Contents

Next value prediction

Data Understanding

There two sets of data one for training (train.csv) and one from testing (test.csv).

The train dataset contains 3 different features: x, y and z components of acceleration. These were collected every 10 seconds for a total of 144911 records.

Here are the first few rows of the train dataset:

No.	x	y	z
0	-24	749	-626
1	-206	930	-63
2	-139	763	-577
3	-503	441	-557
4	-278	705	-396
5	240	839	-310
6	-671	318	-213
7	-45	296	-927
8	102	294	-888
9	15	635	-671

TODO caption table

Here is the time evolution of the three features over time:

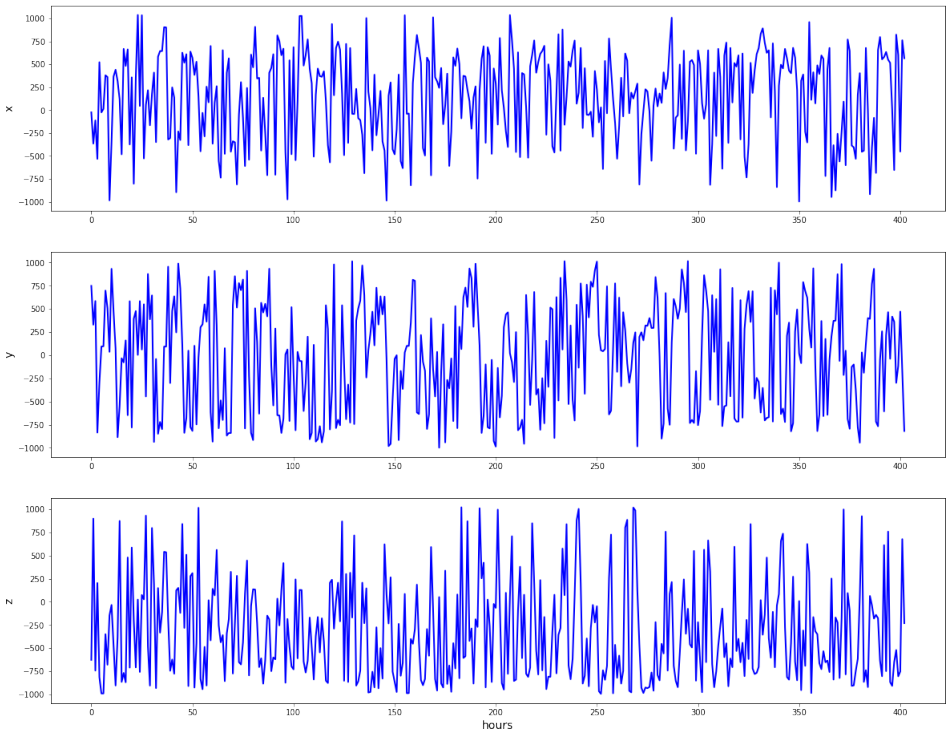

Figure 1

TODO caption figure

Let's take a look at the statistics of the dataset:

column	mean	std	min	25%	50%	75%	max
x	132.732967	491.697810	-1239.0	-291.0	214.0	539.0	1039.0
y	-34.800146	594.977813	-1019.0	-646.0	-14.0	466.0	1078.0
z	-307.588768	538.335654	-1001.0	-794.0	-416.0	83.0	1032.0

