Deep Learning Project

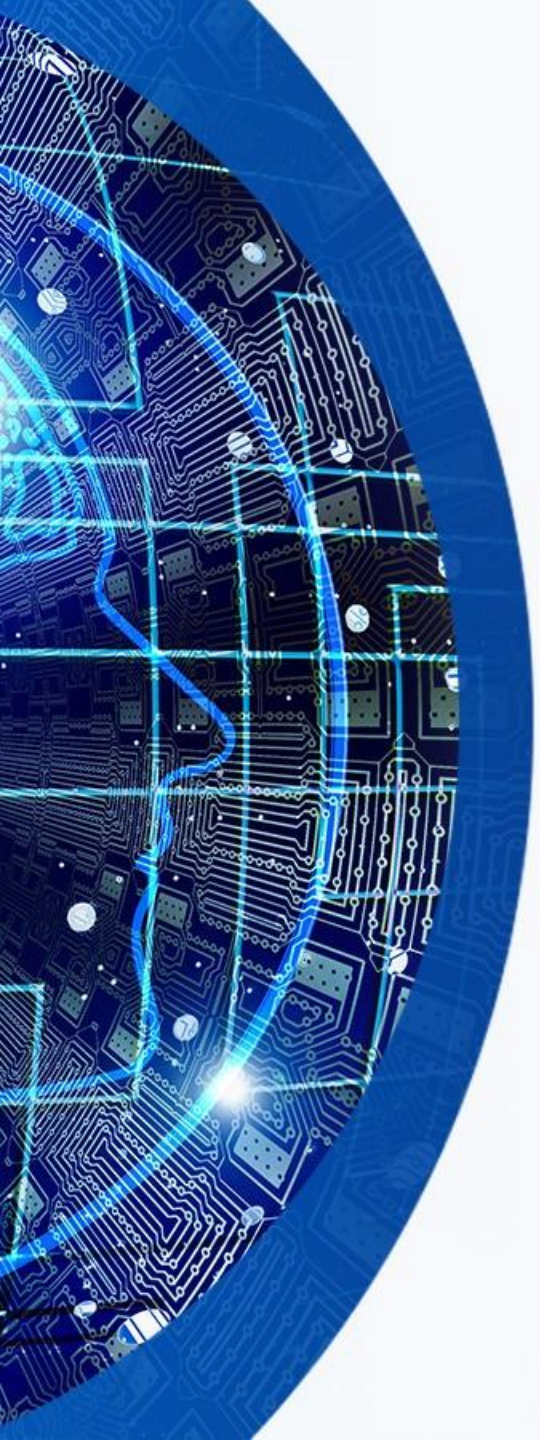
Gruppo **brAIns**

Marco Bellizzi 223966
Domenico Spagnolo 227805
Giuseppe Oliva 223956

TABLE OF CONTENTS

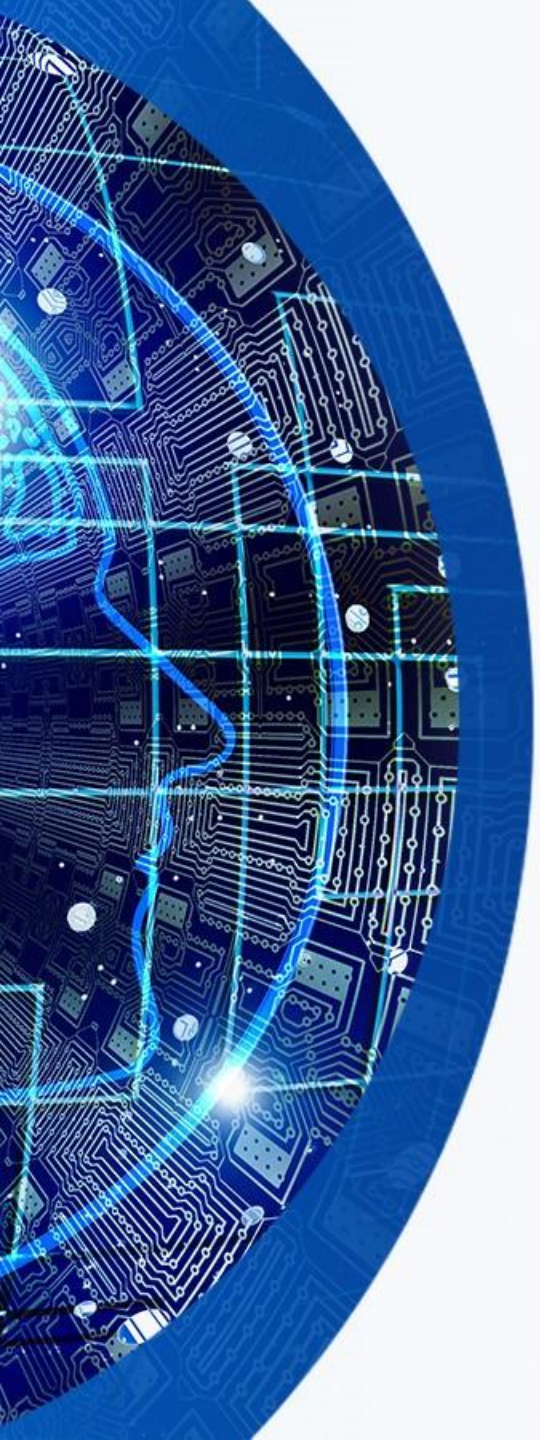
TASK 1.1:
Next value prediction

- Introduction
- Data understanding and preparation
- Modeling
- Evaluation
- Conclusions



Introduction

We have a dataset containing 3 time series X, Y and Z with a time interval of 10 seconds. For each time series we want to predict the next values of subsequences of 5 minutes with a step size of 1 minute.



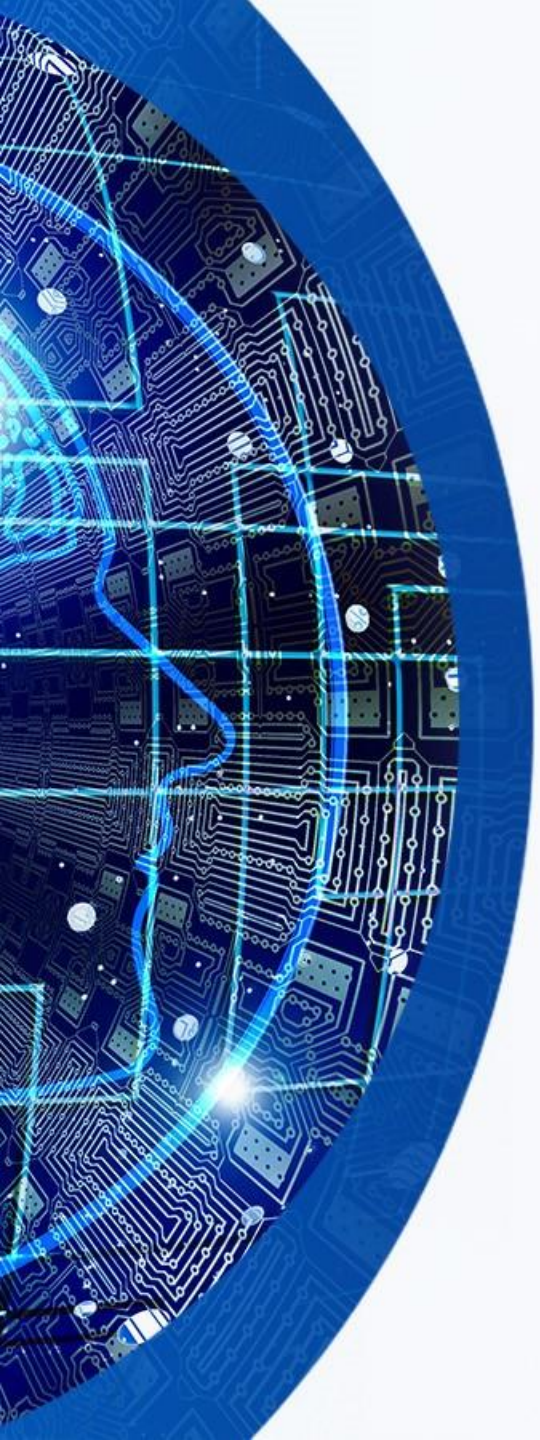
Data understanding and preparation

- We loaded the training and the test set

```
test = pd.read_csv(os.path.join('DeMaCS_Project_Data/task1_next_value_prediction', 'test.csv'), sep = ',')
train = pd.read_csv(os.path.join('DeMaCS_Project_Data/task1_next_value_prediction', 'train.csv'), sep = ',')
```

