循环神经网络 RNN & 在新奇检测中的运用 - Part 2

与卷积网络和多层感知机不同，循环神经网络（Recurrent Neural Network）为了更好地处理时序信息（Time Series）而设计的。它的特征在于利用状态变量来储存过去的信息，并和当前时间的输入一起共同决定输出。

语言模型就是我们生活中最常见的时序信息。一句话里的每一个字都是按时间顺序出现的，之前说的话一定会决定当前说出口的单词。所以RNN被广泛用于语音识别、语言模型。文本型数据也是同理。但RNN不局限于语言和文字，只要是按时间采样的数据都能很好的被RNN处理，比如市场价格。我们可以用RNN来辨别一个时间段中，市场价格的非常规波动点，也就是新奇检测。本文的运用部分也会侧重于介绍RNN在新奇检测中的运用。

本文一共分为两个部分：

- Part 1 主要讲述RNN和其衍生LSTM的理论基础
- **Part 2 是一个LSTM在新奇检测场景中的运用和代码实现**

3 用RNN进行新奇检测

在这里我们主要给读者展示一下RNN在新奇检测场景下的实现。其中我们用到了 `Tensorflow` 和 `Keras` 进行深度学习。以下是我们这次要用到的包：

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

from numpy.random import seed
from sklearn.preprocessing import MinMaxScaler
import joblib
import tensorflow as tf
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

from keras.layers import Input, Dropout, Dense, LSTM, TimeDistributed, RepeatVector
from keras.models import Model
from keras import regularizers

# set random seeds
seed(128)
tf.random.set_seed(128)
```

3.1 数据集

我们将使用来自 NASA 声学和振动数据库的振动传感器读取作为本研究的数据集。 在 NASA 的研究中，传感器读数是在四个轴承上采集的，这些轴承会在多天的恒定负载下运行到故障为止。我们要用到的数据集由很多单个文件组成，这些文件是以 10 分钟为间隔记录的 1 秒振动信号快照。 每个文件包含每个轴承 20,480 个传感器数据点，这些数据点是通过以 20 kHz 的采样率读取轴承传感器而获得的。

由于这些数据还不是 `.csv` 的形式，而且是很多个独立的文件，我们首先要做的就是读取并整合这些数据。

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

for filename in os.listdir(data_dir):
    if filename == "__MACOSX":
        continue
    dataset = pd.read_csv(os.path.join(data_dir, filename), sep='\t')
    dataset_mean_abs = pd.DataFrame(np.array(dataset.abs().mean()).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']

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(10)
```