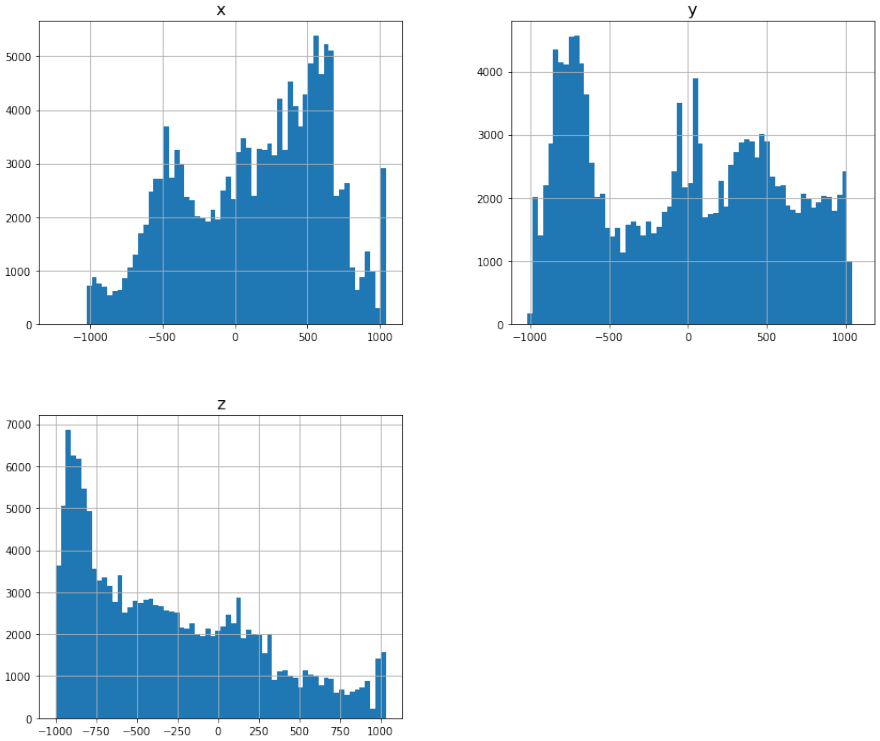

Figure 2

TODO caption figure

Data Preparation

Splitting

We'll use a 80% of the train data (train.csv) for training and remaining 20% for validation. The test data (test.csv) was set aside for testing the model.

```
import pandas as pd

train_data = pd.read_csv('train.csv', sep = ',')
test = pd.read_csv('test.csv', sep = ',')

n = len(train)

train = train_data[          : int(n*0.8)]
valid = train_data[int(n*0.8) :          ]
```

Scaling

We'll transform each feature individually such that it is in the given range on the training set, in our case between -1 and 1. This transformation is often used as an alternative to zero mean, unit variance scaling.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(-1,1))

train = scaler.fit_transform(train_df)
valid = scaler.transform(valid_df)
```

Windowing

Our model will make a set of predictions based on a window of consecutive samples from the data. The main features of the input windows are:

1. the window size (number of time steps) of the input and
2. the window shift.

For example, to make a single prediction 1 minute into the future, given 5 minutes of history, we might define a window like this:

Suppose that x has these values: x = [0, 1, 2, 3, 4]. Assume that the size of the window is 2 and the shift is 1. We create the following sequences: data = [[0, 1], [1, 2], [2, 3]], , label = [[2], [3], [4]], the goal is to predict for the sequence [0, 1] the value 2, for [1, 2] the value [3] and fot the sequence [2, 3] the value [4].

```
def split(data, win_size=30, win_shift=6, offset=1):

  rows = np.arange(win_size, len(data) - win_shift - 1, win_shift)

  samples = np.zeros((len(rows), win_size, data.shape[-1]))
  targets = np.zeros((len(rows),))

  for j, row in enumerate(rows):
    samples[j] = data[j * win_shift: row]
    targets[j] = data[row: row + offset]

