sigmoid', name='output'))

class_model.compile(loss='binary_crossentropy', optimizer='adam')
# Diplay the model summary
print("model summary")
class_model.summary()
```

```
model summary
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 585)               342810
_________________________________________________________________
dense_2 (Dense)              (None, 1170)              685620
_________________________________________________________________
dense_3 (Dense)              (None, 2340)              2740140
_________________________________________________________________
dense_4 (Dense)              (None, 1170)              2738970
_________________________________________________________________
dense_5 (Dense)              (None, 585)               685035
_________________________________________________________________
output (Dense)               (None, 1)                 586
=================================================================
Total params: 7,193,161
Trainable params: 7,193,161
Non-trainable params: 0
_________________________________________________________________
```

In [28]:

```
### Train the residual compressor
earlystopper = EarlyStopping(patience=5, verbose=1)
checkpointer = ModelCheckpoint('VSB_classifier', verbose=1, save_best_onl
y=True)
results = class_model.fit(X, y, validation_split=0.2, batch_size=50, epoc
hs=300,
                    callbacks=[earlystopper, checkpointer])
```

```
Train on 6969 samples, validate on 1743 samples
Epoch 1/300
6800/6969 [============================>.] - ETA: 0s - loss: 0.2369
Epoch 00001: val_loss improved from inf to 0.14968, saving model to VS
B_classifier
6969/6969 [==============================] - 4s 534us/step - loss: 0.2
340 - val_loss: 0.1497
Epoch 2/300
6900/6969 [============================>.] - ETA: 0s - loss: 0.1559
Epoch 00002: val_loss did not improve from 0.14968
6969/6969 [==============================] - 2s 285us/step - loss: 0.1
551 - val_loss: 0.1535
Epoch 3/300
6850/6969 [============================>.] - ETA: 0s - loss: 0.1251
Epoch 00003: val_loss did not improve from 0.14968
6969/6969 [==============================] - 2s 279us/step - loss: 0.1
244 - val_loss: 0.1911
Epoch 4/300
6850/6969 [============================>.] - ETA: 0s - loss: 0.1172
Epoch 00004: val_loss did not improve from 0.14968
6969/6969 [==============================] - 2s 279us/step - loss: 0.1
165 - val_loss: 0.1669
Epoch 5/300
6800/6969 [============================>.] - ETA: 0s - loss: 0.1136
Epoch 00005: val_loss improved from 0.14968 to 0.14360, saving model t
o VSB_classifier
6969/6969 [==============================] - 2s 301us/step - loss: 0.1
138 - val_loss: 0.1436
Epoch 6/300
6850/6969 [============================>.] - ETA: 0s - loss: 0.1019
Epoch 00006: val_loss did not improve from 0.14360
6969/6969 [==============================] - 2s 282us/step - loss: 0.1
010 - val_loss: 0.2064
Epoch 7/300
6850/6969 [============================>.] - ETA: 0s - loss: 0.1038
Epoch 00007: val_loss did not improve from 0.14360
6969/6969 [==============================] - 2s 278us/step - loss: 0.1
033 - val_loss: 0.1904
Epoch 8/300
6850/6969 [============================>.] - ETA: 0s - loss: 0.1361
Epoch 00008: val_loss did not improve from 0.14360
```

```
6969/6969 [==============================] - 2s 289us/step - loss: 0.1
364 - val_loss: 0.1668
Epoch 9/300
6850/6969 [============================>.] - ETA: 0s - loss: 0.0809
Epoch 00009: val_loss did not improve from 0.14360
6969/6969 [==============================] - 2s 279us/step - loss: 0.0
810 - val_loss: 0.2148
Epoch 10/300
6850/6969 [============================>.] - ETA: 0s - loss: 0.0660
Epoch 00010: val_loss did not improve from 0.14360
6969/6969 [==============================] - 2s 281us/step - loss: 0.0
653 - val_loss: 0.2230
Epoch 00010: early stopping
```

In [29]:

```
class_model = load_model('VSB_classifier')
```

In [30]:

```
predicted_y = class_model.predict(X)
predicted_y.shape
```

Out[30]:

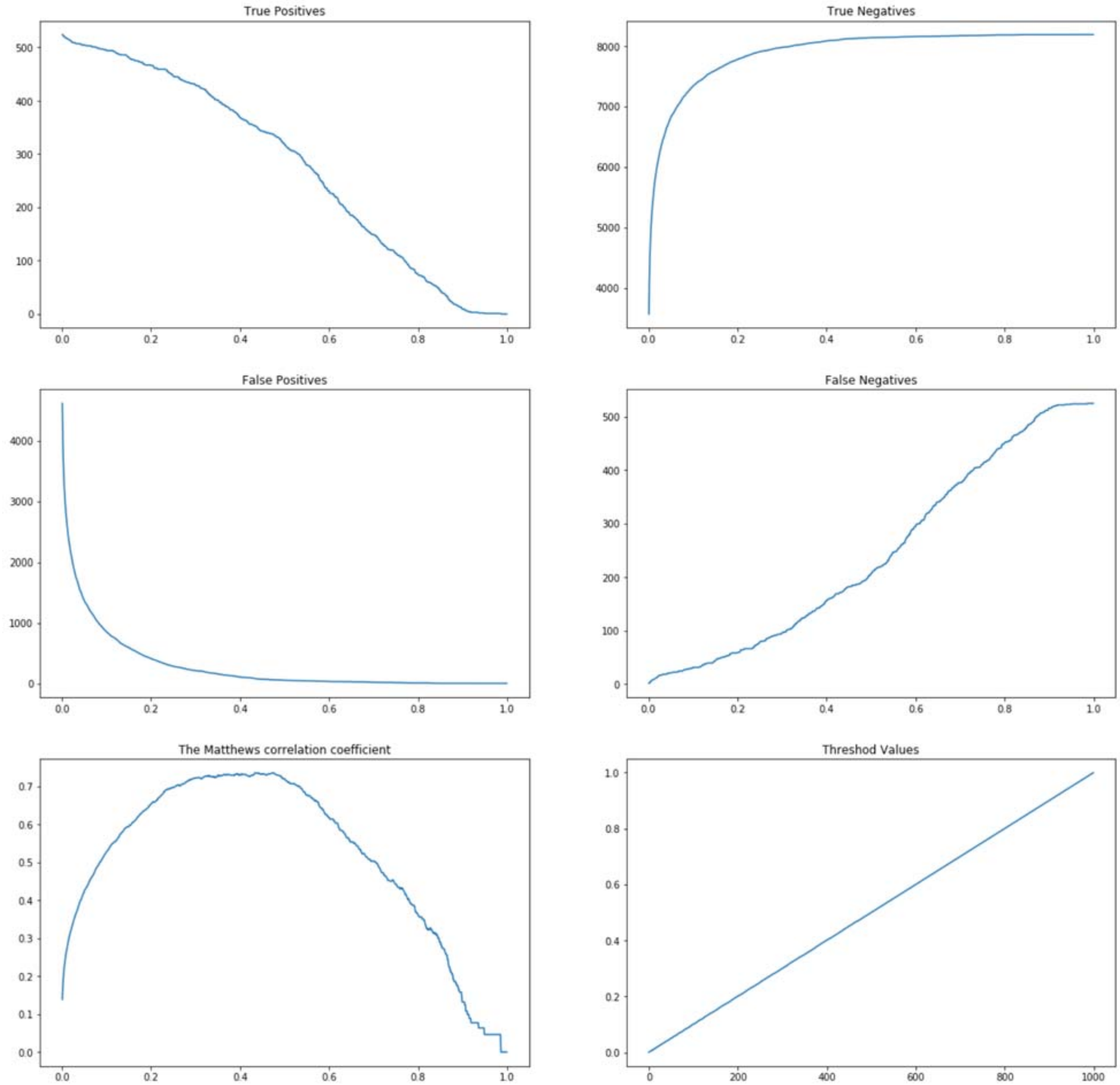
```
(8712, 1)
```

In [31]:

```
### See if the best_threshold is indeed the best by using the Matthews Corr
elation Coefficient
### The Matthews correlation coefficient
### MCC=( TP * TN - FP * FN ) / sqrt  ( (TP+FP) * (TP+FN) * (TN+FP) * (TN+F
N) )

import math

best_thresh = 0
num_threshes = 1000           # number of different thresholds to try
min_thresh = (1 / num_threshes) / 2 # the minimum threshold to try
best_MCC = 0
TPs = np.zeros(num_threshes)
TNs = np.zeros(num_threshes)
FPs = np.zeros(num_threshes)
FNs = np.zeros(num_threshes)
MCCs = np.zeros(num_threshes)
threshes = np.zeros(num_threshes)
# print ("threshold, Data count, TP, TN, FP, FN, TP + TN + FP + FN, MCC")
for this_thresh in range (num_threshes):
    thresh = round(min_thresh + (this_thresh / num_threshes), 3)
    TP = 0
    TN = 0
    FP = 0
    FN = 0
    for i in range(0, int(num_to_use)):
        if (y[i]) :
            if (predicted_y[i] > thresh) :
                TP = TP + 1
            else :
                FN = FN + 1
        else:
            if (predicted_y[i] > thresh) :
                FP = FP + 1
            else :
                TN = TN + 1
        if (math.sqrt ( (TP + FP) * (TP + FN) * (TN + FP) * (TN + FP))):
            MCC = ((TP * TN) - (FP - FN)) / math.sqrt ( (TP + FP) * (TP +
FN) * (TN + FP) * (TN + FP) )
        else :
            MCC = 0
```