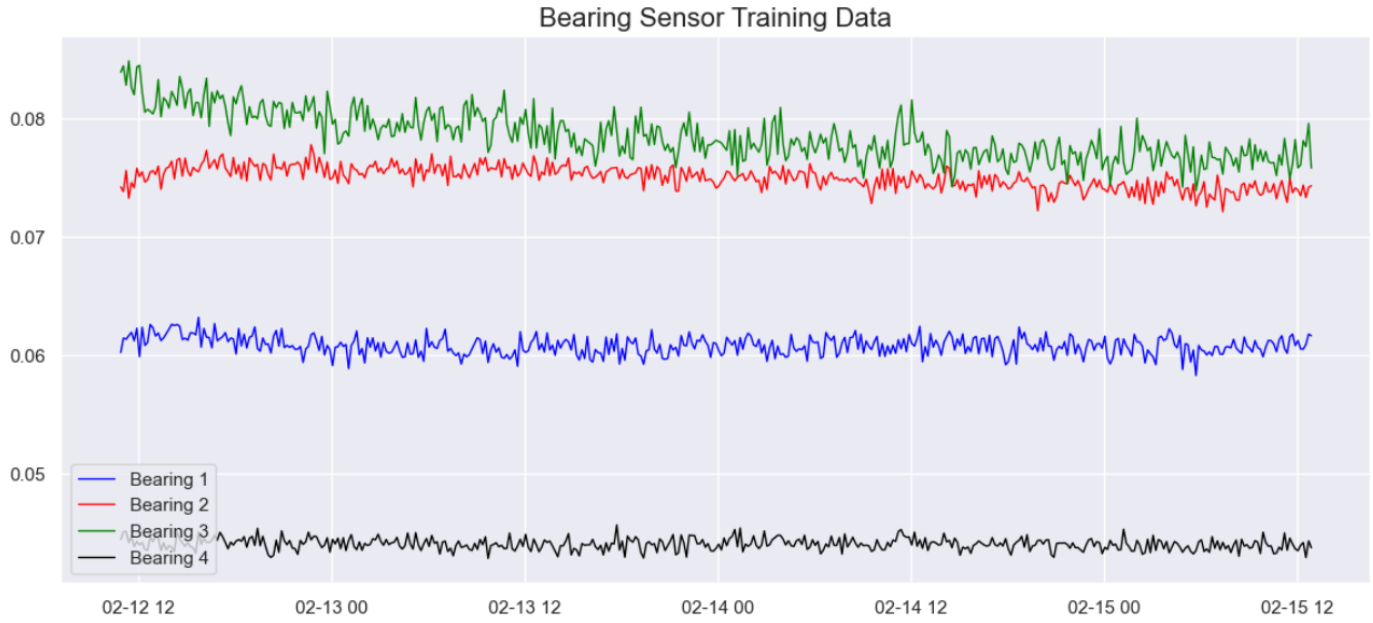
```
Dataset shape: (982, 4)
```

:

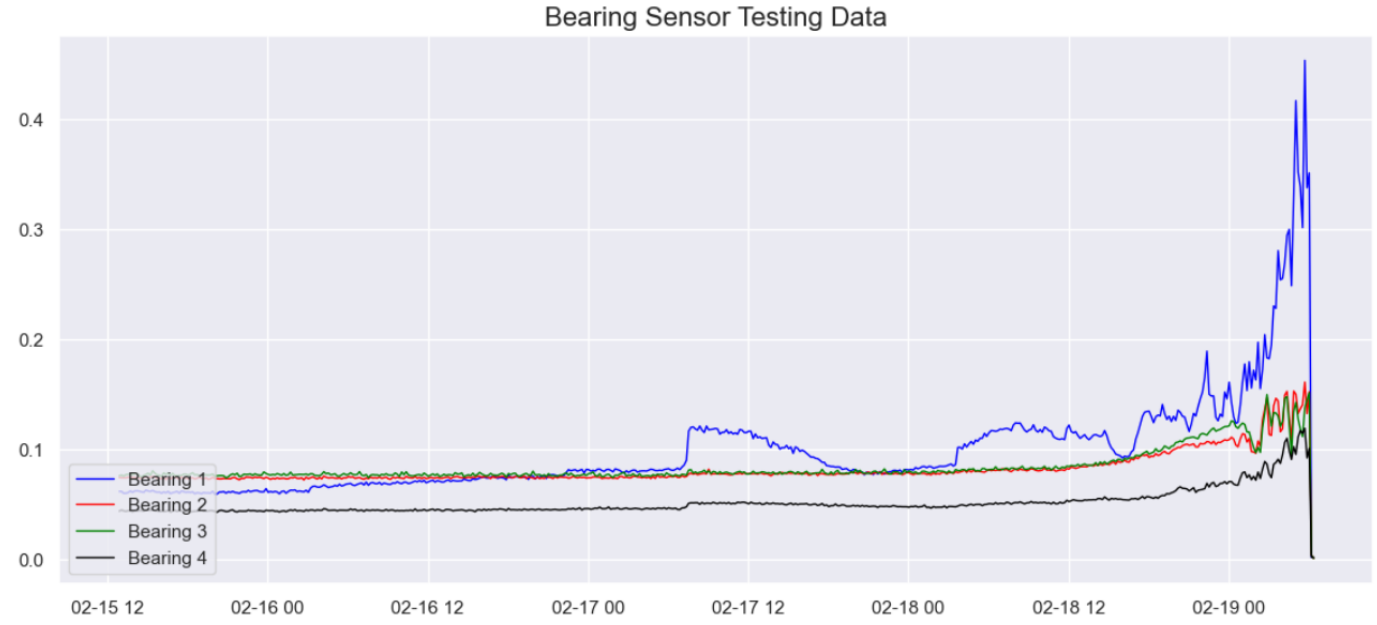
	Bearing 1	Bearing 2	Bearing 3	Bearing 4
2004-02-12 10:52:39	0.060236	0.074227	0.083926	0.044443
2004-02-12 11:02:39	0.061455	0.073844	0.084457	0.045081
2004-02-12 11:12:39	0.061361	0.075609	0.082837	0.045118
2004-02-12 11:22:39	0.061665	0.073279	0.084879	0.044172
2004-02-12 11:32:39	0.061944	0.074593	0.082626	0.044659
2004-02-12 11:42:39	0.061231	0.074172	0.082022	0.043840
2004-02-12 11:52:39	0.062280	0.075808	0.084372	0.044272
2004-02-12 12:02:39	0.059890	0.075115	0.084506	0.043974
2004-02-12 12:12:39	0.062371	0.075505	0.082238	0.044171
2004-02-12 12:22:39	0.060837	0.074648	0.080552	0.043602