  return samples, targets
```

Data Modeling

Time series prediction problems are a difficult type of predictive modeling problem. Unlike regression predictive modeling, time series also adds the complexity of a sequence dependence among the input

variables. A powerful type of neural network designed to handle sequence dependence is called Recurrent Neural Network (RNN).

When reading a sentence the human brain processes it word by word while keeping memories of what came before; this gives humans a fluid representation of the meaning conveyed by the sentence. Biological intelligence processes information incrementally while maintaining an internal model of what it's processing, built from past information and constantly updated as new information comes in.

A RNN adopts the same principle, albeit in an extremely simplified version: it processes sequences by iterating through the sequence elements and maintaining a state containing information relative to what it has seen so far. In effect, an RNN is a type of neural network that has an internal loop. The state of the RNN is reset between processing two different, independent sequences, so you still consider one sequence a single data point: a single input to the network. What changes is that this data point is no longer processed in a single step; rather, the network internally loops over network: a network with a loop sequence elements.

Susch RNNs have a major issue: although it should theoretically be able to retain at time t information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn. This is due to the vanishing gradient problem, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable. Long Short-Term Memory (LSTM) networks tackle this problem. LSTM network adds a way to carry information across many timesteps thus preventing older signals from gradually vanishing during processing. We will use this type of netkwork for modeling our data.

An important constructor argument for all Keras RNN layers, such as tf.keras.layers.LSTM, is the return_sequences argument. This setting can configure the layer in one of two ways:

1. If False, the default, the layer only returns the output of the final time step, giving the model time to warm up its internal state before making a single prediction:

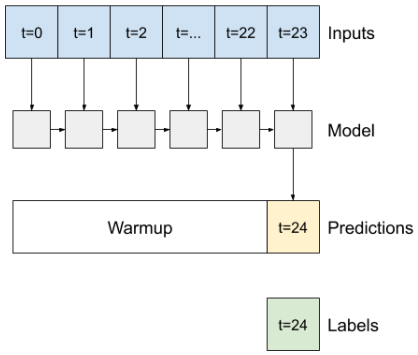

Figure 3

TODO caption figure

1. If True, the layer returns an output for each input. This is useful for:

a) Stacking RNN layers.

b) Training a model on multiple time steps simultaneously.

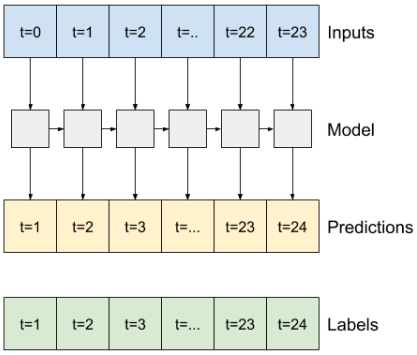

Figure 4

TODO caption figure

Here is our model.

```
lstm = tf.keras.models.Sequential([
        tf.keras.layers.LSTM(16,
                             input_shape=(seq_len, num_feat),
                             return_sequences=True),
        tf.keras.layers.LSTM(8),
        tf.keras.layers.Dense(1, activation='tanh')
])
```

```
lstm.compile(loss = 'mae', optimizer='adam')