- We normalized the training set

```
train_mean = train.mean()
train_std = train.std()

train = (train - train_mean) / train_std
train
```



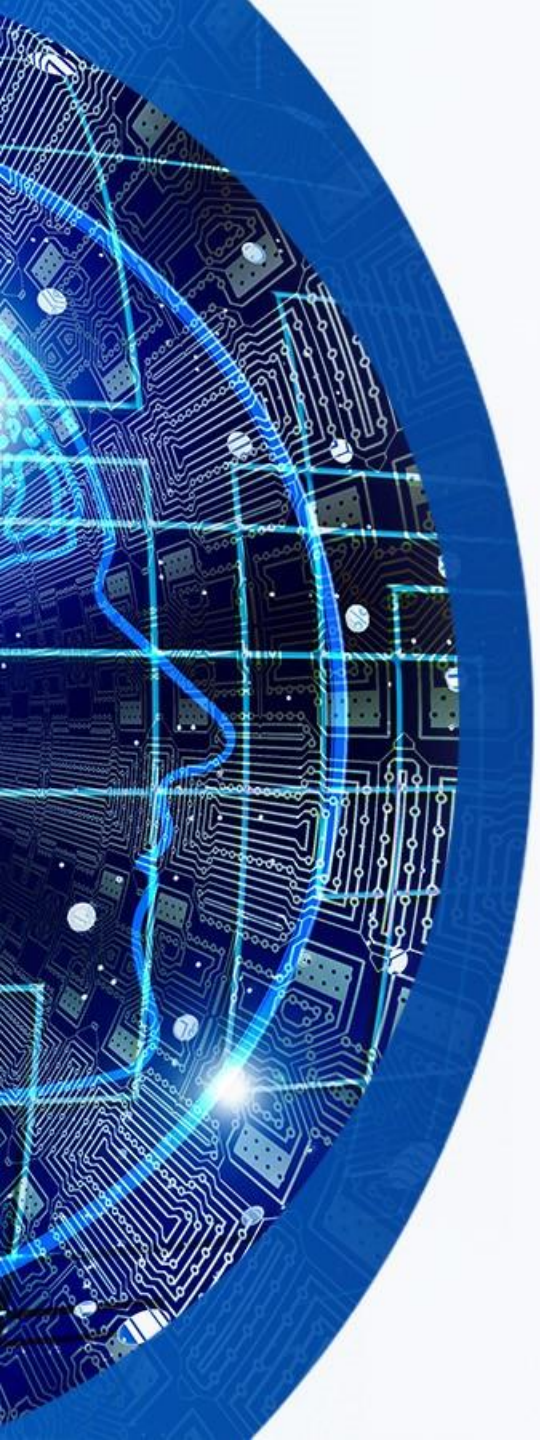
Data understanding and preparation

- We have created a function that given a dataset, the window size, the window shift and the axis of the time series extracts the subsequences with only one dimension (30) and collects the relative next values, that are the values we want to predict.

- Then, we used it for extracting the training sequences and the test sequences for each axis. The extraction was done in both datasets with window size = 30 and window shift = 6

```
def extract_seq(data_set, max_len, step, axis):
  print("extracting sequences")
  sequences = []
  values = []
  for i in range(0, len(data_set) - max_len - 1, step):
    print("\r" + str(int((i+1) / len(data_set) * 100)) + " %", end="")
    sequences.append(data_set[axis][i: i + max_len])
    values.append(data_set[axis][i + max_len])
  sequences = np.array(sequences)
  sequences = sequences.reshape((sequences.shape[0], sequences.shape[1], 1))
  values = np.array(values)
  values = values.reshape((values.shape[0], 1))
  print("\r100 %")
  return sequences, values
```

```
sequences_train_x, values_train_x = extract_seq(train, 30, 6, 'x')
sequences_train_y, values_train_y = extract_seq(train, 30, 6, 'y')
sequences_train_z, values_train_z = extract_seq(train, 30, 6, 'z')

sequences_test_x, values_test_x = extract_seq(test, 30, 6, 'x')
sequences_test_y, values_test_y = extract_seq(test, 30, 6, 'y')
sequences_test_z, values_test_z = extract_seq(test, 30, 6, 'z')
```



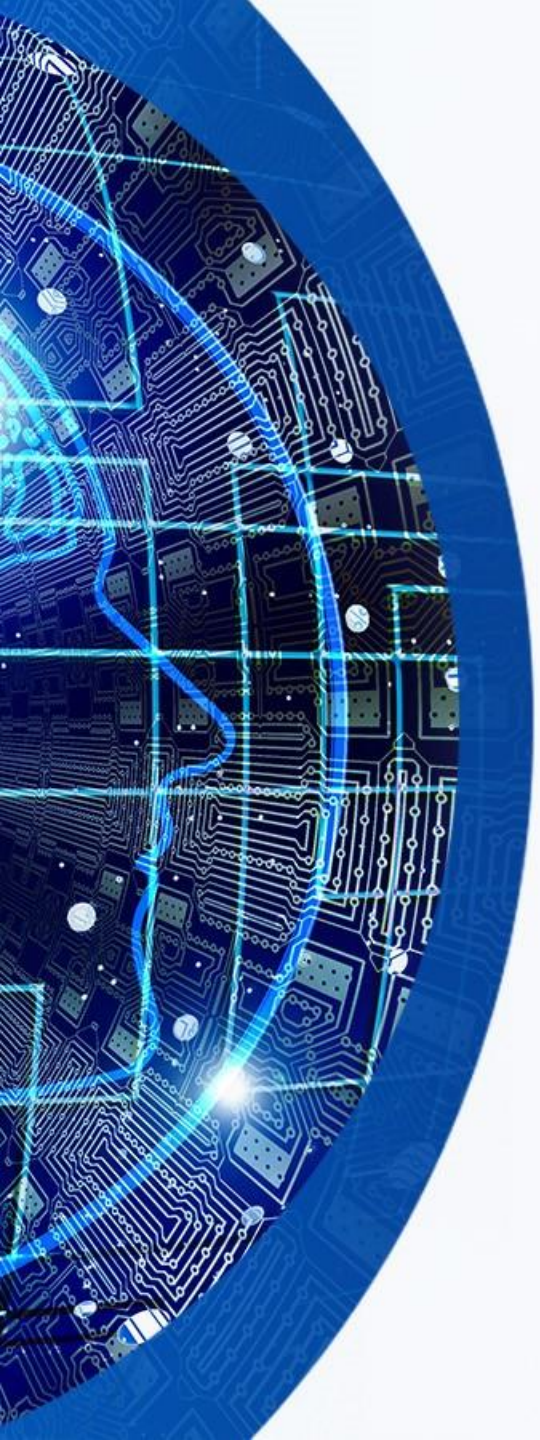
Modeling

Our models are composed by a LSTM layer with the same input shape as sequences and a Dense layer with one node without activation function to make the prediction of one value. As loss function we used the mean absolute error. We used the training sequences to fit the models and the test sequences to evaluate it.