一共有982个数据点，每个数据点采样的间隔均为10秒。所以这是一个**时序信息**。我们把 `2004-02-15 12:52:39` 之前的数据作为我们的训练集，这个时间点之后的数据作为测试集。用这个方法区分是通过将数据可视化之后来来判断的：

- 训练集 `2004-02-12 10:52:39` 到 `2004-02-15 12:52:39`



- 测试集 `2004-02-15 12:52:39` 到最后



请注意两张图的纵轴范围：训练集的波动其实非常小，四个轴承的数据波动都在0.005范围之内。所以测试集里的前半程数据是和训练集的走向是高度重合的。而后半程就开始出现非常规的波动了。所以我们按照上述的时间来区分训练和测试数据是合理的。

3.2 LSTM在新奇检测中的角色：自编码器

首先，我刚才提到过这数据是个**时序信息**，所以我们用LSTM来捕捉信息的流动是合理的。但应该怎么利用LSTM呢？这里我们引入一个概念：**自编码器**（Autoencoder）。

顾名思义，自编码器同时囊括了编码和解码，其本质是让模型学习一个 “恒等函数”。它将输入的数据压缩，用来表示该数据的核心/主要驱动特征；随后将该压缩重新构筑成原数据的模样。举个例子：输入一

张狗的照片，自编码器会学习尽可能地压缩该图片，只保留其作为狗独有的特征；随后再学习如何将这张图片还原出来。

那自编码器又是如何在新奇检测中发挥的？它的训练过程是基于正常数据的，我们会得到一组重构误差。模型在接着面对一些新奇异常点时，也会进行相同的编码和重构步骤。但由于模型没有训练过这些异常点，所以**异常点的重构误差就要比正常数据大很多**。我们可以寻找一个阈值，当一个数据点的重构误差大于这个阈值时，我们就判定他是异常点。接下来我们就来看以上这个算法的实现。

3.3 LSTM自编码器神经网络实现

在训练之前，一定要记得把训练集和数据集都正则化。笔者用的是 `sklearn.preprocessing` 里的 `MinMaxScaler` 。在使用这个包的时候，务必记住：在训练集中训练，再用训练好的正则化模型对测试集进行转换。

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

Out:

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

注意，为了贴合LSTM的输入格式（数据个数 * 时间步长* 特征个数），我们要基于数据集一个额外的时间步长维度：每个数据点的时间步长均为1。

接着我们用Keras包来构建LSTM自编码器神经网络：

```
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
    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
```

其中，L1、L2是由LSTM层组成的编码层，L4、L5是有LSTM层组成的解码层，最后通过全连接层以后输出结果。

我们用‘ADAM’作为优化器（所以这里就没有学习率这个超参数了），重构误差用平均绝对误差 MAE（Mean Squared Error）来计算。关于ADAM可以参考这篇博客。 `model.summary()` 还可以展示神经网络模型的一些信息。

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

Out:

```
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
_________________________________________________________________
```

