Bearing Failure Anomaly Detection

This notebook follows the code of Brent Larzalere (whose words are left in quote blocks for additional info) written for his article [LSTM Autoencoder for Anomaly Detection](#) and has been modified by Sawyer Tang to generate insights on how LSTM Autoencoders can be used for Monitor-dog.

> **In this workbook, we use an autoencoder neural network to identify vibrational anomalies from sensor readings in a set of bearings. The goal is to be able to predict future bearing failures before they happen. The vibrational sensor readings are from the NASA Acoustics and Vibration Database. Each data set consists of individual files that are 1-second vibration signal snapshots recorded at 10 minute intervals. Each file contains 20,480 sensor data points that were obtained by reading the bearing sensors at a sampling rate of 20 kHz.**
>
> **This autoencoder neural network model is created using Long Short-Term Memory (LSTM) recurrent neural network (RNN) cells within the Keras / TensorFlow framework.**

In [1]:

```
import os
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import joblib
import seaborn as sns
sns.set(color_codes=True)
import matplotlib.pyplot as plt
%matplotlib inline

from numpy.random import seed
import tensorflow as tf #tensorflow2

from tensorflow.keras.layers import Input, Dropout, Dense, LSTM, TimeDistributed, Repeat
Vector
from tensorflow.keras.models import Model
from tensorflow.keras import regularizers
```

In [2]:

```
# set random seed
seed(10)
tf.random.set_seed(10)
```

Data loading and pre-processing

> **An assumption is that mechanical degradation in the bearings occurs gradually over time; therefore, we use one datapoint every 10 minutes in the analysis. Each 10 minute datapoint is aggregated by using the mean absolute value of the vibration recordings over the 20,480 datapoints in each file. We then merge together everything in a single dataframe.**

This data includes 4 columns in which the encoder will ingest and predict on a contextual basis to one another. This could be useful while using time-series data from multiple metrics from one source such as response-time vs error logs count. In order to have a reliable prediction model, each column should be aligned with each other with equal time-series intervals.

In [3]:

```
# load, average and merge sensor samples
data_dir = 'data/bearing_data'
merged_data = pd.DataFrame()
```

```
for filename in os.listdir(data_dir):
    dataset = pd.read_csv(os.path.join(data_dir, filename), sep='\t')
    dataset_mean_abs = np.array(dataset.abs().mean())
    dataset_mean_abs = pd.DataFrame(dataset_mean_abs.reshape(1,4))
    dataset_mean_abs.index = [filename]
    merged_data = merged_data.append(dataset_mean_abs)

merged_data.columns = ['Bearing 1', 'Bearing 2', 'Bearing 3', 'Bearing 4']
```

In [4]:

```
# transform data file index to datetime and sort in chronological order
merged_data.index = pd.to_datetime(merged_data.index, format='%Y.%m.%d.%H.%M.%S')
merged_data = merged_data.sort_index()
merged_data.to_csv('Averaged_BearingTest_Dataset.csv')
print("Dataset shape:", merged_data.shape)
merged_data.head()
```

```
Dataset shape: (982, 4)
```

Out[4]:

	Bearing 1	Bearing 2	Bearing 3	Bearing 4
2004-02-12 10:52:39	0.060236	0.074227	0.083926	0.044443
2004-02-12 11:02:39	0.061455	0.073844	0.084457	0.045081
2004-02-12 11:12:39	0.061361	0.075609	0.082837	0.045118
2004-02-12 11:22:39	0.061665	0.073279	0.084879	0.044172
2004-02-12 11:32:39	0.061944	0.074593	0.082626	0.044659

Define train/test data

> **Before setting up the models, we need to define train/test data. To do this, we perform a simple split where we train on the first part of the dataset (which should represent normal operating conditions) and test on the remaining parts of the dataset leading up to the bearing failure.**

The model needs a set of training data that is used to train the LSTM autoencoder. The autoencoder works best when non-anomalous data is used for training. If no non-anomalous data is available to train the auto-encoder, a labeled anomaly method of training is also available where another column composed of anomaly points is used to train the encoder model. This method can be more accurate and robust, however, it also requires a much more robust training dataset with many different anomaly types that are labeled for the encoder to learn.

In [5]:

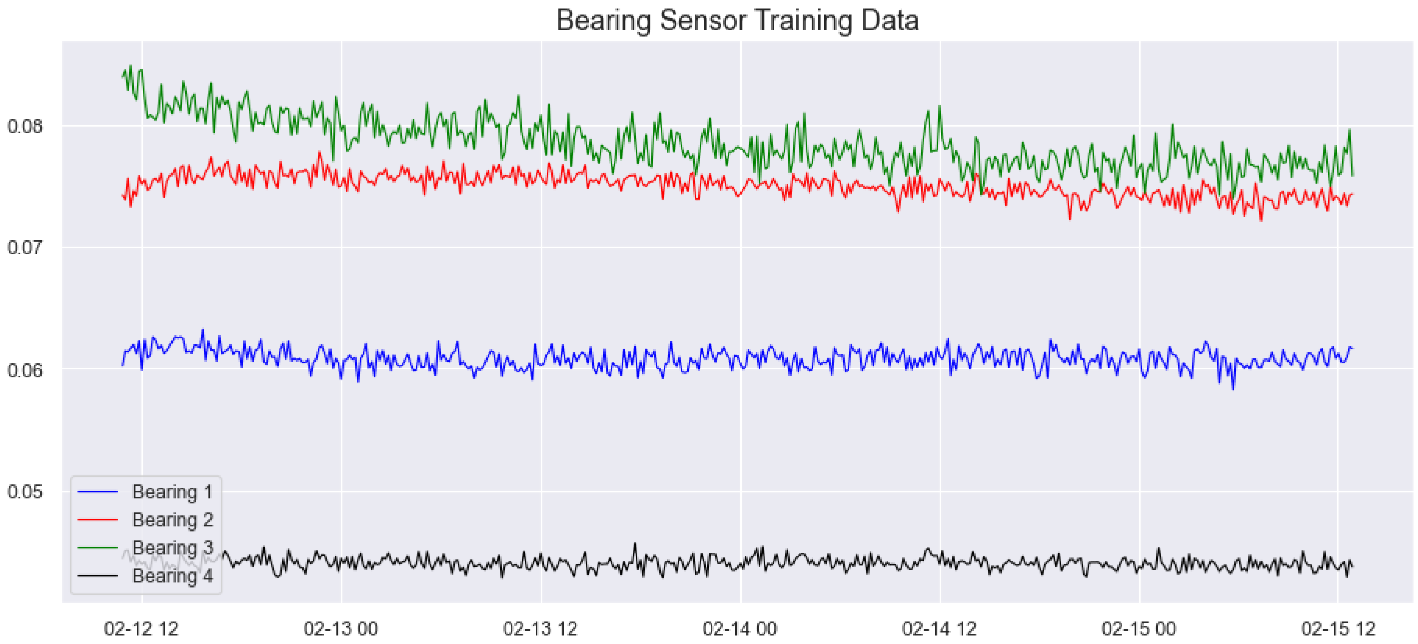
```
train = merged_data['2004-02-12 10:52:39': '2004-02-15 12:52:39']
test = merged_data['2004-02-15 12:52:39':]
print("Training dataset shape:", train.shape)
print("Test dataset shape:", test.shape)
```

```
Training dataset shape: (445, 4)
Test dataset shape: (538, 4)
```

In [6]:

```
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(train['Bearing 1'], label='Bearing 1', color='blue', animated = True, linewidth=1
)
ax.plot(train['Bearing 2'], label='Bearing 2', color='red', animated = True, linewidth=1)
ax.plot(train['Bearing 3'], label='Bearing 3', color='green', animated = True, linewidth=
1)
ax.plot(train['Bearing 4'], label='Bearing 4', color='black', animated = True, linewidth=
1)
plt.legend(loc='lower left')
ax.set_title('Bearing Sensor Training Data', fontsize=16)
```

```
plt.show()
```



> **Let’s get a different perspective of the data by transforming the signal from the time domain to the frequency domain using a discrete Fourier transform.**

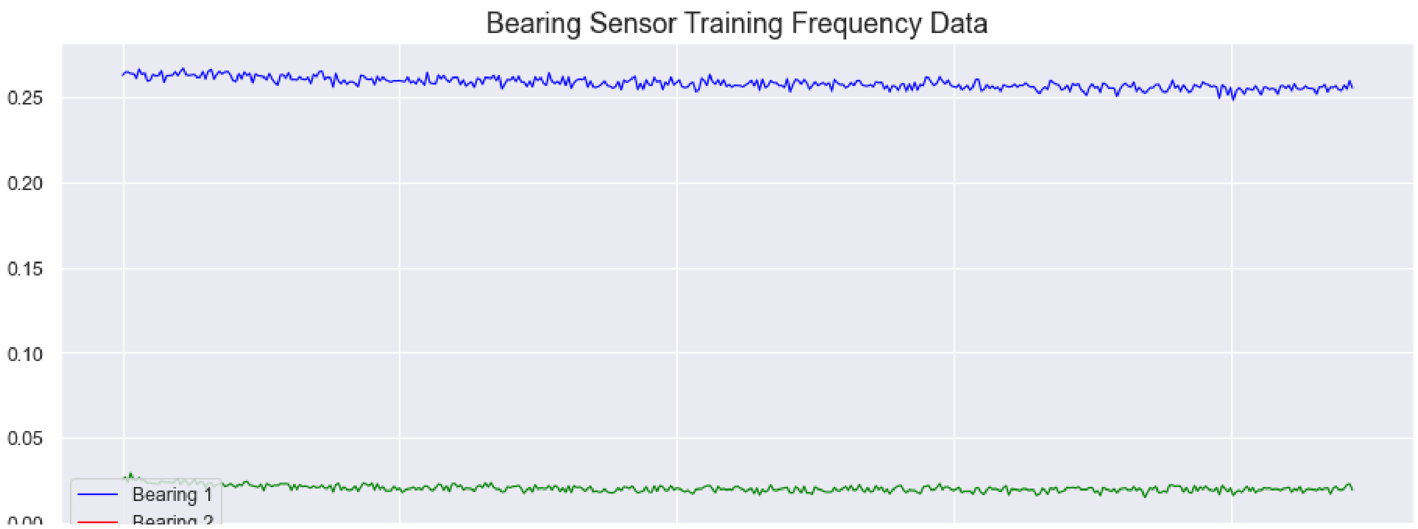
[Fast Fourier transform](#) is used here to reduce the computing time of the dataset from `O(N^2)` to `O(Nlog(N))`.