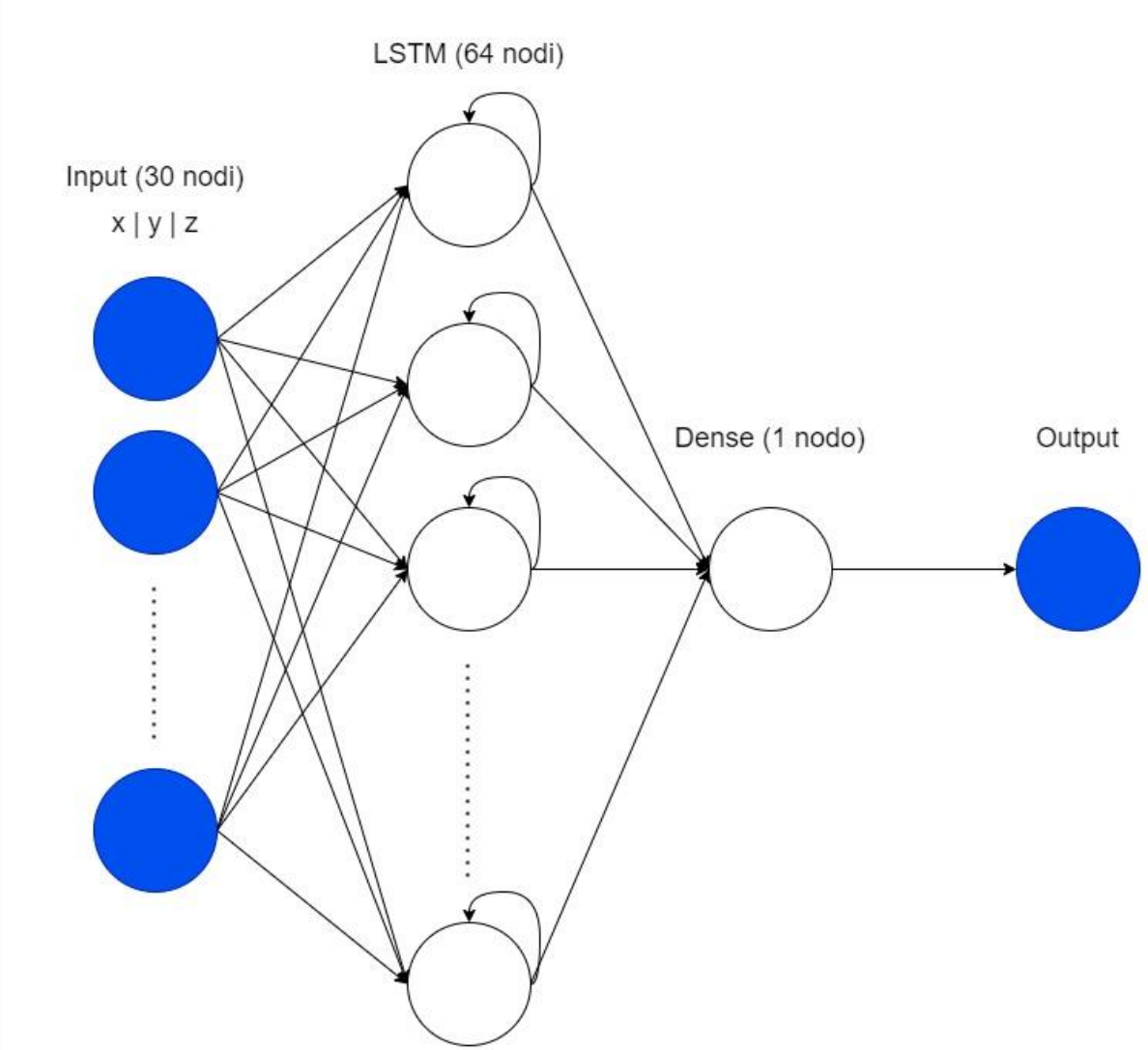
```
model_x = Sequential()
model_x.add(layers.LSTM(64, input_shape=(sequences_train_x.shape[1], sequences_train_x.shape[2])))
model_x.add(layers.Dense(1))
model_x.compile(loss='mean_absolute_error', optimizer = 'adam')
history_x = model_x.fit(sequences_train_x, values_train_x, epochs=5)
```

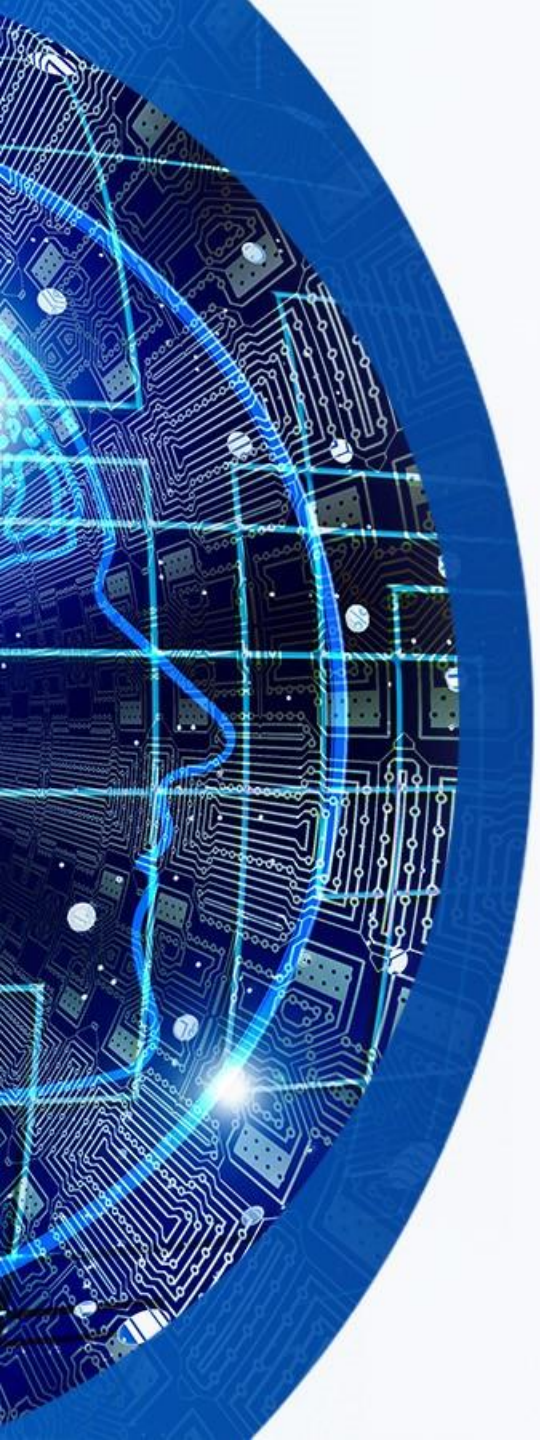
```
model_y = Sequential()
model_y.add(layers.LSTM(64, input_shape=(sequences_train_x.shape[1], sequences_train_x.shape[2])))
model_y.add(layers.Dense(1))
model_y.compile(loss='mean_absolute_error', optimizer = 'adam')
history_y = model_y.fit(sequences_train_y, values_train_y, epochs=5)
```

```
model_z = Sequential()
model_z.add(layers.LSTM(64, input_shape=(sequences_train_x.shape[1], sequences_train_x.shape[2])))
model_z.add(layers.Dense(1))
model_z.compile(loss='mean_absolute_error', optimizer = 'adam')
history_z = model_z.fit(sequences_train_z, values_train_z, epochs=5)
```