可以开始训练了，用GPU还是很快的：

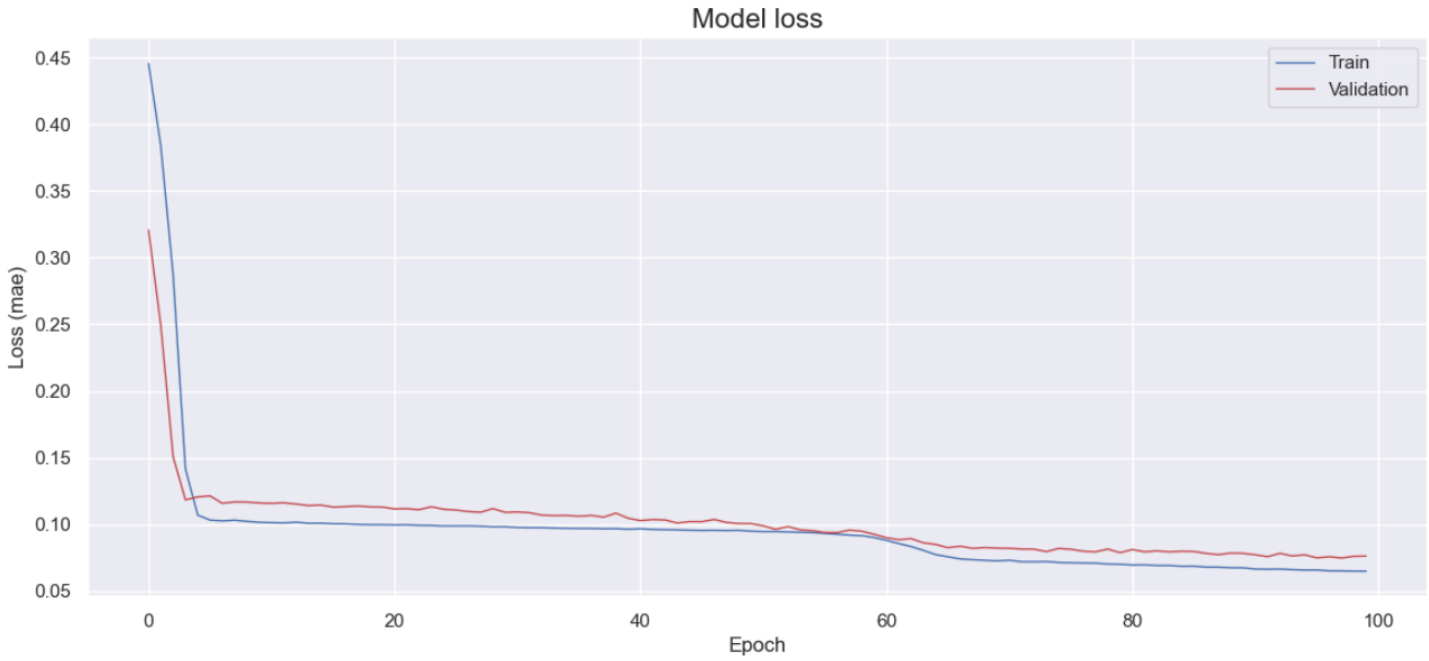
```
# fit the model to the data
n_epochs = 100
batch_size = 10

import time

t0 = time.time()
history = model.fit(X_train, X_train, epochs=n_epochs, batch_size=batch_size,
                    validation_split=0.05).history
print('Fitting time: %.4f s'% (time.time()-t0))
```