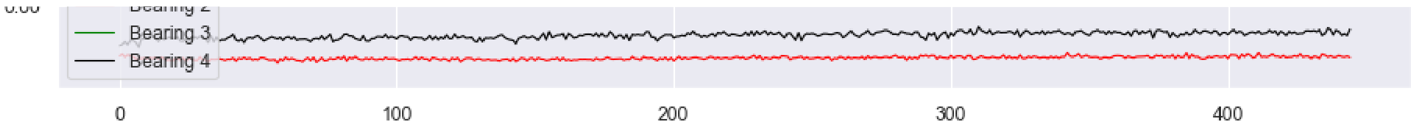
In [7]:

```
# transforming data from the time domain to the frequency domain using fast Fourier trans
form
train_fft = np.fft.fft(train)
test_fft = np.fft.fft(test)
```

In [8]:

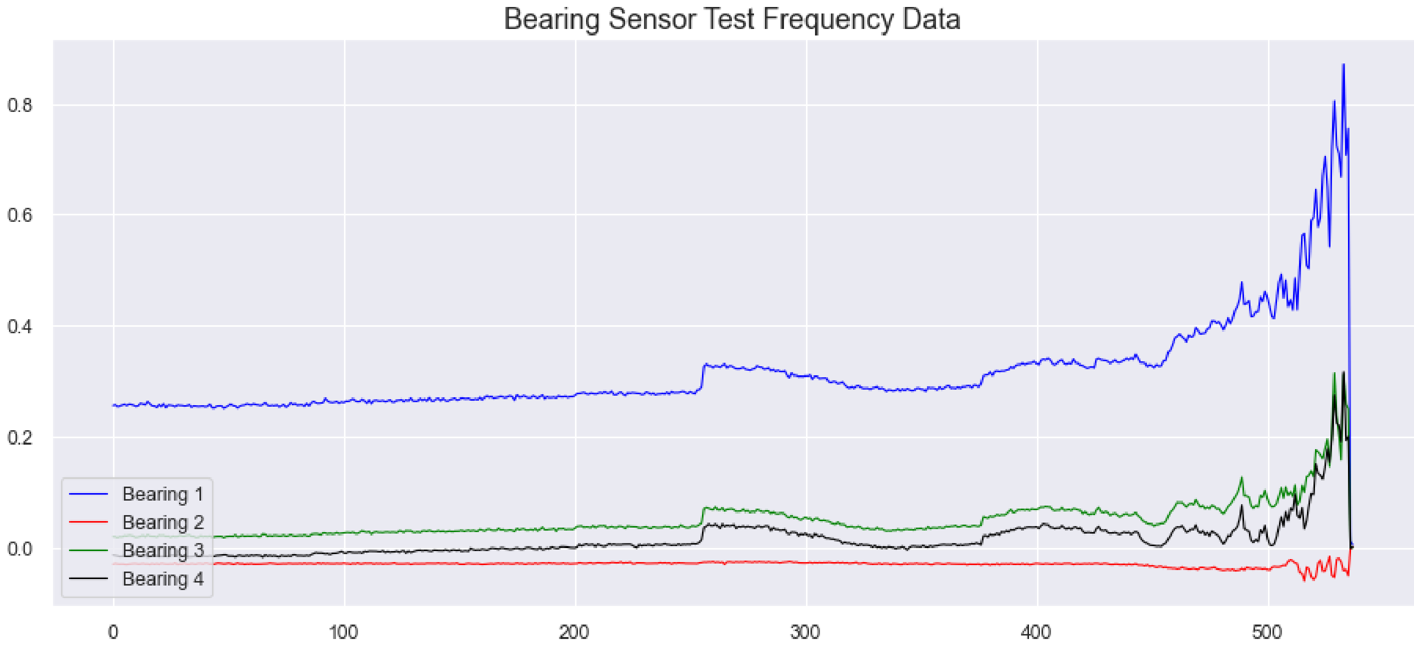
```
# frequencies of the healthy sensor signal
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(train_fft[:,0].real, label='Bearing 1', color='blue', animated = True, linewidth=
1)
ax.plot(train_fft[:,1].imag, label='Bearing 2', color='red', animated = True, linewidth=1
)
ax.plot(train_fft[:,2].real, label='Bearing 3', color='green', animated = True, linewidth
=1)
ax.plot(train_fft[:,3].real, label='Bearing 4', color='black', animated = True, linewidth
=1)
plt.legend(loc='lower left')
ax.set_title('Bearing Sensor Training Frequency Data', fontsize=16)
plt.show()
```





