Machine Learning Engineer Nanodegree

Capstone Report (LANL Earthquake Prediction from Kaggle Competition)

Tobias Steidle

(tobias.steidle@softwaredev.de)

June 24th, 2019

Report

I. Definition

Projekt Overview

This project is about predicting when an earthquake will occur. Based on seismic data and using a machine learning method, it will be determined when the earthquake will occur and how much time will elapse before the earthquake occurs.

The project is designed as a Kaggle Challenge. This means that the training and test data will be made available on the Kaggle platform. And it is a competition to achieve the lowest "Mean Absolute Error" on the test data. If the problem is solved, the trained model can be scaled to real data. This gives researchers the potential to improve predictions of earthquake hazards and save lives and billions of dollars in infrastructure.

There are already publications dealing with the topic of predictions of laboratory earthquakes. A publication can be found e.g. here[1]. In this publication a Random Forest Algorithm achieves a high accuracy (R^2=0.91 training / R^2=0.89 test) to predict the "time to failure". This shows that the approach is practicable and the problem can be solved.
Other Publications. [2] [3] [4]

Problem Statement

The goal in this project is to achieve the lowest possible MAE (Mean Absolute Error) on the test data set. A MAE as small as possible in this case means that the predicted time period, until the earthquake occurs, is close to the real-time periods on which the test data is based.

The following steps are necessary to solve this problem:
1. download the data from Kaggle
2. visualizing the data for a better understanding
3. preprocessing of the data
4. training a suitable regressor
5. writing the submission file for the Kaggle Competition

The submission file is then transmitted to Kaggle to set the score.

Metrics

The used metric is the Mean Absolute Error. The Mean Absolute Error measures the average magnitude of the errors in a set of predictions, without considering their direction. It's the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight.

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

n = Number of predicted values
y_j = real value
$\hat{y}_j$ = predicted value

For each record oft he test dataset the prediction error is calculated. Convert each error to a positive figure by taking the absolute value for each error. Finally, calculate the mean value for all recorded absolute errors.

Since this is a Kaggle Competition, the metric is given in this case.

This metric makes sense because it punishes high errors, but not as high as the Mean Square Error (MSE). This makes the metric less susceptible to outliers. The MAE is a linear score, i.e. all differences are weighted equally on average.

II. Analysis

Data Exploration

The data sets are freely available for download on the Kaggle competition website.

The data consists of a train.csv file containing a single, continuous training segment of experimental data. And test data in the test.zip file.

Input Data fields

- acoustic_data - the seismic signal [int16]
- time_to_failure - the time (in seconds) until the next laboratory earthquake [float64]
- Total rows in this train dataset: 629145480.

Test data set

- test.zip containing many small segments of test data.
- test dataset that is split into 2624 files , each one containing 150,000 acoustic_data points

	acoustic_data	time_to_failure
0	12.0	1.4691
1	6.0	1.4691
2	8.0	1.4691
3	5.0	1.4691
4	8.0	1.4691

Auszug aus den Trainingsdaten

What is not apparent at a glance on the data extract is that the "time_to_failure" in these 5 lines looks identical, but it is not. There are several digits after the comma in the real dataset. The "time_to_failure" is descending until the earthquake occurs. After the earthquake, the "time_to_failure" is increased again. This means that the training data contains several segments (start of measurement until the earthquake occurs).

The signal amplitude distribution (e.g., its variance and higher-order moments) are highly effective at forecasting failure. The variance, which characterizes overall signal amplitude fluctuation, is the strongest single feature early in time. As the system nears failure, other outlier statistics such as the kurtosis and thresholds become predictive as well.

Exploratory Visualization

The data consists of the "acoustic_data" points and the "time_to_failure".

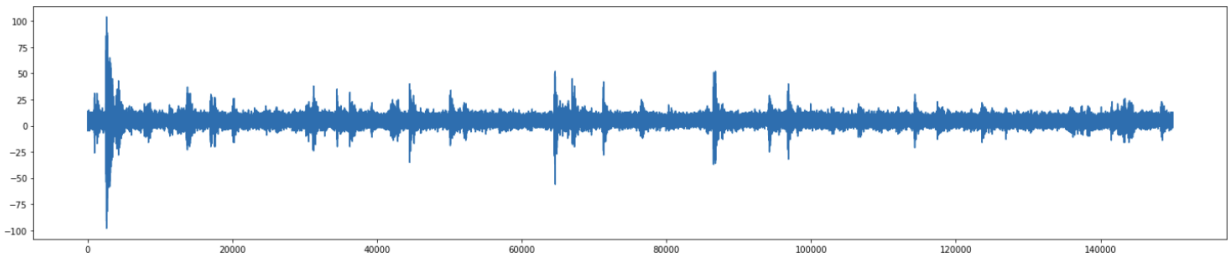

Fig. 1

Fig. 1 shows the acoustic data. In this plot, you can see the height of the amplitude on the y-axis or in other words the strength of the signal. The higher the amplitude, the stronger the signal. The x-axis shows the time course.