history = lstm.fit(x_train, y_train, epochs=100,
                   validation_data=(x_valid, y_valid),
                   verbose = 1, batch_size=32, shuffle=False)
```

	x	y	z
MAE	54.55	59.22	61.74

Window (min)	MAE
1	65.14
2	55.27
3	63.99
4	55.63
5	60.96
6	55.34
7	63.05
8	58.61
9	55.33
10	54.85
11	54.47
12	55.03
13	58.72
14	54.69
15	64.59

Shift (min)	MAE
0.5	61.46
1	57.68
1.5	57.92
2	55.09
2.5	54.76
3	54.62
3.5	58.28
4	57.29
4.5	60.71
5	56.14
5.5	59.64
6	55.21
6.5	60.45
7	58.58
7.5	53.86

Anomaly detection

Data Understanding

The training set is composed by control patient, i.e. volunteers without Parkinson's Desease.

The train dataset contains 7 different features: identification of patient, accelerometer readings in the three axes (x, y, z), heart rate, date and timestamp. Data were collected every 10 seconds for a total of 943522 records and 13 volunteers. Missing value on heart rate attribute are labeled with -1.

Here are the first few rows of the train dataset:

No.	patient	x	y	z	heartRate	timestamp	tsDate
0	1502	23	569	878	-1	1568073600000	2019-09-10 00:00:00.003
1	1502	23	571	878	-1	1568073601000	2019-09-10 00:00:01.014
2	1502	23	570	878	-1	1568073602000	2019-09-10 00:00:02.025
3	1502	23	570	878	-1	1568073603000	2019-09-10 00:00:03.035
4	1502	23	570	878	-1	1568073604000	2019-09-10 00:00:04.046
5	1502	23	570	879	-1	1568073605000	2019-09-10 00:00:05.057
6	1502	23	569	879	-1	1568073606000	2019-09-10 00:00:06.066
7	1502	22	570	879	-1	1568073607000	2019-09-10 00:00:07.078
8	1502	23	570	879	-1	1568073608000	2019-09-10 00:00:08.088
9	1502	24	570	878	-1	1568073609000	2019-09-10 00:00:09.099

Here is the time evolution of the three components of acceleration and heart rate over time for one volunteer:

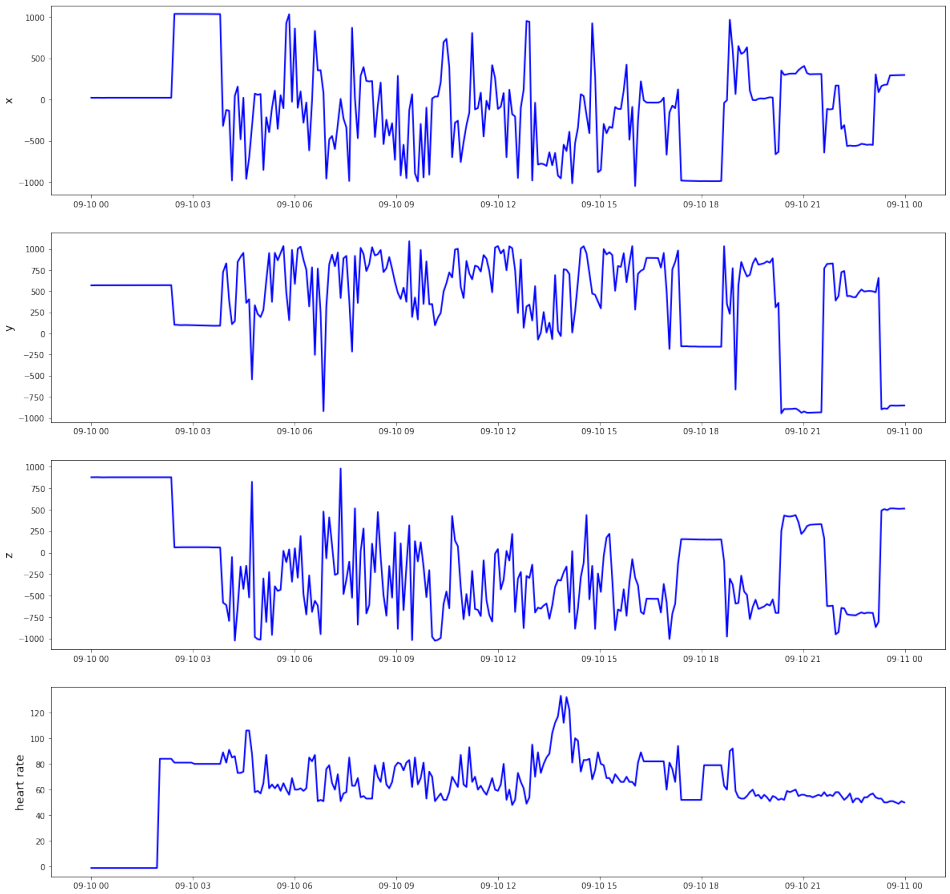

Figure 5

TODO caption figure

Let's take a look at the statistics of the dataset:

column	mean	std	min	25%	50%	75%	max
x	-82.99378	577.0332	-1649.000	-532.0000	-91.00000	311.0000	1796.000
y	265.3513	539.7477	-1082.000	-59.00000	313.0000	7.390000	1644.000
z	-203.9305	534.9554	-1256.0003	-665.0000	-223.0000	125.0000	1127.000
heartRate	70.20357	20.97376	-1.000000	61.000001	70.00000	82.00000	182.0000

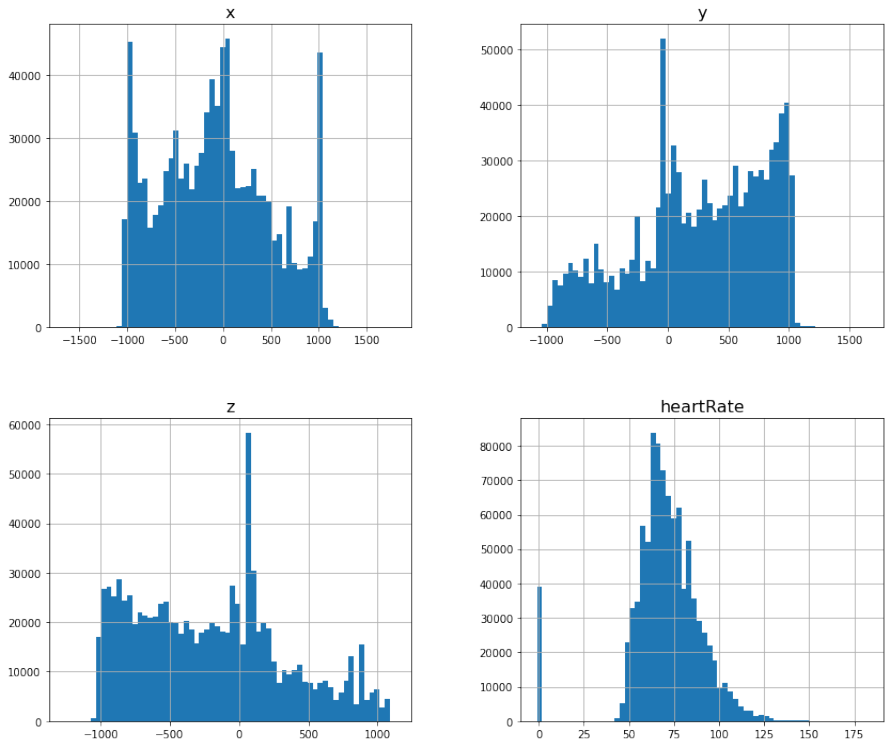

Figure 6

TODO caption figure

Data Preparation

First, we replace missing values on heart rate attribute (labeled with -1) with median.