In [9]:

```
# frequencies of the degrading sensor signal
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(test_fft[:,0].real, label='Bearing 1', color='blue', animated = True, linewidth=1
)
ax.plot(test_fft[:,1].imag, label='Bearing 2', color='red', animated = True, linewidth=1)
ax.plot(test_fft[:,2].real, label='Bearing 3', color='green', animated = True, linewidth=
1)
ax.plot(test_fft[:,3].real, label='Bearing 4', color='black', animated = True, linewidth=
1)
plt.legend(loc='lower left')
ax.set_title('Bearing Sensor Test Frequency Data', fontsize=16)
plt.show()
```



> **To complete the pre-processing of our data, we will first normalize it to a range between 0 and 1. Then we reshape our data into a format suitable for input into an LSTM network. LSTM cells expect a 3 dimensional tensor of the form [data samples, time steps, features]. Here, each sample input into the LSTM network represents one step in time and contains 4 features — the sensor readings for the four bearings at that time step.**

In [10]:

```
# normalize the data
scaler = MinMaxScaler()
X_train = scaler.fit_transform(train)
X_test = scaler.transform(test)
scaler_filename = "scaler_data"
joblib.dump(scaler, scaler_filename)
```

Out[10]:

```
['scaler_data']
```

In [11]:

```
# reshape inputs for LSTM [samples, timesteps, features]
X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
print("Training data shape:", X_train.shape)
X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])
```

```
print("Test data shape:", X_test.shape)
```

```
Training data shape: (445, 1, 4)
Test data shape: (538, 1, 4)
```

In [12]:

```
# define the autoencoder network model
def autoencoder_model(X):
    inputs = Input(shape=(X.shape[1], X.shape[2]))
    L1 = LSTM(16, activation='relu', return_sequences=True,
              kernel_regularizer=regularizers.l2(0.00))(inputs)
    L2 = LSTM(4, activation='relu', return_sequences=False)(L1)
    L3 = RepeatVector(X.shape[1])(L2)
    L4 = LSTM(4, activation='relu', return_sequences=True)(L3)
    L5 = LSTM(16, activation='relu', return_sequences=True)(L4)
    output = TimeDistributed(Dense(X.shape[2]))(L5)
    model = Model(inputs=inputs, outputs=output)
    return model
```

In [13]:

```
# create the autoencoder model
model = autoencoder_model(X_train)
model.compile(optimizer='adam', loss='mae')
model.summary()
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 1, 4)]            0         
_________________________________________________________________
lstm (LSTM)                  (None, 1, 16)             1344      
_________________________________________________________________
lstm_1 (LSTM)                (None, 4)                 336       
_________________________________________________________________
repeat_vector (RepeatVector) (None, 1, 4)              0         
_________________________________________________________________
lstm_2 (LSTM)                (None, 1, 4)              144       
_________________________________________________________________
lstm_3 (LSTM)                (None, 1, 16)             1344      
_________________________________________________________________
time_distributed (TimeDistri (None, 1, 4)              68        
=================================================================
Total params: 3,236
Trainable params: 3,236
Non-trainable params: 0
_________________________________________________________________
```

In [14]:

```
# fit the model to the data
nb_epochs = 100
batch_size = 10
history = model.fit(X_train, X_train, epochs=nb_epochs, batch_size=batch_size,
                    validation_split=0.05).history
```

```
Train on 422 samples, validate on 23 samples
Epoch 1/100
422/422 [==============================] - 10s 23ms/sample - loss: 0.4449 - val_loss: 0.3
185
Epoch 2/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.3809 - val_loss: 0.246
0
Epoch 3/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.2928 - val_loss: 0.164
4
Epoch 4/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.1667 - val_loss: 0.167
6
Epoch 5/100
```

```
422/422 [==============================] - 0s 1ms/sample - loss: 0.1133 - val_loss: 0.125
6
Epoch 6/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.1057 - val_loss: 0.122
0
Epoch 7/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.1032 - val_loss: 0.115
0
Epoch 8/100
422/422 [==============================] - 0s 982us/sample - loss: 0.1018 - val_loss: 0.1
132
Epoch 9/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.1011 - val_loss: 0.112
7
Epoch 10/100
422/422 [==============================] - 0s 984us/sample - loss: 0.1008 - val_loss: 0.1
132
Epoch 11/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.1004 - val_loss: 0.113
7
Epoch 12/100
422/422 [==============================] - 0s 981us/sample - loss: 0.1013 - val_loss: 0.1
108
Epoch 13/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0999 - val_loss: 0.111
5
Epoch 14/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0998 - val_loss: 0.110
6
Epoch 15/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0994 - val_loss: 0.111
1
Epoch 16/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0995 - val_loss: 0.109
5
Epoch 17/100
422/422 [==============================] - 0s 981us/sample - loss: 0.0990 - val_loss: 0.1
106
Epoch 18/100
422/422 [==============================] - 0s 994us/sample - loss: 0.0992 - val_loss: 0.1
097
Epoch 19/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0986 - val_loss: 0.108
8
Epoch 20/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0983 - val_loss: 0.109
5
Epoch 21/100
422/422 [==============================] - 0s 978us/sample - loss: 0.0983 - val_loss: 0.1
074
Epoch 22/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0980 - val_loss: 0.106
3
Epoch 23/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0979 - val_loss: 0.104
2
Epoch 24/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0981 - val_loss: 0.105
5
Epoch 25/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0977 - val_loss: 0.105
8
Epoch 26/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0974 - val_loss: 0.104
6
Epoch 27/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0973 - val_loss: 0.106
9
Epoch 28/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0970 - val_loss: 0.103
8
Epoch 29/100
```