Modeling



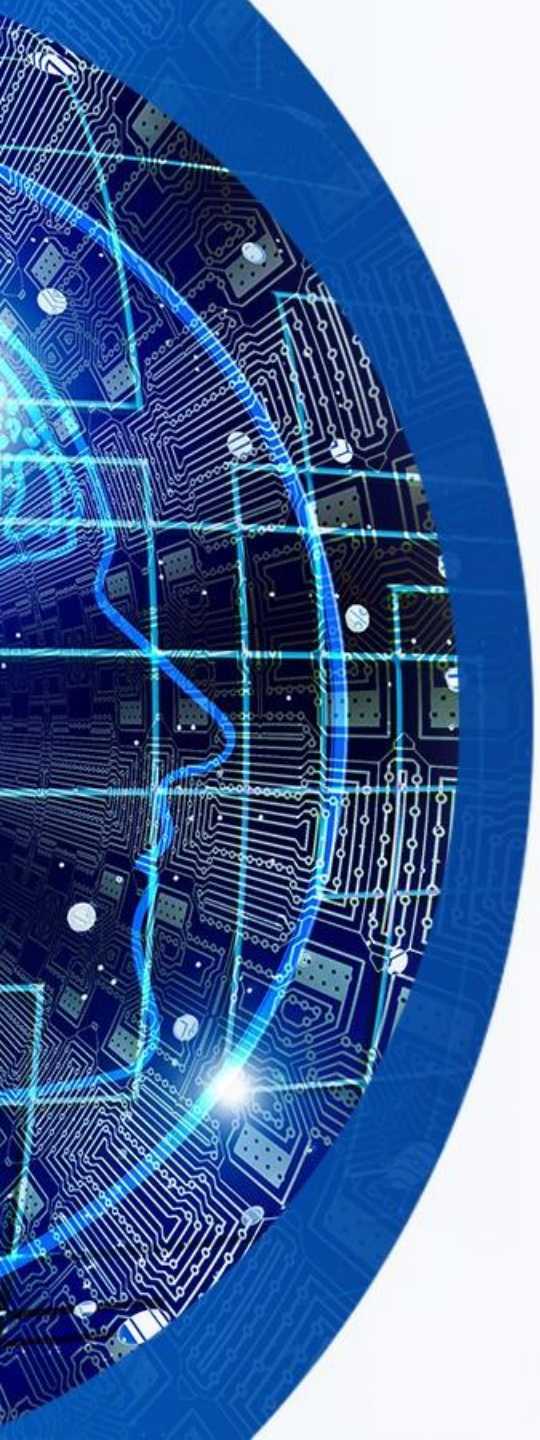


Evaluation

We evaluated the models with the test sequences of each axis. For each sequences we used the model to make the prediction of the sequence. We have normalized the values of the sequences, made a prediction on each sequence and than renormalized the predictions. Finally, we calculated the mean absolute error of the predictions. To do this, the `get_error()` function has been defined

```
def get_error(sequences, values, axis, model):
  print("getting error")
  error = 0
  for i in range(sequences.shape[0]):
    print("\r" + str(int((i+1) / sequences.shape[0] * 100)) + " %", end="")
    prediction = model.predict((sequences[[i]] - train_mean[axis]) / train_std[axis])
    prediction = (prediction * train_std[axis]) + train_mean[axis]
    error += abs(prediction[0][0] - values[i][0])
  print("\r100 %")
  return error / sequences.shape[0]
```

```
# x=81.06; y=85.26; z=79.94
print("errore sull' asse x = " + str(get_error(sequences_test_x, values_test_x, 'x', model_x)))
print("errore sull' asse y = " + str(get_error(sequences_test_y, values_test_y, 'y', model_y)))
print("errore sull' asse z = " + str(get_error(sequences_test_z, values_test_z, 'z', model_z)))
```



Conclusions

In conclusion, we can be satisfied with the results obtained since the errors obtained on the three axes are below the maximum error thresholds provided.

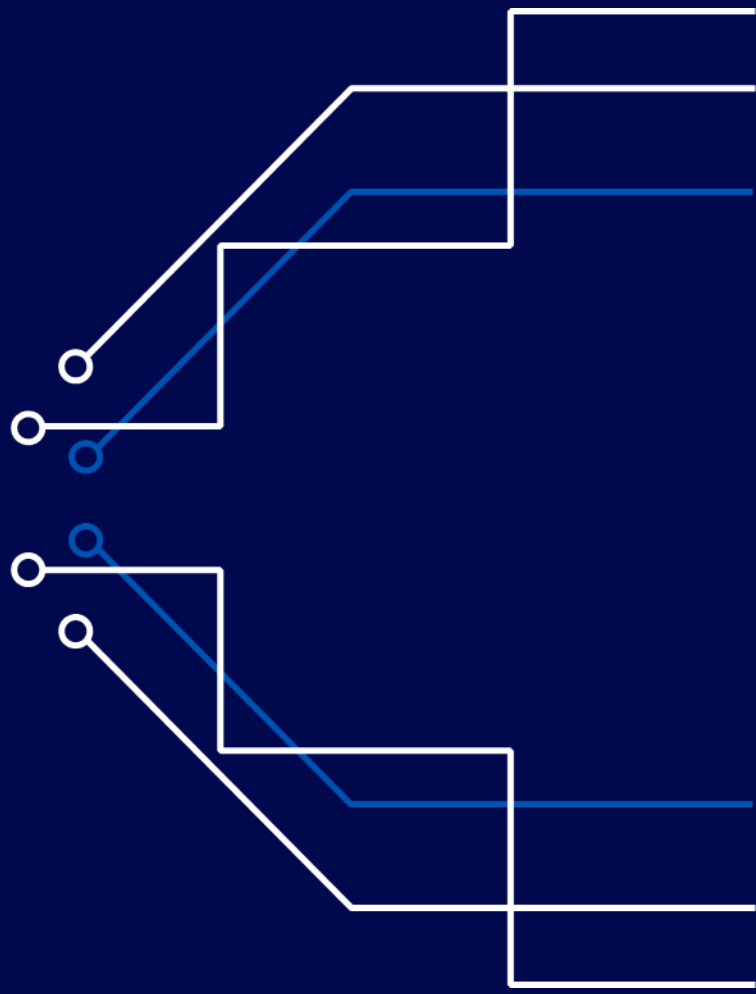
(x = 75.72; y = 81.06; z = 73.73) errors obtained

(x = 81.06; y = 85.26; z = 79.94) errors thresholds

```
getting error
100 %
errore sull' asse x = 75.72323637960645
getting error
100 %
errore sull' asse y = 81.05911578277306
getting error
100 %
errore sull' asse z = 73.73348487640465
```

TASK 1.2

- Analysis of the effects of changes of window size and window shift



Analysis of the effects of changes of window size and window shift

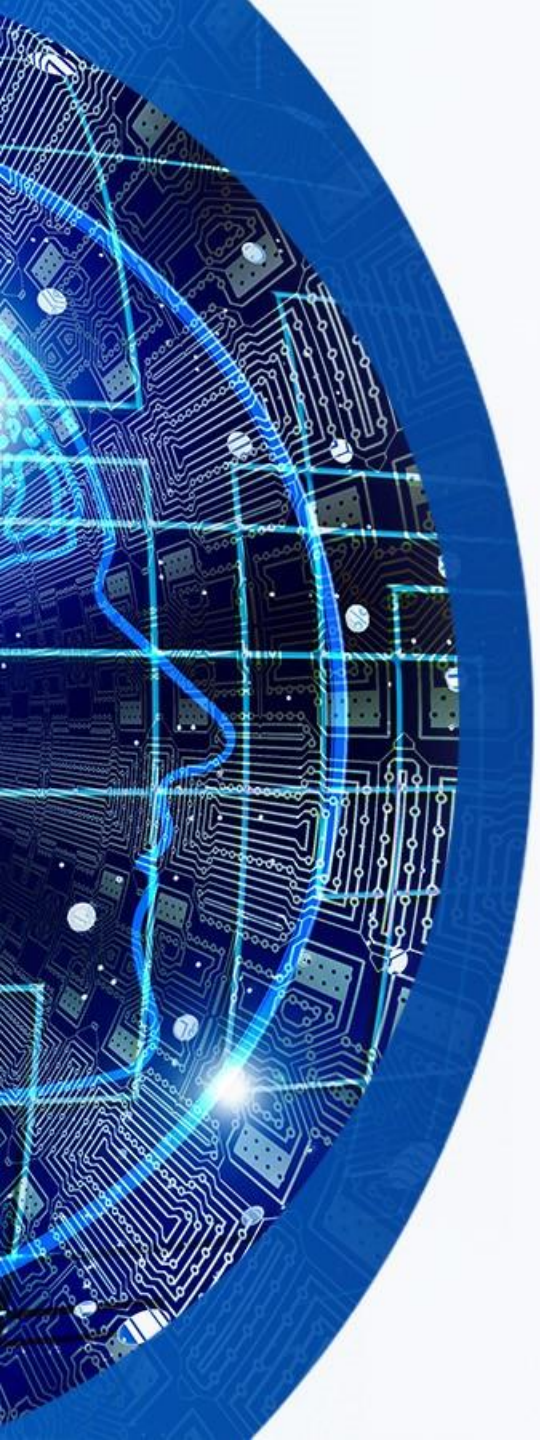
Similarly, as what we did before, we calculated the mean absolute error of the predictions changing the values of window size and window shift for extracting the sequences and than we plotted the results. We chose to perform this operation only on the x time series.

```
labels = []
errors = []

for step_ in [3, 6, 12]:
  for maxlen_ in [15, 30, 60]:
    print("\nstep " + str(step_) + " - maxlen " + str(maxlen_) + "\n")

    sequences_train, values_train = extract_seq(train, maxlen_, step_, 'x')
    sequences_test, values_test = extract_seq(test, maxlen_, step_, 'x')

    model = Sequential()
    model.add(layers.LSTM(64, input_shape=(sequences_train.shape[1], sequences_train.shape[2])))
    model.add(layers.Dense(1))
    model.compile(loss='mean_absolute_error', optimizer = 'adam')
    model.fit(sequences_train, values_train, epochs=5)

    labels.append(str(step_) + " " + str(maxlen_))
    errors.append(get_error(sequences_test, values_test, 'x', model))
```

The plot shows that the best choices for window shift and window size are respectively 12 and 30. With these parameters, in fact, there is the lowest error.