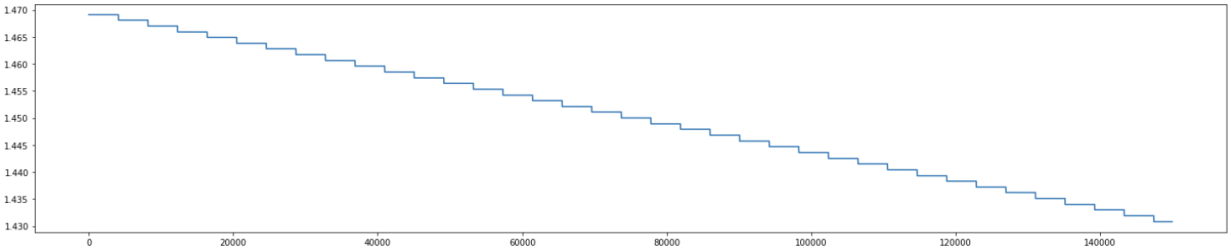

Fig. 2

Fig. 2 shows the value "time_to_failure". The time until the event is shown on the y-axis. On the x-axis, as in Fig. 1, the time course. The further the time progresses, the shorter the time until the event occurs.

The plots show how the acoustic signal in Fig. 1 behaves until the event occurs.

Alorithms and Techniques

To solve this problem I plan to use a Recurrent Neural Network (RNN). An RNN is especially well suited to perform e.g. a speech processing by machine. The available data is an acoustic signal. So it is obvious that methods which are used e.g. for the processing of natural speech can also help to solve this problem.

Depending on the neural network there are various parameters that can be adapted or optimized:

- Batch Size
- Epochs
- Train/test split of data
- Number and type of individual layers (LSTM, Dropout, Dense, ...)
- Optimizer (incl. associated parameters)
- learning rate

Recurrent neural networks use sequences as input data. In an RNN, the information goes through a loop. When RNN makes a decision, it takes into account the current input and what it has learned from the input it has received. Recurrent Neural Networks, therefore, has two inputs, the present and the past. This is important because the order of the data contains important information about it.

Imagine that you have a normal neural feedforward network and specify the word "matrix" as input and it processes the word character by character. By the time it reaches the letter "r," it has already forgotten "M," "a," and "t," making it almost impossible for this type of neural network to predict which letter will come next.

The situation is similar with this problem. Not a single value of "accustic_data" is relevant, but the whole sequence.

How the data is prepared is dealt with separately in the section Data Preprocessing.

Benchmark

The benchmark for the final RNN is a Mean Absolute Error achieved by KNeighborsRegressor. The KNeighborsRegressor is trained like the final model with the preprocessed data. The achieved Mean Absolute Error can be compared as a reference value with the MAE of the final model. This allows you to determine whether the final model shows an improvement over a non-optimized model.

The mean absolute error for the benchmark is approx. **2.16**

III. Methodology

Data Preprocessing

To process the acoustic signals, the signal is not used directly but an MFCC (Mel Frequency Cepstral Coefficients) is applied. The MFCC leads to a compact representation of the frequency spectrum. For my final version, I have chosen a number of 10 coefficients. The sample rate was set to 44100. Here is a visualization of the MFCC.