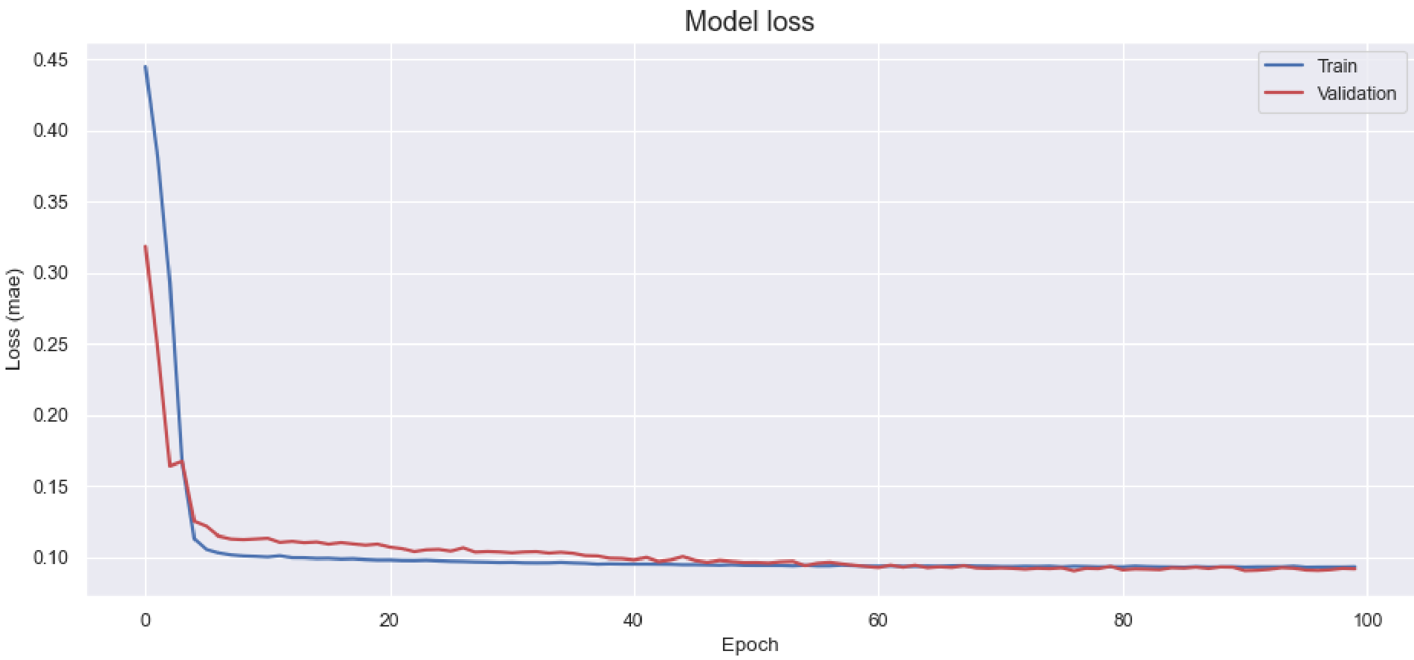
```
422/422 [==============================] - 0s 994us/sample - loss: 0.0969 - val_loss: 0.1
042
Epoch 30/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0966 - val_loss: 0.103
9
Epoch 31/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0968 - val_loss: 0.103
3
Epoch 32/100
422/422 [==============================] - 0s 995us/sample - loss: 0.0964 - val_loss: 0.1
039
Epoch 33/100
422/422 [==============================] - 0s 987us/sample - loss: 0.0963 - val_loss: 0.1
042
Epoch 34/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0964 - val_loss: 0.103
1
Epoch 35/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0967 - val_loss: 0.103
7
Epoch 36/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0963 - val_loss: 0.103
0
Epoch 37/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0961 - val_loss: 0.101
3
Epoch 38/100
422/422 [==============================] - 0s 998us/sample - loss: 0.0955 - val_loss: 0.1
011
Epoch 39/100
422/422 [==============================] - 0s 999us/sample - loss: 0.0957 - val_loss: 0.0
995
Epoch 40/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0956 - val_loss: 0.099
3
Epoch 41/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0955 - val_loss: 0.098
4
Epoch 42/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0956 - val_loss: 0.100
1
Epoch 43/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0955 - val_loss: 0.097
2
Epoch 44/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0954 - val_loss: 0.098
4
Epoch 45/100
422/422 [==============================] - 0s 988us/sample - loss: 0.0951 - val_loss: 0.1
007
Epoch 46/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0951 - val_loss: 0.098
0
Epoch 47/100
422/422 [==============================] - 0s 999us/sample - loss: 0.0950 - val_loss: 0.0
967
Epoch 48/100
422/422 [==============================] - 0s 991us/sample - loss: 0.0948 - val_loss: 0.0
981
Epoch 49/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0951 - val_loss: 0.097
3
Epoch 50/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0947 - val_loss: 0.096
6
Epoch 51/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0946 - val_loss: 0.096
6
Epoch 52/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0946 - val_loss: 0.096
3
Epoch 53/100
```

```
422/422 [==============================] - 0s 1ms/sample - loss: 0.0946 - val_loss: 0.097
1
Epoch 54/100
422/422 [==============================] - 0s 995us/sample - loss: 0.0943 - val_loss: 0.0
974
Epoch 55/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0947 - val_loss: 0.094
5
Epoch 56/100
422/422 [==============================] - 0s 986us/sample - loss: 0.0942 - val_loss: 0.0
962
Epoch 57/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0943 - val_loss: 0.096
8
Epoch 58/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0948 - val_loss: 0.095
7
Epoch 59/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0944 - val_loss: 0.094
7
Epoch 60/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0941 - val_loss: 0.093
9
Epoch 61/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0941 - val_loss: 0.093
1
Epoch 62/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0943 - val_loss: 0.094
8
Epoch 63/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0940 - val_loss: 0.093
3
Epoch 64/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0940 - val_loss: 0.094