TASK 2:
Anomaly detection

- Introduction
- Data understanding and preparation
- Modeling
- Evaluation
- Conclusions



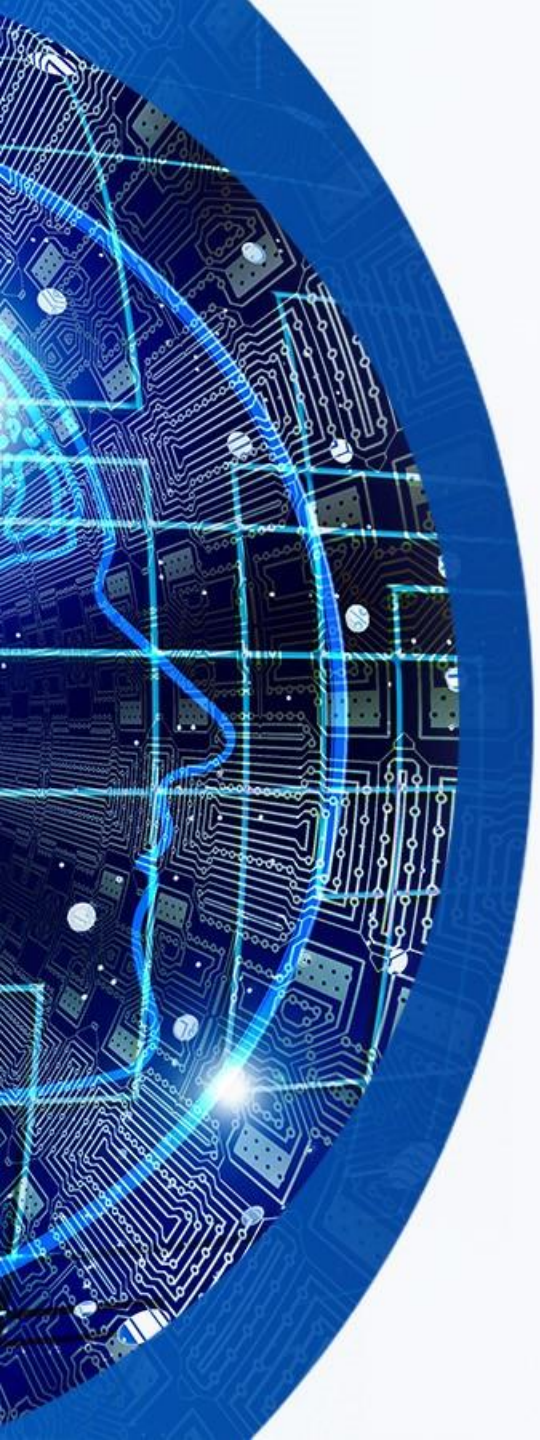
Introduction

The goal of this task is to detect off period in patients affected by Parkinson desease. An off period is a period in which the patient has tremors and corrisponds as an anomaly on the data. The data are time series with the following informations:

- **patient**: patient identification code;
- **timestamp**: timestamp of the survey;
- **tsDate**: date and time of the survey;
- **x, y, z**: accelerometer variables;
- **heartRate**: heart rate.

The test set has a time interval of 10 seconds, while the training set has a time interval of 1 second and contains also some missing values on heart rate attribute, so we need to resample the training set by mean of 10 seconds and dropping the records containing missing values.

For identify anomalies we use an autoencoder that try to replicate sequences of records. The sequence is considered anomaly if has an high error, so the autoencoder wasn't be able to predict it well. We have calculated the error of a sequence as sum of absolute error of each attribute of each records in that sequence with the predicted ones. Then, for each patient we identified a threshold and we considered anomaly the sequences with an higher error than its relative threshold. Finally, we reported the anomaly records in a csv file.



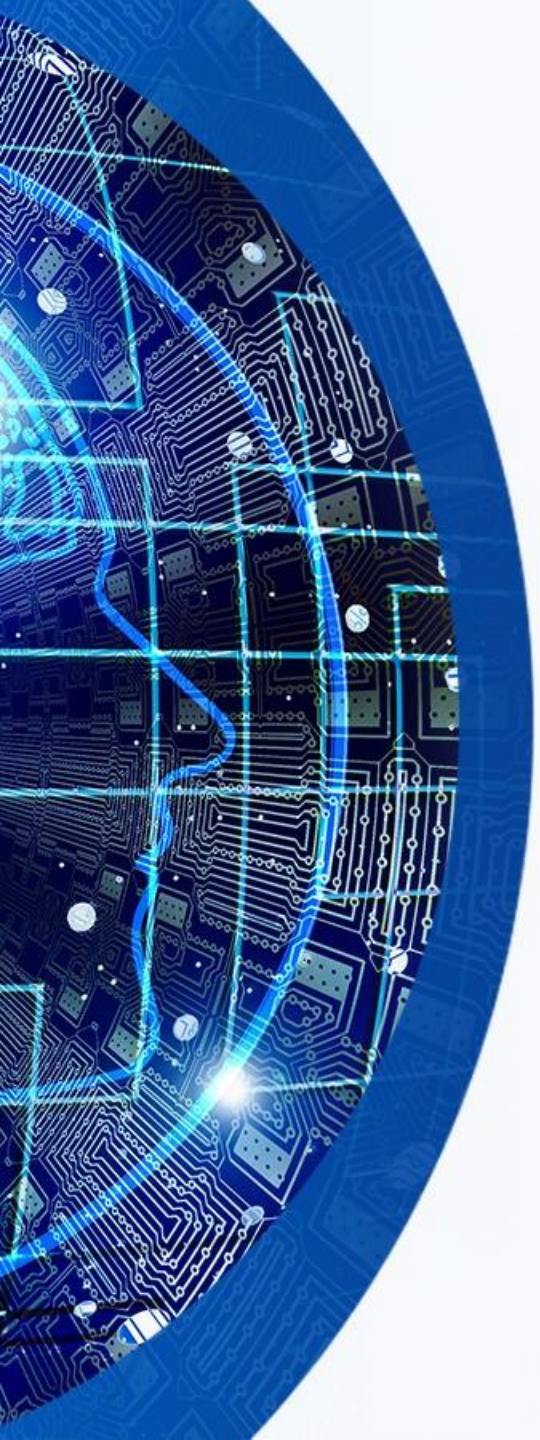
Data understanding and preparation - Training set

- Resampling by mean of 10 seconds

```
train_ad_sampled = []
cont = 0
cont_not_null = 0
x = 0
y = 0
z = 0
heart = 0

for line in train_ad:
    cont += 1
    values = line.split(",")

    if not cont == 1 and not int(values[4]) == -1:
        cont_not_null += 1
        x += int(values[1])
        y += int(values[2])
        z += int(values[3])
        heart += int(values[4])

    if cont % 10 == 0:
        if cont_not_null > 0:
            train_ad_sampled.append([str(x / cont_not_null), str(y / cont_not_null), str(z / cont_not_null), str(heart / cont_not_null)])
        x = 0
        y = 0
        z = 0
        heart = 0
        cont_not_null = 0
```

- Normalization

```
train_ad_sampled_mean = train_ad_sampled.mean()
train_ad_sampled_std = train_ad_sampled.std()

train_ad_sampled = (train_ad_sampled - train_ad_sampled_mean) / train_ad_sampled_std
train_ad_sampled
```



Data understanding and preparation – Test set

- We loaded test set

```
test_ad = pd.read_csv(os.path.join('DeMaCS_Project_Data/task2_anomaly_detection', 'ad_test.csv'), sep = ',')
test_ad
```

- We splitted it grouping by **patient** and we updated indexes (start from 0) of each subset