```
import pandas as pd

trn_df = pd.read_csv('ad_train.csv', sep = ',')
tst_df = pd.read_csv('ad_test.csv', sep = ',')

bad_hr = trn_df['heartRate'] == -1
trn_df['heartRate'][bad_hr] = np.median(trn_df['heartRate'])
```

Select only relevant attributes meaning the three components of the acceleration and heart rate and sub-sample the data from 1 second interval to 10 second interval.

Splitting

As before we'll use a 80% of the train data (ad_train.csv) for training and remaining 20% for validation. The test data (ad_test.csv) was set aside for testing the model.

```
n = len(train_df)

train_data = train_df[           : int(n*0.8)].to_numpy()
valid_data = train_df[int(n*0.8) :           ].to_numpy()
test_data  = tst_df[['x', 'y', 'z', 'heartRate']].to_numpy()
```

Scaling

We'll transform each feature individually such that it is in the given range on the training set, in our case between 0 and 1.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

train = scaler.fit_transform(train_data.reshape(-1, 1))
valid = scaler.transform(valid_data.reshape(-1, 1))
test = scaler.transform(test_data.reshape(-1, 1))
```

Data Modeling

In anomaly detection, the goal is to find objects that do not conform to normal patterns or behavior. Often, anomalous objects are known as **outliers**, since, on a scatter plot of the data, they lie far away from other data points. Anomaly detection is also known as **deviation detection**, because anomalous objects have attribute values that deviate significantly from the expected or typical attribute values, or as textbfexception mining, because anomalies are exceptional in some sense. There are a variety of anomaly detection approaches from several areas, including statistics, machine learning, and data mining. All try to capture the idea that an anomalous data object is unusual or in some way inconsistent with other objects.