7
Epoch 65/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0942 - val_loss: 0.093
0
Epoch 66/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0940 - val_loss: 0.093
7
Epoch 67/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0944 - val_loss: 0.093
2
Epoch 68/100
422/422 [==============================] - 0s 996us/sample - loss: 0.0945 - val_loss: 0.0
944
Epoch 69/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0942 - val_loss: 0.092
8
Epoch 70/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0941 - val_loss: 0.092
5
Epoch 71/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0938 - val_loss: 0.092
8
Epoch 72/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0938 - val_loss: 0.092
5
Epoch 73/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0940 - val_loss: 0.091
9
Epoch 74/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0939 - val_loss: 0.092
6
Epoch 75/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0941 - val_loss: 0.092
3
Epoch 76/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0936 - val_loss: 0.092
9
Epoch 77/100
```

```
422/422 [==============================] - 0s 1ms/sample - loss: 0.0940 - val_loss: 0.090
7
Epoch 78/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0939 - val_loss: 0.092
6
Epoch 79/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0936 - val_loss: 0.092
3
Epoch 80/100
422/422 [==============================] - 0s 988us/sample - loss: 0.0937 - val_loss: 0.0
939
Epoch 81/100
422/422 [==============================] - 0s 974us/sample - loss: 0.0936 - val_loss: 0.0
914
Epoch 82/100
422/422 [==============================] - 0s 984us/sample - loss: 0.0941 - val_loss: 0.0
921
Epoch 83/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0938 - val_loss: 0.091
9
Epoch 84/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0936 - val_loss: 0.091
6
Epoch 85/100
422/422 [==============================] - 0s 987us/sample - loss: 0.0935 - val_loss: 0.0
929
Epoch 86/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0934 - val_loss: 0.092
7
Epoch 87/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0938 - val_loss: 0.093
4
Epoch 88/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0935 - val_loss: 0.092
4
Epoch 89/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0936 - val_loss: 0.093
5
Epoch 90/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0937 - val_loss: 0.093
4
Epoch 91/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0934 - val_loss: 0.090
9
Epoch 92/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0936 - val_loss: 0.091
2
Epoch 93/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0936 - val_loss: 0.091
8
Epoch 94/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0936 - val_loss: 0.093
0
Epoch 95/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0941 - val_loss: 0.092
6
Epoch 96/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0933 - val_loss: 0.091
3
Epoch 97/100
422/422 [==============================] - 0s 999us/sample - loss: 0.0935 - val_loss: 0.0
910
Epoch 98/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0935 - val_loss: 0.091
5
Epoch 99/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0935 - val_loss: 0.092
5
Epoch 100/100
422/422 [==============================] - 0s 1ms/sample - loss: 0.0937 - val_loss: 0.092
2
```