```
    TPs[this_thresh] = TP
    TNs[this_thresh] = TN
    FPs[this_thresh] = FP
    FNs[this_thresh] = FN
    MCCs[this_thresh] = MCC
    threshes[this_thresh] = thresh
#     print(thresh, num_to_use * 3, TP, TN, FP, FN, (TP+TN+FP+FN), MCC)
    if (MCC >= best_MCC):
        best_MCC = MCC
        best_thresh = thresh

fig, ((ax1, ax2), (ax3, ax4), (ax5,ax6)) = plt.subplots(3,2, figsize=(20,
20))
ax1.plot(threshes, TPs)
ax1.set_title("True Positives");
ax2.plot(threshes, TNs) ;
ax2.set_title("True Negatives");
ax3.plot(threshes, FPs) ;
ax3.set_title("False Positives");
ax4.plot(threshes, FNs) ;
ax4.set_title("False Negatives");
ax5.plot(threshes, MCCs) ;
ax5.set_title("The Matthews correlation coefficient");
ax6.plot(threshes) ;
ax6.set_title("Threshod Values");

print(' ')
print('best threshold, ', best_thresh, ' yielding best MCC, ', best_MCC)
```

```
best threshold,  0.434  yielding best MCC,  0.7359889620528433
```



In [32]:

```
y = predicted_y
# See how many doubles and triples we got on the test data
# How often did we predict power line faults occur on all three phases simu
ltaneously, versus on fewer than all 3?
triples = 0
doubles = 0
singles = 0
for i in range(0,int(num_to_use),3) :
    y[i] = int(y[i] > best_thresh)
    y[i+1] = int(y[i+1] > best_thresh)
    y[i+2] = int(y[i+2] > best_thresh)
    num_phases_faulty = y[i] + y[i+1] + y[i+2]
    if (num_phases_faulty == 3):
#        print('triple',meta_train.signal_id[i], meta_train.phase[i] )
        triples = triples + 1
    elif (num_phases_faulty == 2):
#        print('double',meta_train.signal_id[i], meta_train.phase[i])
        doubles = doubles + 1
    elif (num_phases_faulty == 1):
#        print('single',meta_train.signal_id[i], meta_train.phase[i])
        singles = singles + 1

print('triples', triples, 'doubles', doubles, 'singles', singles)
print('sanity check: ', 'total faults', y.sum(), ' sum of above ', 3 * tr
iples + 2 * doubles + singles)
```

```
triples 84 doubles 53 singles 72
sanity check:  total faults 430.0  sum of above  430
```

In [33]:

```
# quick and dirty RAM cleanup
X_unscaled = 5
X = 5
y = 5
predicted_y = 5
TPs = 5
TNs = 5
FPs = 5
FNs = 5
MCCs = 5
threshes = 5
results = 5
X_residual = 5
X_decompressed = 5
```

In [34]:

```
%%time
# Now load the test data
meta_test = pd.read_csv('../input/metadata_test.csv')

# Calculate Test Value Parameters

# number of test examples to use
num_to_use = 20337
# use a smaller subset for testing if you don't want to wait (must be a m
ultiple of 3)
# num_to_use = 30

X = 5
X_unscaled = 5

X_unscaled = extract_features(8712, num_to_use, '../input/test.parquet')

X = scaler.transform(X_unscaled)
```

```
CPU times: user 21min 1s, sys: 2min 36s, total: 23min 38s
Wall time: 23min 32s
```

In [35]:

```
y = class_model.predict(X)

# How often did we predict power line faults occur on all three phases simu
ltaneously, versus on fewer than all 3?
triples = 0
doubles = 0
singles = 0
for i in range(0,int(num_to_use),3) :
    y[i] = int(y[i] > best_thresh)
    y[i+1] = int(y[i+1] > best_thresh)
    y[i+2] = int(y[i+2] > best_thresh)
    num_phases_faulty = y[i] + y[i+1] + y[i+2]
    if (num_phases_faulty == 3):
#        print('triple',meta_train.signal_id[i], meta_train.phase[i] )
        triples = triples + 1
    elif (num_phases_faulty == 2):
#        print('double',meta_train.signal_id[i], meta_train.phase[i])
        doubles = doubles + 1
    elif (num_phases_faulty == 1):
#        print('single',meta_train.signal_id[i], meta_train.phase[i])
        singles = singles + 1

print('triples', triples, 'doubles', doubles, 'singles', singles)
print('sanity check: ', 'total faults', y.sum(), ' sum of above ', 3 * tr
iples + 2 * doubles + singles)
# plt.plot(meta_train.target)
```

```
triples 109 doubles 148 singles 236
sanity check:  total faults 859.0  sum of above  859
```

In [36]:

```
output = pd.DataFrame({"signal_id":meta_test.signal_id[0:int(num_to_use *
3)]})
output["target"] = pd.Series(y[:,0])
output['signal_id'] = output['signal_id'].astype(np.int64)
output['target'] = output['target'].astype(np.int64)
output.to_csv("submission.csv", index=False)
output
```