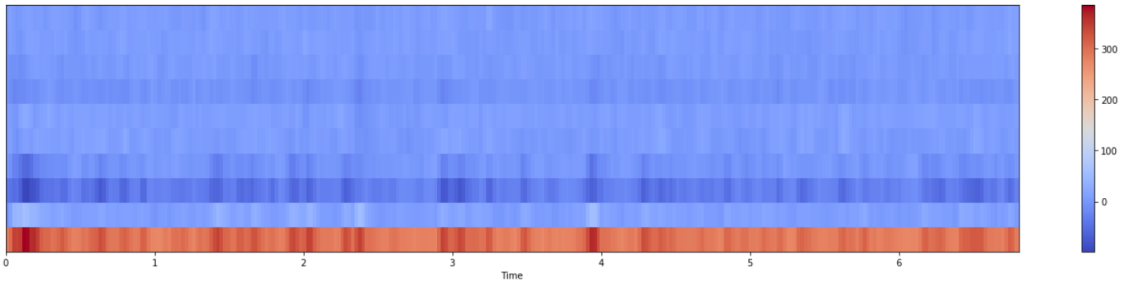

Fig. 3

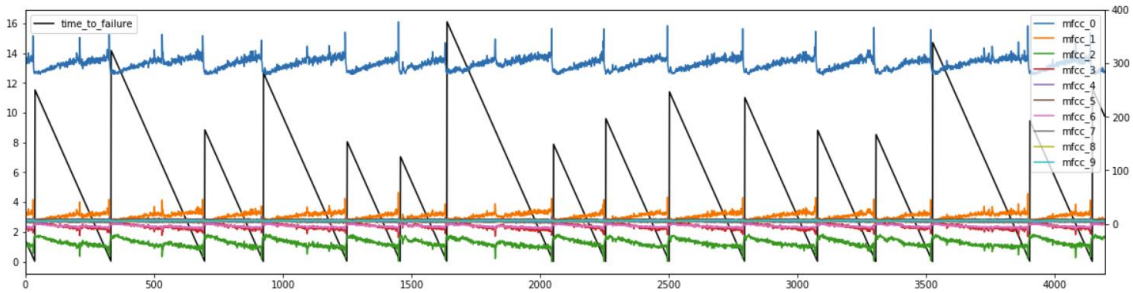

Fig. 4

Fig. 3 shows the visualization of the frequency spectrum. Fig. 4 shows the individual MFCC features in connection with "time_to_failure".

As can be seen on this plot, the curves of individual features are relatively identical. For example, there is a great similarity between "mfcc_2" and "mfcc_3". Therefore, I decided to reduce the MFCC features additionally via a Principal Component Analysis (PCA) from 10 to 6 features. This leaves only the most relevant features and the training time is reduced.

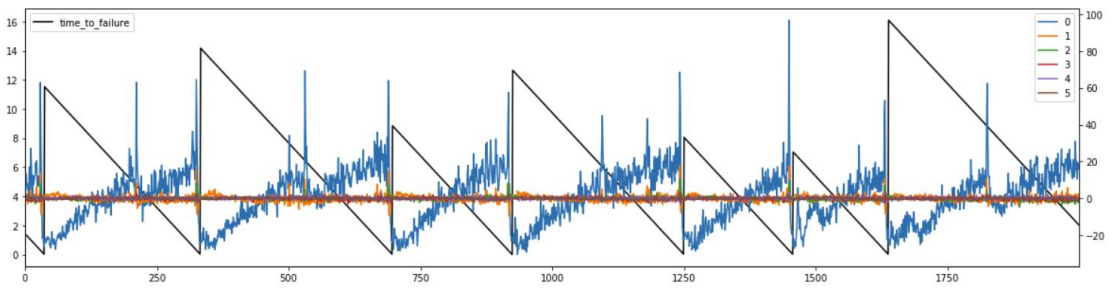

Fig. 5

Fig. 5 shows the visualization with the remaining 6 features used for the training.

Implementation

For the implementation 2 approaches were tried. The first step was to solve the problem with Scikit and an XGBRegressor. The final model was an RNN.

After pre-processing the data via MFCC and PCA, the data was split into training and test data:

```
from sklearn.model_selection import train_test_split

X = preprocessed_data_frame.drop(['time_to_failure'], axis=1).values
y = preprocessed_data_frame['time_to_failure'].values

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=seed)
```

After splitting the data, the implementation of the XGBRegressor begins.

XGBRegressor

The implementation of the XGBRegressor is as follows:

```
from sklearn import pipeline, metrics
from xgboost import XGBRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import RandomizedSearchCV

est = pipeline.Pipeline([
        ('xgb', XGBRegressor(silent=True)),
])

params = {
    'xgb__learning_rate': [0.003, 0.005, 0.01, 0.05, 0.1],
    'xgb__min_child_weight': [5, 6, 7, 8, 9, 10, 11, 12],
    'xgb__subsample': [0.5, 0.7, 0.9],
    'xgb__colsample_bytree': [0.5, 0.7, 0.9],
    'xgb__max_depth': [1, 3, 5, 7, 9, 11],
    'xgb__n_estimators': [10, 50, 100, 150],
}

cv_sets = ShuffleSplit(n_splits=10, test_size=0.2, random_state=seed)

search = RandomizedSearchCV(estimator=est,
                            param_distributions=params, scoring='neg_mean_absolute_error', random_state=seed,
                            n_iter=100, cv=cv_sets, verbose=1, n_jobs=1, return_train_score=True)


search.fit(X_train, y_train)
print(search.best_estimator_)

print_scores(search.best_estimator_, X_test, y_test)
print_mean_absolute_error(search.best_estimator_)
```