Although unusual objects or events are, by definition, relatively rare, their detection and analysis provides critical insights that are useful in a number of applications for instance: Medicine and Public Health. For a particular patient, unusual symptoms or test results may indicate potential health problems. However, whether a particular test result is anomalous may depend on many other characteristics of the patient, such as age, sex, and genetic makeup. Furthermore, the categorization of a result as anomalous or not incurs a cost—unneeded additional tests if a patient is healthy and potential harm to the patient if a condition is left undiagnosed and untreated. The detection of emerging disease outbreaks which result in unusual and alarming test results in a series of patients, is also important for monitoring the spread of diseases and taking preventive actions.

The nature of the input data plays a key role in deciding the choice of a suitable anomaly detection technique.

If the data contains a single attribute, the question of whether an object is anomalous depends on whether the object's value for that attribute is anomalous. However, if the data objects are represented using many attributes, a data object may have anomalous values for some attributes but ordinary values for other attributes. Furthermore, an object may be anomalous even if none of its attribute values are

individually anomalous. Identifying an anomaly in a multivariate setting is thus challenging, particularly when the dimensionality of the data is high.

Since anomalies are infrequent, most input data sets have a predominance of normal instances. The input data set is thus often used as an imperfect representation of the normal class in most anomaly detection techniques. However, the performance of such methods needs to be robust to the presence of outliers in the input data.

Anomalies, unlike normal objects, are often unrelated to each other and hence distributed sparsely in the space of attributes. Indeed, the successful operation of most anomaly detection methods depends on anomalies not being tightly clustered.

Reconstruction-based Approaches

Reconstruction-based techniques rely on the assumption that the normal class resides in a space of lower dimensionality than the original space of attributes. In other words, there are patterns in the distribution of the normal class that can be captured using lower-dimensional representations, e.g., by using dimensionality reduction techniques.

To illustrate this, consider a data set of normal instances, where every instance is represented using p continuous attributes, x_1, ... , x_p . If there is a hidden structure in the normal class, we can expect to approximate this data using fewer than p derived features.

One common approach for deriving useful features from a data set is to use autoencoders. An autoencoder (also referred to as an autoassociator or a mirroring net- work) is a multi-layer neural network, where the number of input and output neurons is equal to the number of original attributes. The general architecture of an autoencoder involves two basic steps: encoding and decoding. During encoding, a data instance $\mathbf{x}$ is transformed to a low-dimensional representation $\mathbf{y}$, using a number of nonlinear transformations in the encoding layers. The number of neurons reduces at every encoding layer, so as to learn low-dimensional representations from the original data. The learned representation $\mathbf{y}$ is then mapped back to the original space of attributes using the decoding layers, resulting in a reconstruction of $\mathbf{x}$ ($\hat{\mathbf{x}}$). The distance between $\mathbf{x}$ and $\hat{\mathbf{x}}$ (the reconstruction error) is then used as a measure of an anomaly score.

The autoencoder scheme provides a powerful approach for learning complex and nonlinear representations of the normal class.

TODO mejorar el paso del parrafo anterior al modelo

Here is our model.

```
encoder_inputs =
        tf.keras.layers.Input(shape=(sequence_lenght, number_of_features))

encoder_l1 = tf.keras.layers.LSTM(16, return_state=True, activation='tanh')
encoder_outputs1 = encoder_l1(encoder_inputs)
encoder_states1 = encoder_outputs1[1:]

decoder_inputs =
        tf.keras.layers.RepeatVector(sequence_lenght)(encoder_outputs1[0])

decoder_l1 =
        tf.keras.layers.LSTM(16, return_sequences=True, activation='tanh')
```

```
        (decoder_inputs, initial_state = encoder_states1)

decoder_outputs1 =
        tf.keras.layers.TimeDistributed(tf.keras.layers.Dense(number_of_features))
        (decoder_l1)

autoencoder = tf.keras.models.Model(encoder_inputs,decoder_outputs1)

autoencoder.summary()
```

```
reduce_lr = tf.keras.callbacks.LearningRateScheduler(lambda x: 1e-3 * 0.90 ** x)

autoencoder.compile(loss = 'mae', optimizer='adam')
history = autoencoder.fit(train, train, epochs=60,
                validation_data=(valid, valid), verbose=1,
                batch_size=32, shuffle=False, callbacks=[reduce_lr])
```