Out:

```
Fitting time: 166.3652 s
```

3.4 训练成果与预测

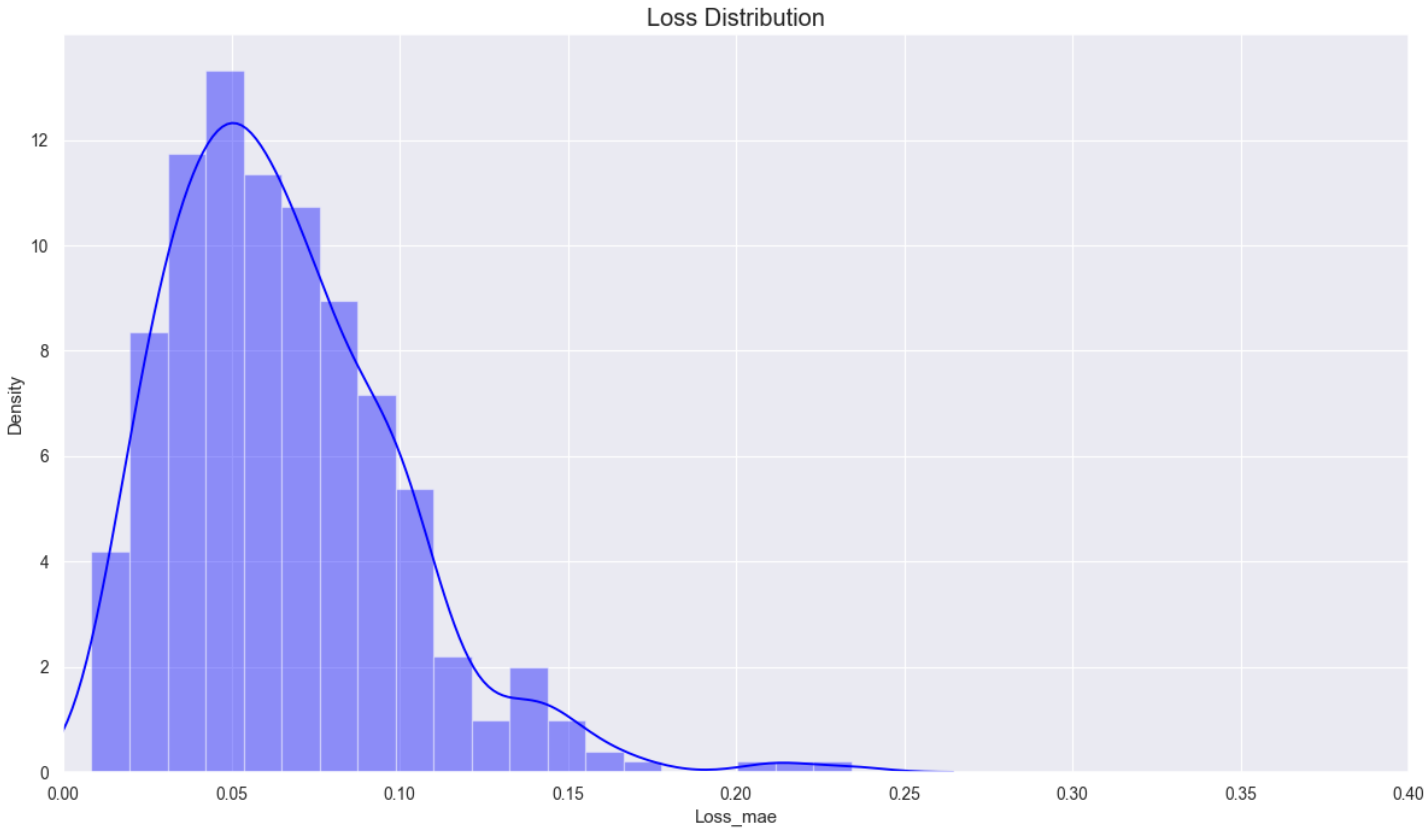
先画一下训练的MAE，直接从 `history` 里调用就行了：



要注意的是不能把这个计算出的loss去和原数据做对比，因为原数据已经被正则化过了。

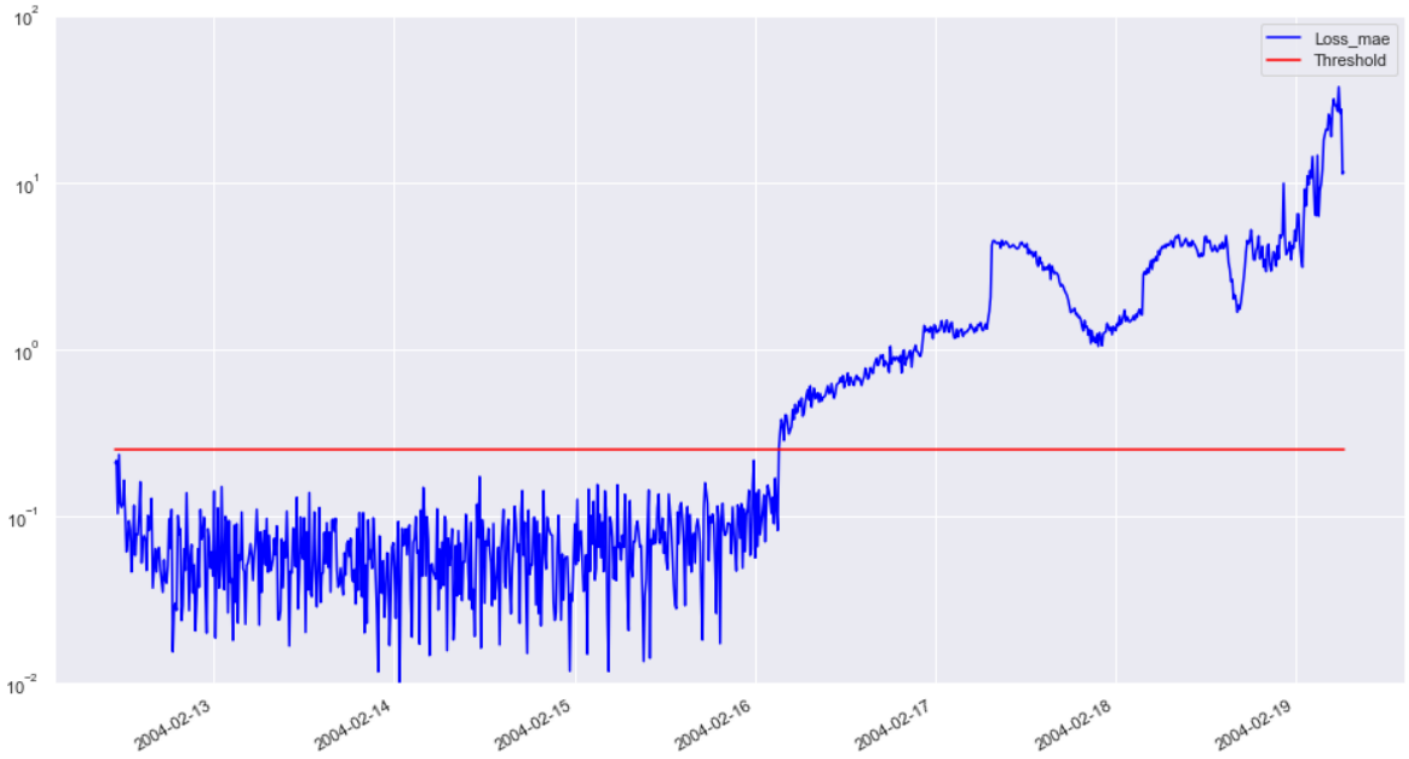
根据上个部分对算法的描述，我们应该重新预测一遍训练集，得到一组重构误差作为基准线。画出重组误差的分布：

```
# plot the loss distribution of 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_train = pd.DataFrame(index=train.index)
Xtrain = X_train.reshape(X_train.shape[0], X_train.shape[2])
scored_train['Loss_mae'] = np.mean(np.abs(Xtrain-X_pred), axis = 1)
plt.figure(figsize=(16,9), dpi=90)
plt.title('Loss Distribution', fontsize = 16)
sns.distplot(scored_train['Loss_mae'], bins=20, kde=True, color = 'blue')
plt.xlim([0.0, .4])
```