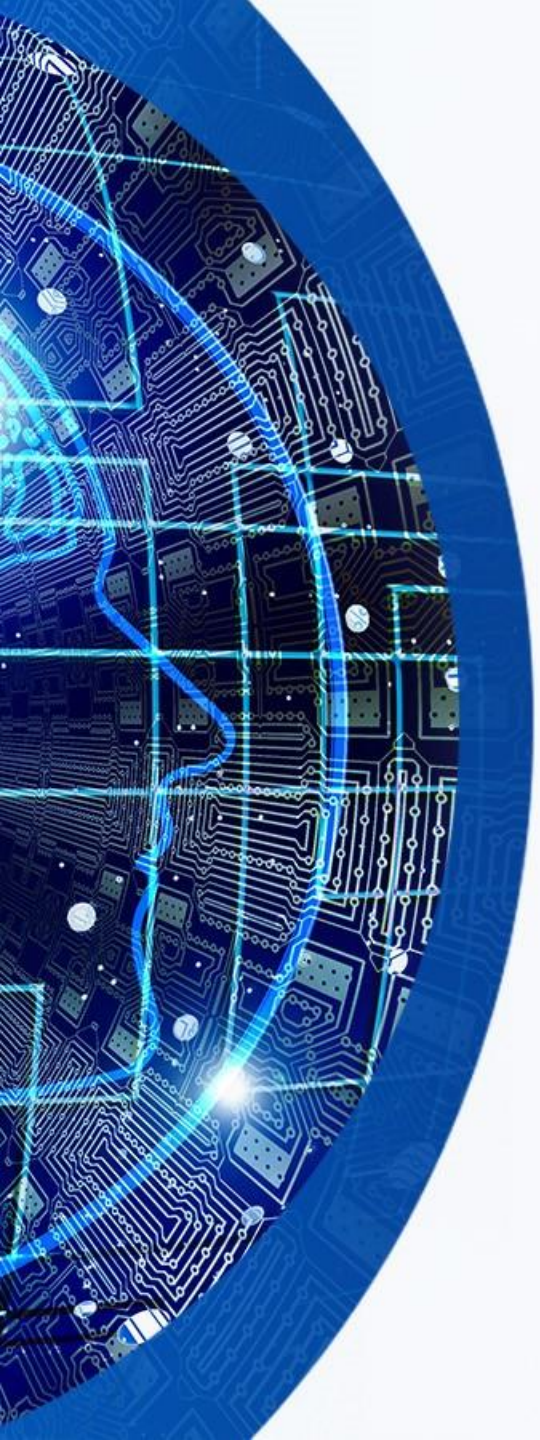
```
df_list = [d for _, d in test_ad.groupby(['patient'])]
# update indexes
for d in df_list:
    d.index = pd.RangeIndex(len(d.index))
```

- Copy of the datasets

```
# copy the datasets
df_copy = df_list
```

- Removing **patient**, **timestamp**, **tsdate** and normalization of each subset

```
test_ad_norm_list = []
for d in df_list:
  test_ad_norm = d.drop(['patient', 'timestamp', 'tsDate'], 1)
  test_ad_norm = (test_ad_norm - train_ad_sampled_mean) / train_ad_sampled_std
  test_ad_norm_list.append(test_ad_norm)
```

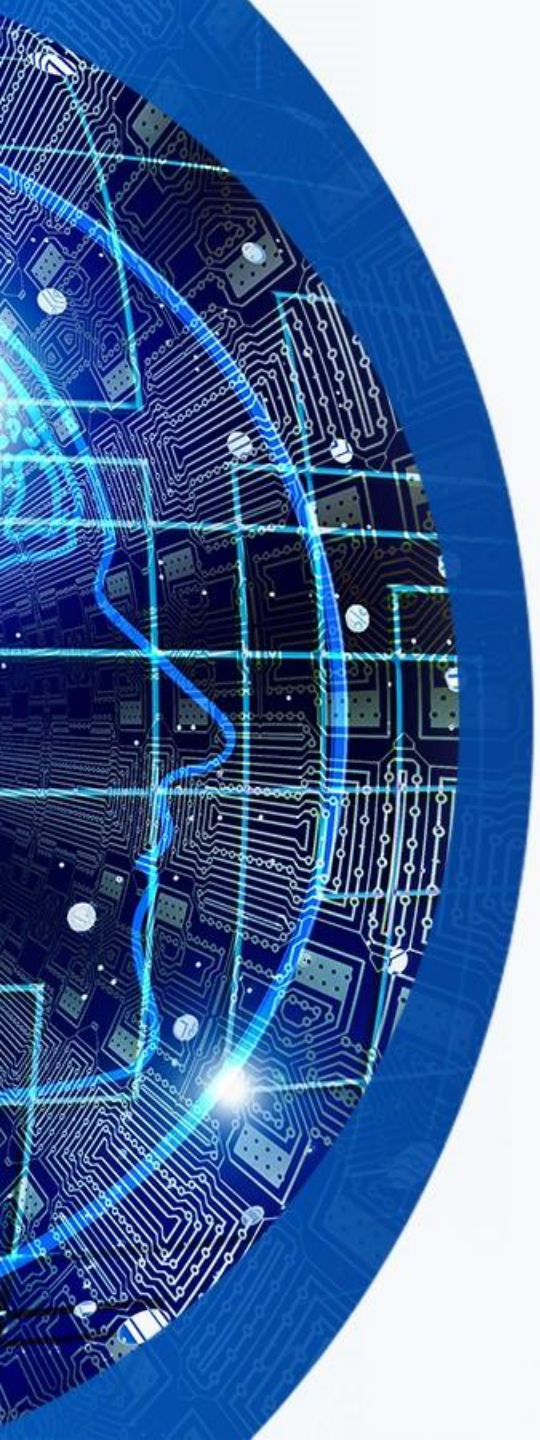


Data understanding and preparation - Sequences

Finally, the time series were extracted: differently from the task 1, in which the input consists of sequences containing only the values of an axis, in this case we have created a function that given a dataset, the window size and the window shift extracts the sequences. The sequences have 2 dimensions: the first is the length of the subsequences (30) and the second is the number of attribute we want to add (4). Then, we used it for extracting the training sequences and the test sequences of each patient.

```
def extract_seq_ad(data_set, maxlen, step):
  print("extracting sequences")
  sequences = []
  #for i in range(0, 1000, step):
  for i in range(0, len(data_set) - maxlen, step):
    print("\r" + str(int((i+1) / len(data_set) * 100)) + " %", end="")
    sequence = []
    for j in range(0, maxlen):
      entry = []
      entry.append(data_set['x'][i+j])
      entry.append(data_set['y'][i+j])
      entry.append(data_set['z'][i+j])
      entry.append(data_set['heartRate'][i+j])
      sequence.append(entry)

    sequences.append(sequence)

  sequences = np.array(sequences)
  print("\r100 %")
  return sequences
```

```
sequences_ad = extract_seq_ad(train_ad_sampled, 30, 1)

sequences_ad_test = []
for d in test_ad_norm_list:
    sequences_ad_test.append(extract_seq_ad(d, 30, 1))
```



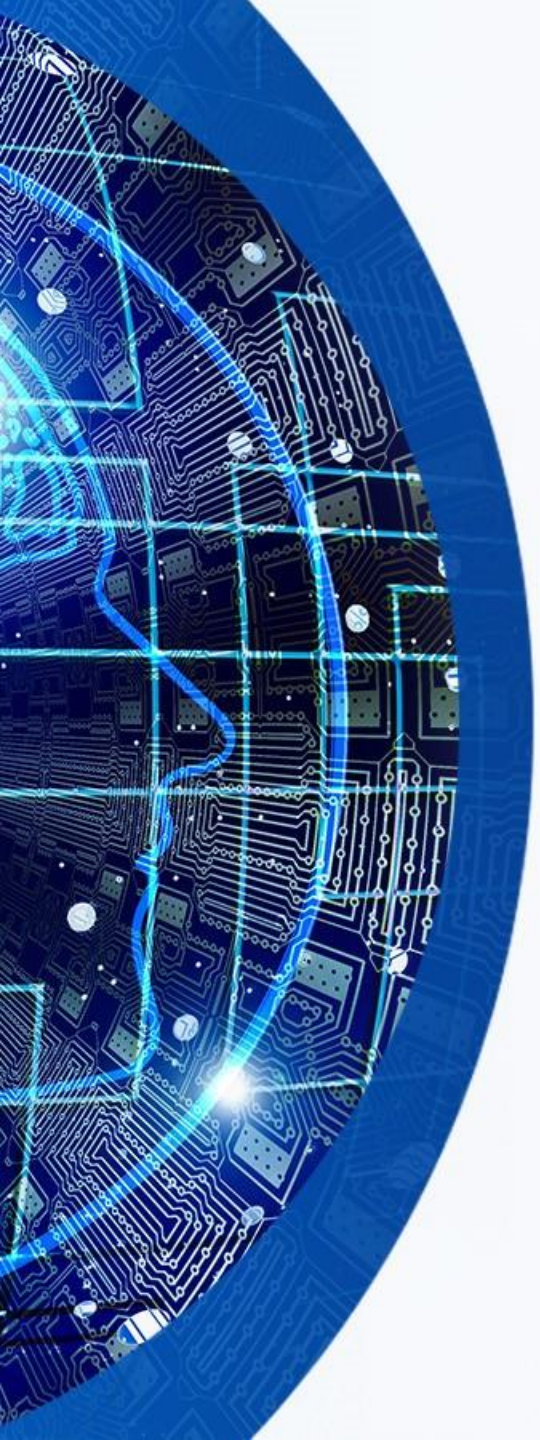
Modeling

In this phase the modeling techniques are selected and applied on the data in the required form. In the case in question, an autoencoder was used. Autoencoders are neural networks with the aim of generating new data by first compressing the input into a space of latent variables and, subsequently, reconstructing the output based on the acquired information.