In [15]:

```
# plot the training losses
fig, ax = plt.subplots(figsize=(14, 6), dpi=80)
ax.plot(history['loss'], 'b', label='Train', linewidth=2)
ax.plot(history['val_loss'], 'r', label='Validation', linewidth=2)
ax.set_title('Model loss', fontsize=16)
ax.set_ylabel('Loss (mae)')
ax.set_xlabel('Epoch')
ax.legend(loc='upper right')
plt.show()
```



Distribution of Loss Function

> **By plotting the distribution of the calculated loss in the training set, one can use this to identify a suitable threshold value for identifying an anomaly. In doing this, one can make sure that this threshold is set above the "noise level" and that any flagged anomalies should be statistically significant above the background noise.**

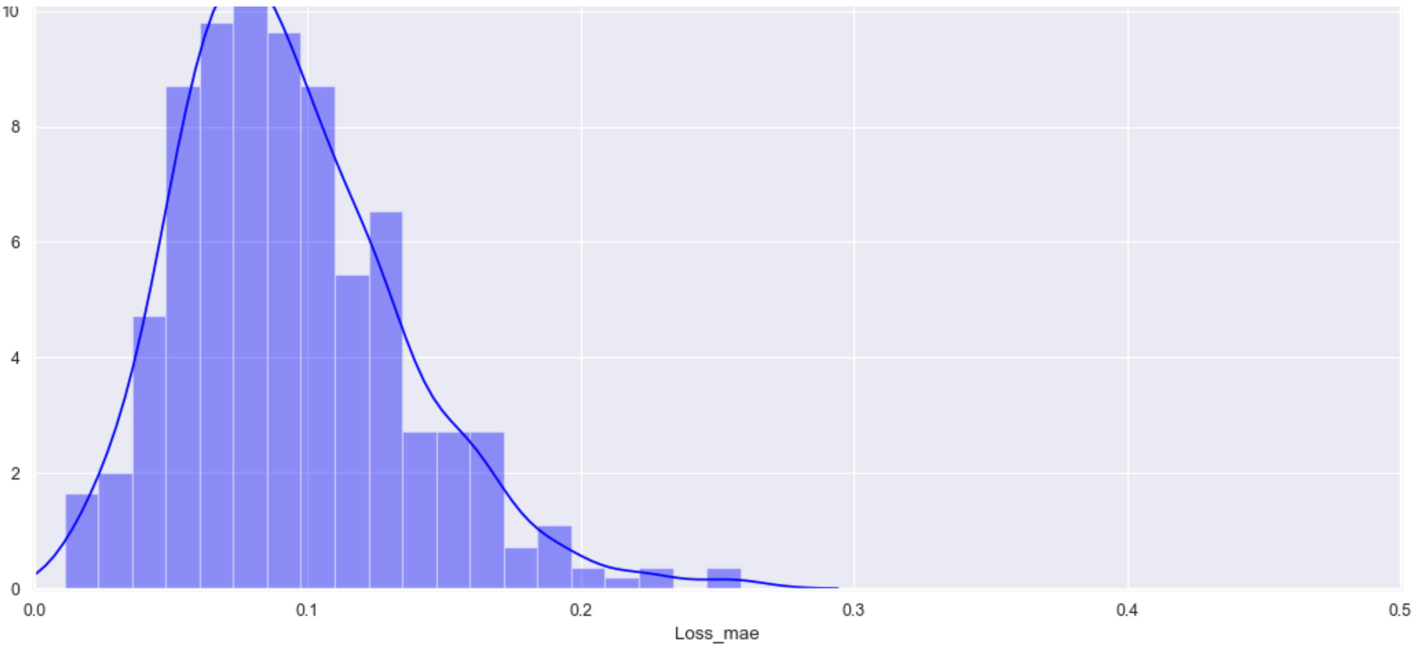
In [16]:

```
# plot the loss distribution of the training set
X_pred = model.predict(X_train)
X_pred = X_pred.reshape(X_pred.shape[0], X_pred.shape[2])
X_pred = pd.DataFrame(X_pred, columns=train.columns)
X_pred.index = train.index

scored = pd.DataFrame(index=train.index)
Xtrain = X_train.reshape(X_train.shape[0], X_train.shape[2])
scored['Loss_mae'] = np.mean(np.abs(X_pred-Xtrain), axis = 1)
plt.figure(figsize=(16,9), dpi=80)
plt.title('Loss Distribution', fontsize=16)
sns.distplot(scored['Loss_mae'], bins = 20, kde= True, color = 'blue');
plt.xlim([0.0,.5])
```

Out[16]:

```
(0.0, 0.5)
```



> **From the above loss distribution, let's try a threshold value of 0.275 for flagging an anomaly. We can then calculate the loss in the test set to check when the output crosses the anomaly threshold.**

The threshold can also be determined by using the maximum MAE loss or any way that seems reasonable.

In [17]:

```
# calculate the loss on the test set
X_pred = model.predict(X_test)
X_pred = X_pred.reshape(X_pred.shape[0], X_pred.shape[2])
X_pred = pd.DataFrame(X_pred, columns=test.columns)
X_pred.index = test.index

scored = pd.DataFrame(index=test.index)
Xtest = X_test.reshape(X_test.shape[0], X_test.shape[2])
scored['Loss_mae'] = np.mean(np.abs(X_pred-Xtest), axis = 1)
scored['Threshold'] = 0.275
scored['Anomaly'] = scored['Loss_mae'] > scored['Threshold']
scored.head()
```

Out[17]:

	Loss_mae	Threshold	Anomaly
2004-02-15 12:52:39	0.091773	0.275	False
2004-02-15 13:02:39	0.171712	0.275	False
2004-02-15 13:12:39	0.066633	0.275	False
2004-02-15 13:22:39	0.052919	0.275	False
2004-02-15 13:32:39	0.039105	0.275	False