A large part of the implementation here is the use of RandomizedSearchCV. RandomizedSearch was chosen here because it requires less computing power than GridSearchCV and the best hyperparameters can be determined more quickly. It would be possible to combine the methods and after a run of the RandomizedSearchCV with the most important parameters to perform a GridSearchCV afterward. However, it has been shown that RNN delivers better results right away, so the XGBRegressor for the final version was discarded. For comparison, however, this method was retained in the Jupyter notebook.

Recurrent Neural Network

The implementation of the RNN is as follows:

```
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, Activation, GaussianNoise
from keras.optimizers import Adam, Adadelta, SGD, RMSprop, Nadam
from keras import losses
from keras.callbacks import TensorBoard
import time

X_train_reshape = np.reshape(X_train, (X_train.shape[0], 1, X_train.shape[1]))
X_test_reshape = np.reshape(X_test, (X_test.shape[0], 1, X_test.shape[1]))

model = Sequential()
model.add(LSTM(48, return_sequences=True, input_shape=(None,6)))
model.add(LSTM(36, return_sequences=True))
model.add(GaussianNoise(0.2))
model.add(Dropout(0.4))
model.add(LSTM(32))
model.add(GaussianNoise(0.1))
model.add(Dropout(0.2))
model.add(Dense(1, activation="relu"))

optimizer = RMSprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)

model.compile(loss=losses.mean_absolute_error, optimizer=optimizer, metrics=['mae'])

now = time.strftime("%Y%m%d_%H%M%S")
tensorboard_callback = TensorBoard(log_dir='./logs/'+now, histogram_freq=0, write_graph=True, write_images=True)


model.fit(X_train_reshape, y_train,
              epochs=200,
              batch_size=128,
              validation_split=0.05,
              verbose=1,
              callbacks=[tensorboard_callback]
             )

print(model.summary())
score = model.evaluate(X_test_reshape, y_test, batch_size=128)
```

First of all, the training and test data had to be reshaped so that it could be processed by the LSTM. Then the model was built with different layers, the optimizer was defined and the model was compiled. Tensorboard was used to visualize the course of the training.

The difficulty in implementing the RNN and especially in using the LSTM layers was to understand how to prepare the sequence data for the LSTM. The input for each LSTM layer must be 3-dimensional.

A good tutorial to understand how to reshaped the data can be found here. [5]

Refinement

During the training of the model, it was shown that it is not necessary to train more than 200 episodes. For better understanding, here is the course of the Mean Absolute Error during training.