Detect Anomaly

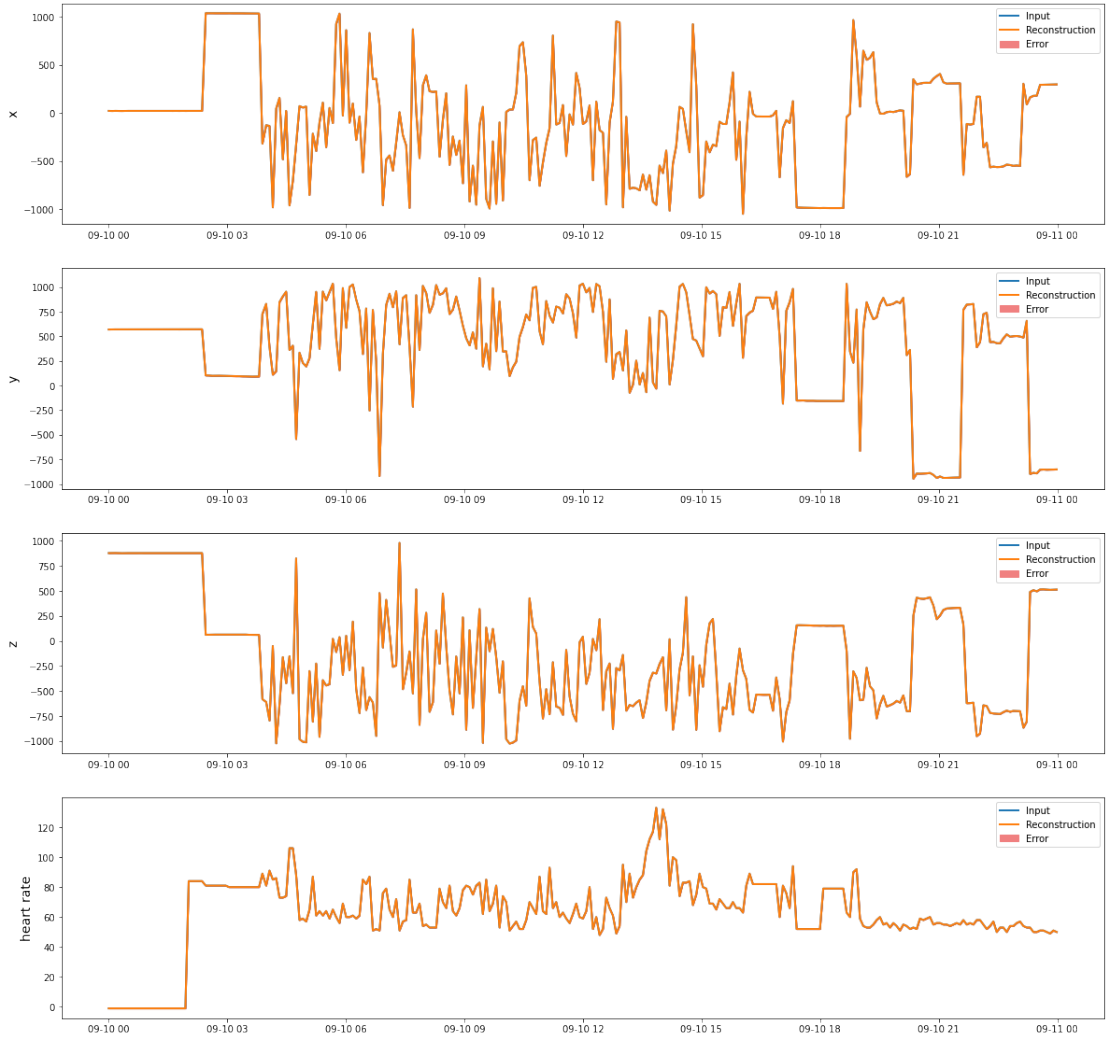

Figure 7

TODO caption figure

We'll detect anomalies by calculating whether the reconstruction loss is greater than a fixed threshold.