In [18]:

```
# calculate the same metrics for the training set
# and merge all data in a single dataframe for plotting
X_pred_train = model.predict(X_train)
X_pred_train = X_pred_train.reshape(X_pred_train.shape[0], X_pred_train.shape[2])
X_pred_train = pd.DataFrame(X_pred_train, columns=train.columns)
X_pred_train.index = train.index

scored_train = pd.DataFrame(index=train.index)
scored_train['Loss_mae'] = np.mean(np.abs(X_pred_train-Xtrain), axis = 1)
scored_train['Threshold'] = 0.275
scored_train['Anomaly'] = scored_train['Loss_mae'] > scored_train['Threshold']
```

```
scored = pd.concat([scored_train, scored])
```

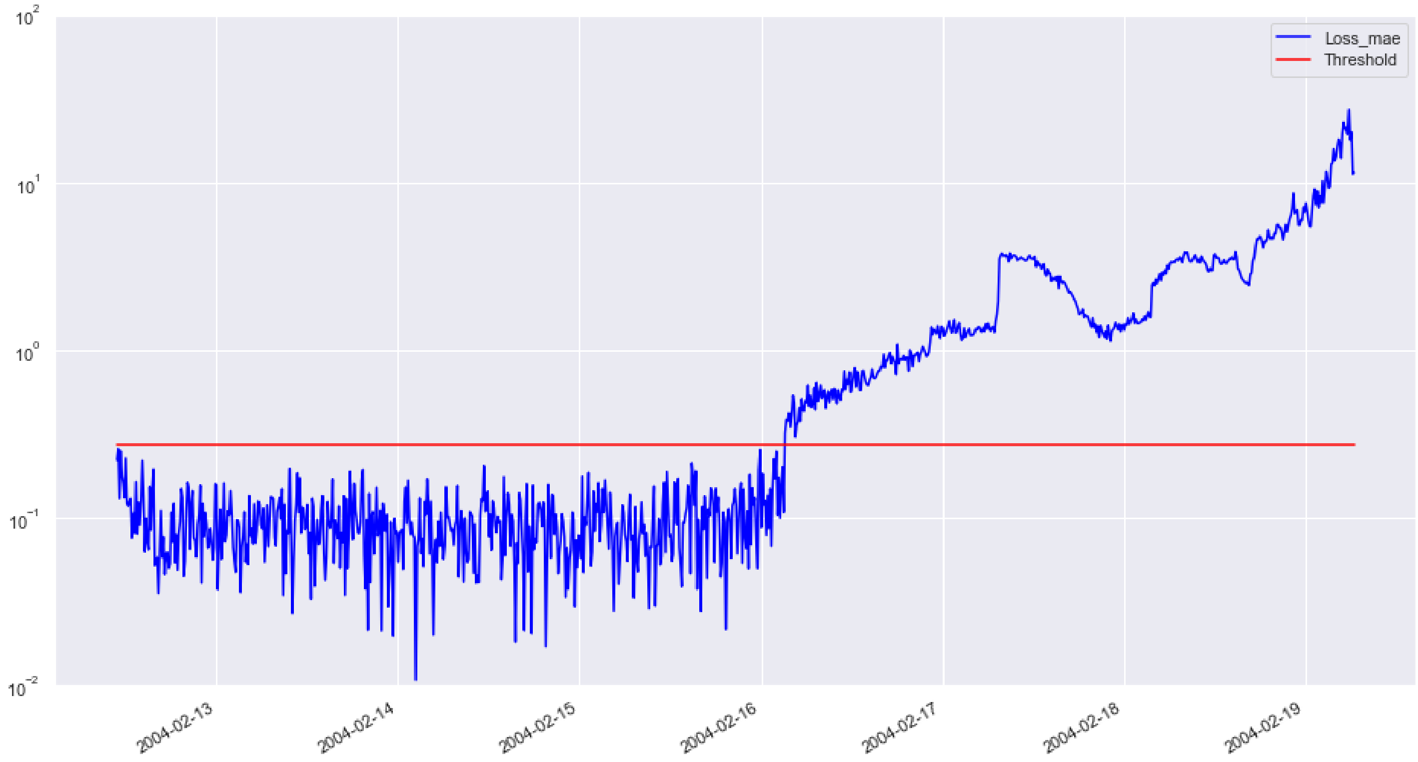
> **Having calculated the loss distribution and the anomaly threshold, we can visualize the model output in the time leading up to the bearing failure.**

In [19]:

```
# plot bearing failure time plot
scored.plot(logy=True,  figsize=(16,9), ylim=[1e-2,1e2], color=['blue','red'])
```

Out[19]:

```
<AxesSubplot:>
```



> **This analysis approach is able to flag the upcoming bearing malfunction well in advance of the actual physical failure. It is important to define a suitable threshold value for flagging anomalies while avoiding too many false positives during normal operating conditions.**

In [20]:

```
# save all model information, including weights, in h5 format
model.save("keras_model.h5")
```

Convert keras model to Onnx

In [21]:

```
# convert to onnx model
import onnx
import keras2onnx

onnx_model = keras2onnx.convert_keras(model, model.name)
onnx.save(onnx_model, 'onnx_model.onnx')
```

```
tf executing eager_mode: True
tf.keras model eager_mode: False
The ONNX operator number change on the optimization: 26 -> 16
```

```
The maximum opset needed by this model is only 11.
```