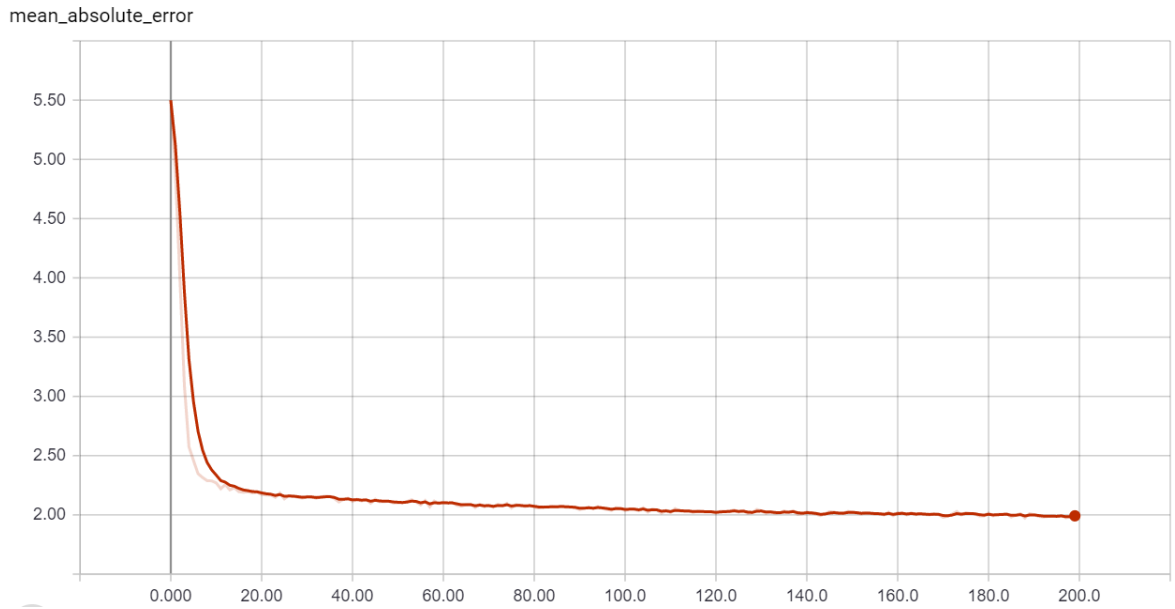

Fig. 6

Depending on the parameters used, the curve is different, but it has been shown that 200 episodes are sufficient at this point.

Due to the few episodes and the features reduced to 6 by PCA, the training time was relatively short. Thus it was possible to tune the architecture as well as the individual parameters manually.

Any versions showed that a larger model with many trainable parameters is counterproductive in this case. The MAE either stagnated at a relatively high value or the MAE increased again after a short valley. Overfitting also seemed to be a problem. It was therefore decided to use a model with relatively few trainable parameters.

In the first step, a relatively simple RNN was implemented with only one LSTM Layer (128) and one Dense Layer with an Adam Optimizer.

Over several optimization rounds the model became more complex but also the result was better until the final solution. Different optimizers (Adam, SGD, etc.) with different learning rates were tested. Noise and various dropouts have also experimented.

Fig. 7 show a comparison of the first version (red Line) with the final version (blue line) in Tensorboard.