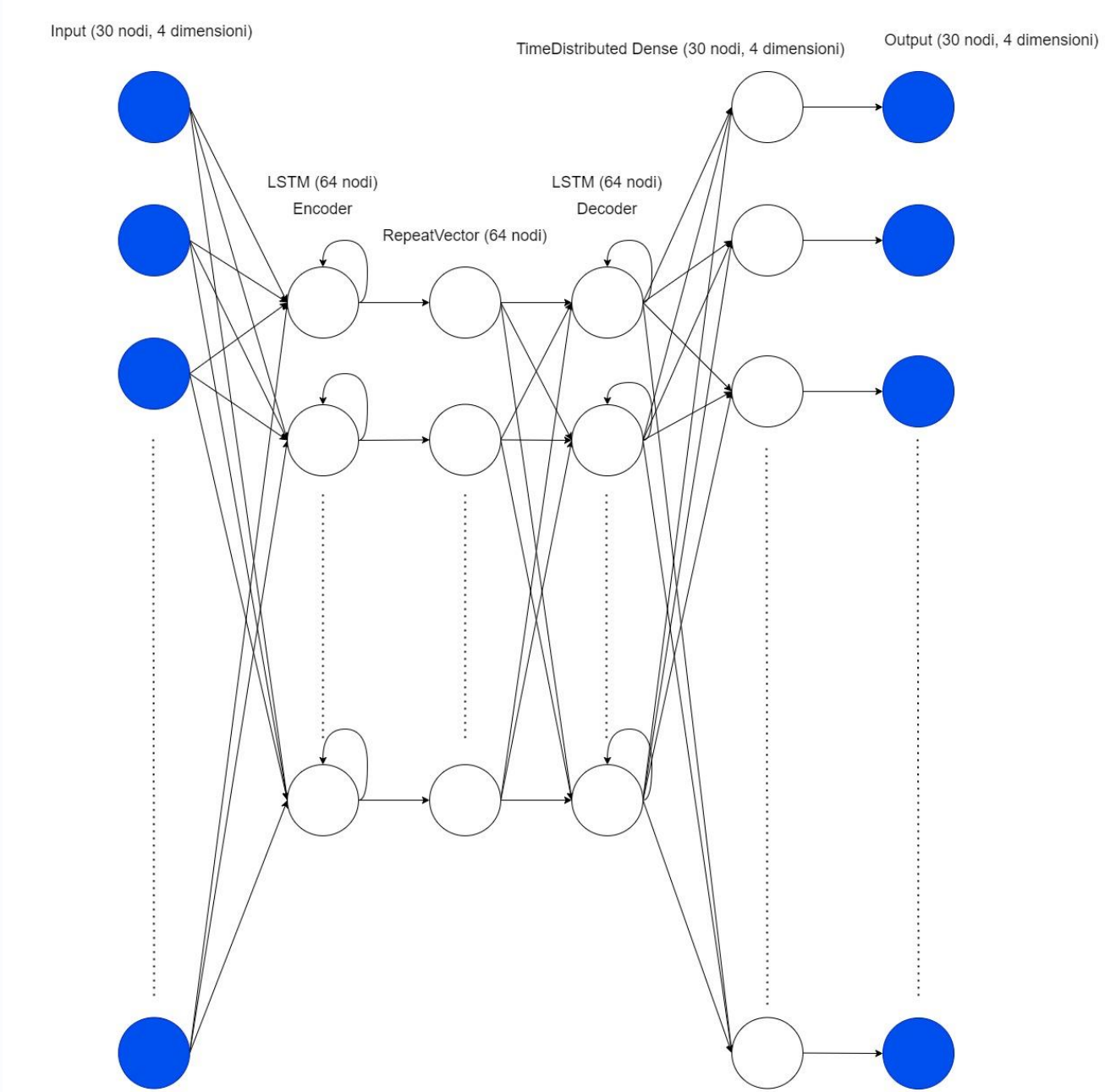
Our model is an autoencoder and is composed by:

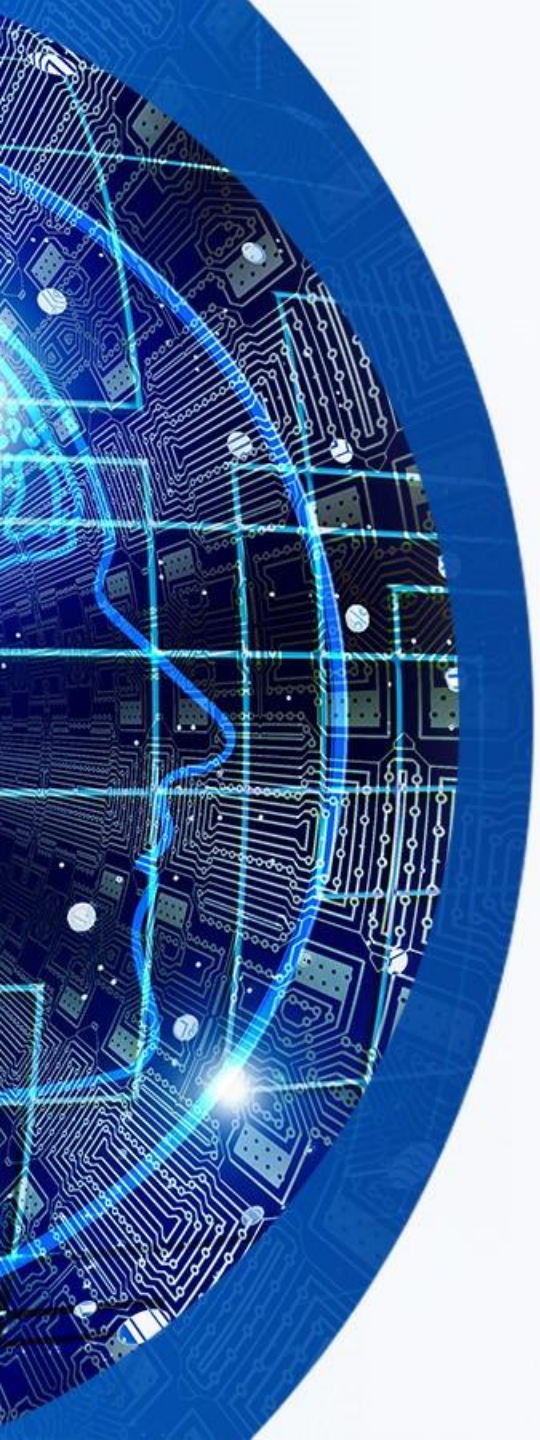
- A LSTM layer with the same input shape as the sequences
- A RepeatVector layer that replicate the output of the previous layer to correctly pass the output to the next layer
- Another LSTM layer
- A TimeDistributed Dense layer without activation function to make to predict of the original input

```
model_ad = Sequential()
model_ad.add(layers.LSTM(64, input_shape=(sequences_ad.shape[1], sequences_ad.shape[2])))
model_ad.add(layers.RepeatVector(sequences_ad.shape[1]))
model_ad.add(layers.LSTM(64, return_sequences=True))
model_ad.add(layers.TimeDistributed(layers.Dense(sequences_ad.shape[2])))
model_ad.compile(loss='mean_absolute_error', optimizer = 'adam')

history = model_ad.fit(sequences_ad, sequences_ad, epochs=10)
```



Modeling


Input (30 nodi, 4 dimensioni)
LSTM (64 nodi)
Encoder
RepeatVector (64 nodi)
LSTM (64 nodi)
Decoder
TimeDistributed Dense (30 nodi, 4 dimensioni)
Output (30 nodi, 4 dimensioni)