我们需要以这张图作为依据选出一个阈值来区分是否为异常值：重构误差在阈值之下是正常值，之上是异常值。0.25 刚好卡在这个训练集的重构误差的最大值左右，是一个挺不错的选择，可以有效避免假正值（FP）过高的情况。

我们来看一下这个阈值的效果：用和上面极其相似的代码可以实现计算预测集的重构误差，并且将其和训练集的重构误差放在一张图中，加入阈值进行观察。



我们重点对这张图进行分析：

- 红线是阈值。抛开阈值不谈，我们可以观察到重构误差的两极分化很明显：
  在 `2004-02-16` 到 `2004-02-17` 之间有一次非寻常的跳动。
- 在该跳动的时间点之前，重构误差有着较为规律性的波动；在该时间点之后，误差的波动变得非常混沌。
- 阈值的引入成功地分割了这个跳动：在这个跳动的时间点之前，所有误差均在阈值之下；在该时间点之后，所有的误差都在阈值之上。
- 训练集的最终时间点是 `2004-02-15 12:52:39` ，所以误差在阈值之下的数据点包含了一部分预测集。LSTM成功地捕捉到了这些正常的时序信息。
- 如果我们把该非寻常跳动的时间点和之前训练集的可视化图进行对比，会发现在同一时间处，轴承1（Bearing 1）有一次非寻常的显著上升。所以在这个时间点，轴承1已经不是正常运作了，随着时间的推移，带动了别的轴承减减偏离正常趋势。

3.5 保存模型

最后，我们可以把模型储存在一个h5文件里方便下次调用：

```
model.save("name.h5")
```

Reference

1. https://towardsdatascience.com/lstm-autoencoder-for-anomaly-detection-e1f4f2ee7ccf