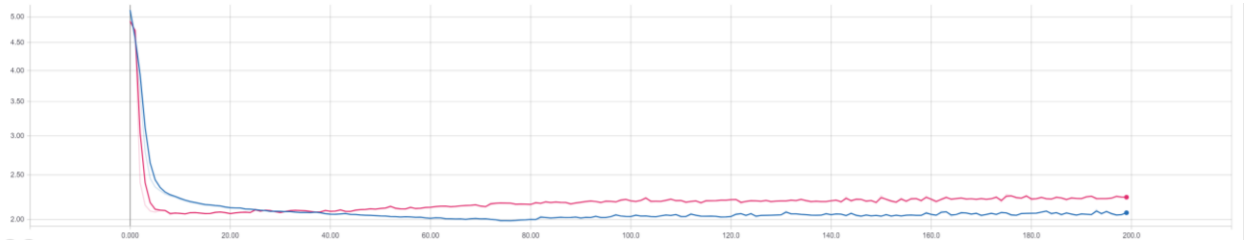

Fig. 7

IV. Results

Model Evaluation and Validation

The final model is structured as follows:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, None, 48)          10560
_________________________________________________________________
lstm_2 (LSTM)                (None, None, 36)          12240
_________________________________________________________________
gaussian_noise_1 (GaussianNo (None, None, 36)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, None, 36)          0
_________________________________________________________________
lstm_3 (LSTM)                (None, 32)                8832
_________________________________________________________________
gaussian_noise_2 (GaussianNo (None, 32)                0
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 31,665
Trainable params: 31,665
Non-trainable params: 0
```

I used 3 Long short-term memory (LSTM) layers for the final implementation. 2 right at the beginning and an additional one in the further course. To avoid overfitting I added 2 Gaussian Noise layers and dropouts. The model is completed by a dense layer that provides a final "time_to_failure" prediction.

The model was tested with different layers as well as with different parameters. The final model provides a repeatable result of stable values.

From a realistic point of view, the whole process is not yet robust enough for completely unknown data. Probably outliers would also influence the result. It would make sense to normalize the data in advance e.g. with MinMaxScaler. In the current state, I wouldn't trust the model completely.

Justification

Since this project is a Kaggle Competition, in addition to the benchmark score, the achieved score for the Public Leaderboard can also be used for comparison.

In this section, I compare the values of the final model with those of the benchmark and the test solution with the XGB Regressor.

KNeighborsRegressor – Benchmark	
Mean Absolute Error	*Kaggle Leaderboard*
2.156	1.72618

XGB Regressor – interim solution	
Mean Absolute Error	*Kaggle Leaderboard*
2.026	1.49898

RNN Model – final solution	
Mean Absolute Error	*Kaggle Leaderboard*
1.992	1.36877

As can be seen from the MEA values, there were significant improvements in each version. In any case, with both the interim solution (XGB Regressor) and the final solution via RNN, the error was reduced compared to the benchmark. In the Kaggle Leaderboard the improvement of the score becomes even clearer.

I think it becomes clear that my final solution is able to solve the problem.

V. Conclusion

Free-Form Visualisation

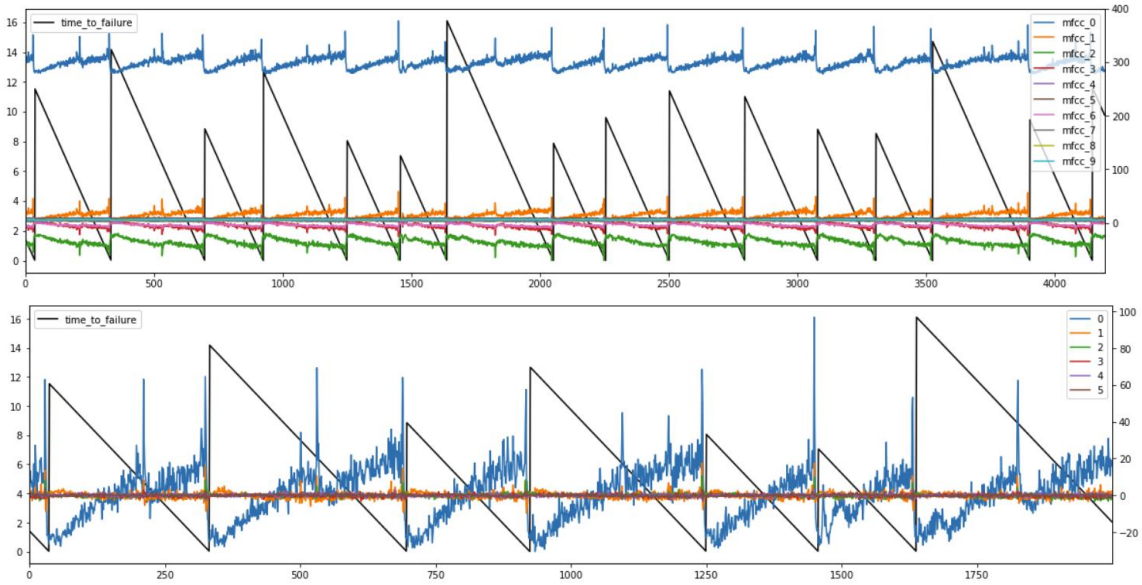

Fig. 7

Fig. 8 shows the visualization of the features used for training before and after using the PCA. As can be seen on the plot above, some MFCC features are very similar in their course (not the values). It has been shown that a reduction of the features by the PCA produces a better result since otherwise a higher weighting has been created by equal running measurement curves and possibly more significant measurement curves have been superimposed.

Reflection

The entire process can be summarized in a few steps:
1. the problem is defined and a relevant public dataset is available
2. the data were examined and analyzed
3. the data has been preprocessed and features have been extracted
4. a metric for evaluating the solution has been established
5. a simple model was created as a benchmark
6. a final model was defined and the parameters optimized
7. the final solution has been evaluated

The most interesting as well as the most difficult aspect of this project was to identify the right features. I've never been involved with signal analysis before, so it was a challenge to find the right method to work within a final solution.

On the whole, the final solution meets my expectations. However, I wouldn't see it as a general approach to solving problems of this kind, because I lack experience in signal analysis and therefore don't have a complete toolbox of knowledge to solve problems of this kind.

Improvement

I see great potential for improvement above all in the determination of the features. I had already made several attempts to suppress the original signal with various filters (e.g. Savgol Filter). But that didn't lead to a better result. Also adding new features by "classic" signal analysis would be an option. Both options require a deeper introduction to this topic.

For the implementation it would be cleaner if extracting the features for training and prediction would be done using the same method. Currently this is still a small source of error.

[1] *https://doi.org/10.1002/2017GL074677*
[2] *https://doi.org/10.1002/2017GL076708*
[3] *https://rdcu.be/bdG8Y*
[4] *https://rdcu.be/bdG9r*
[5] https://machinelearningmastery.com/reshape-input-data-long-short-term-memory-networks-keras/