We choose a threshold value then classify future examples as anomalous if the reconstruction error is higher than the threshold.

We will use three different thresholds values: * calculate the mean and standard deviation of the reconstruction error of the training set, then classify future examples as anomalous if the reconstruction error is higher than one standard deviation from the training set. * calculte the third quartile, then classify future examples as anomalous if the reconstruction error is higher than this value * classify future examples as anomalous if the reconstruction error is higher than 0.003

Plot the reconstruction error on the train data.

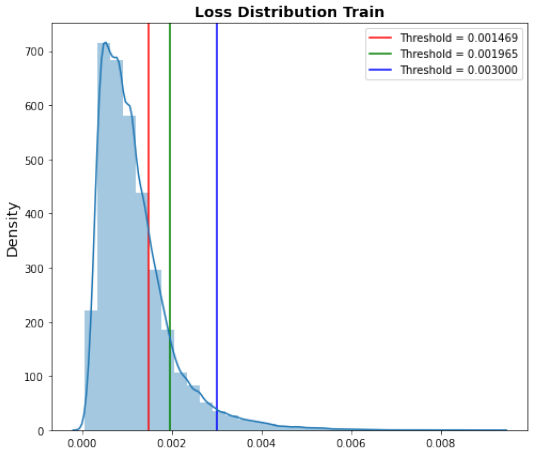

Figure 8

TODO caption figure

Plot the reconstruction error on the test data.

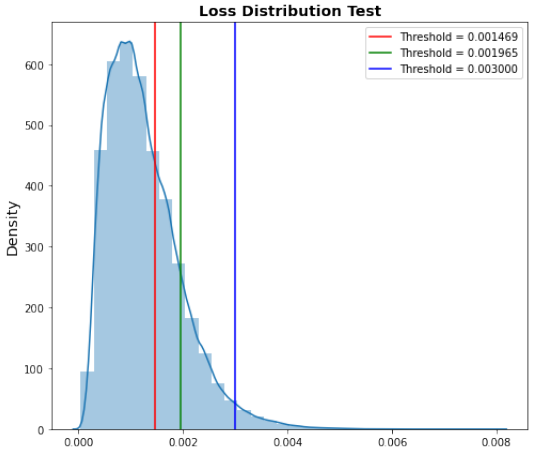

Figure 9

TODO caption figure

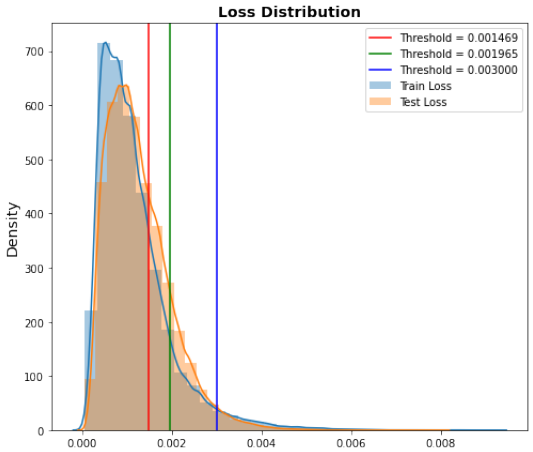

Figure 10

TODO caption figure

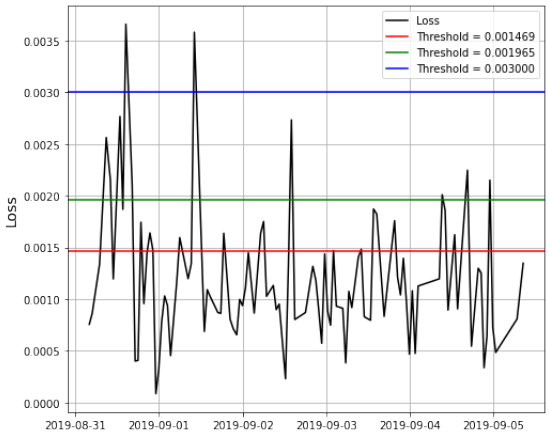

Figure 11

TODO caption figure

Figure 12

TODO caption figure