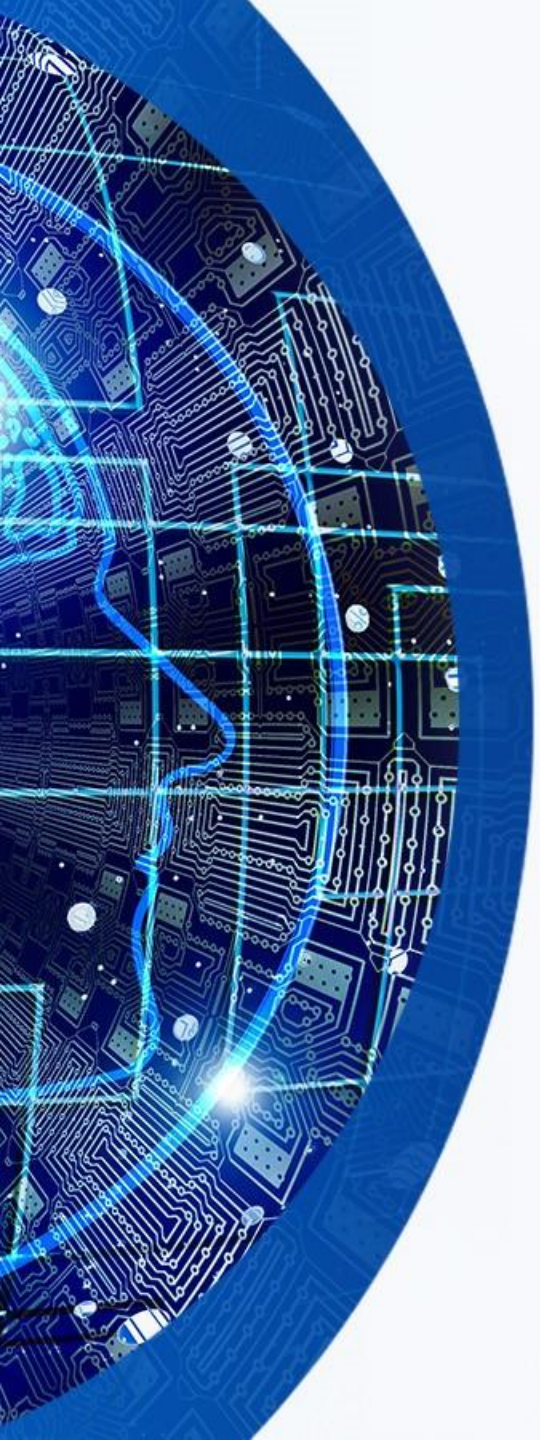
Evaluation

In this phase, we performed the error calculation of each sequence. We calculated the error of a sequence as sum of the absolute error of each attribute of each records in that sequence with the predicted ones.

```
def get_errors(sequences):
    print("getting errors")
    errors = []
    for i in range(sequences.shape[0]):
    #for i in range(100):
      print("\r" + str(int((i+1) / sequences.shape[0] * 100)) + " %", end="")
      error = 0
      prediction = model_ad.predict(sequences[[i]])
      for j in range(sequences.shape[1]):
        for k in range(sequences.shape[2]):
          error += abs(sequences[[i]][0][j][k] - prediction[0][j][k])
      errors.append(error)
    print("\r100 %")
    return errors
```

```
errors = []
for seq in sequences_ad_test:
  errors.append(get_errors(seq))
```

The `get_errors` function receives the time series as input and calculates the errors.

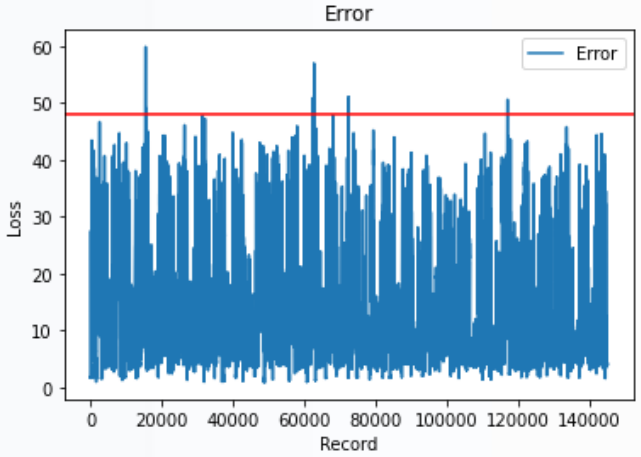


Evaluation

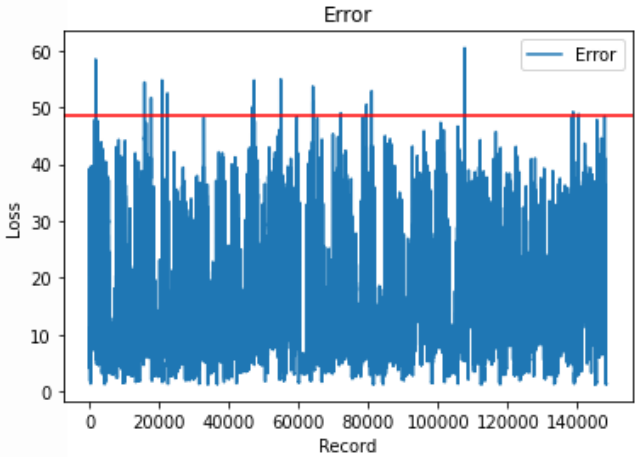
Then, for each patient we have identified a threshold and we plotted the results.

```
tr = []
for error in errors:
  _min = error[0]
  _max = error[0]
  for i in range(len(error)):
    if error[i] < _min:
      _min = error[i]
    if error[i] > _max:
      _max = error[i]
  t = ((_max - _min) / 10 * 8) + _min
  tr.append(t)
  plt.plot(range(len(error)), error,  label='Error')
  plt.axhline(t, color='r')
  plt.title('Error')
  plt.xlabel('Record')
  plt.ylabel('Loss')
  plt.legend()
  plt.show()
```

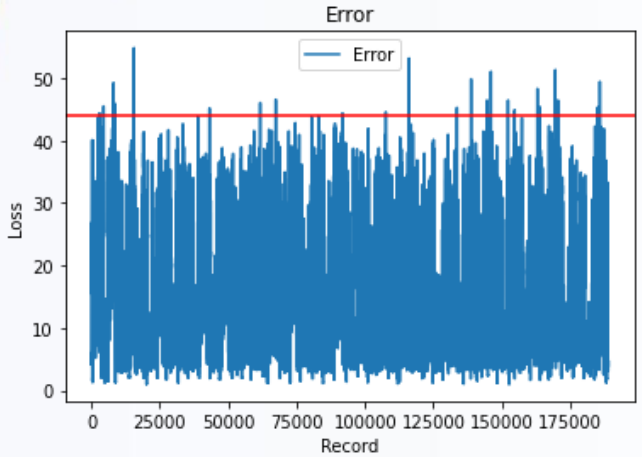
Evaluation



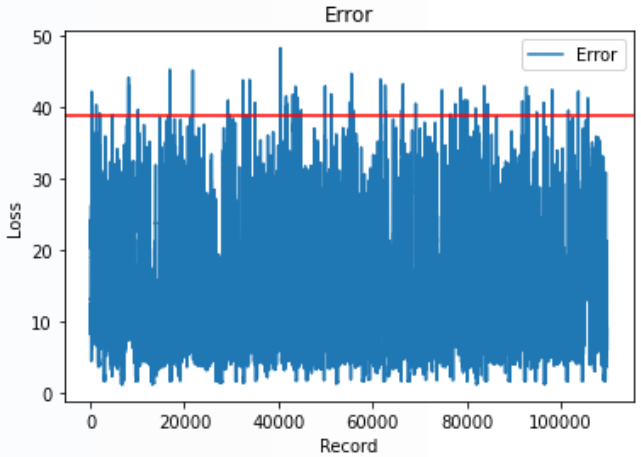
Patient: **1004**



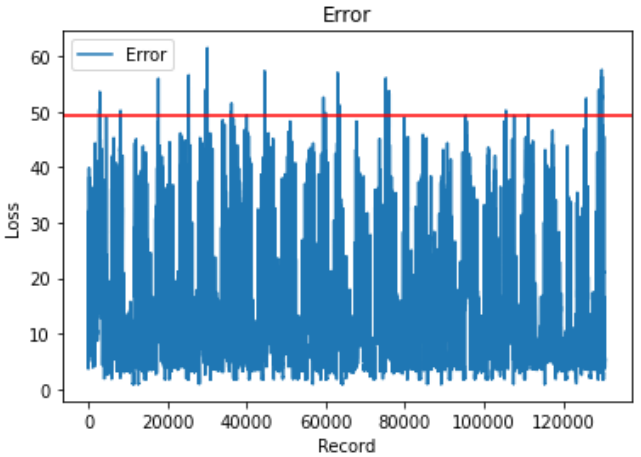
Patient: **1006**



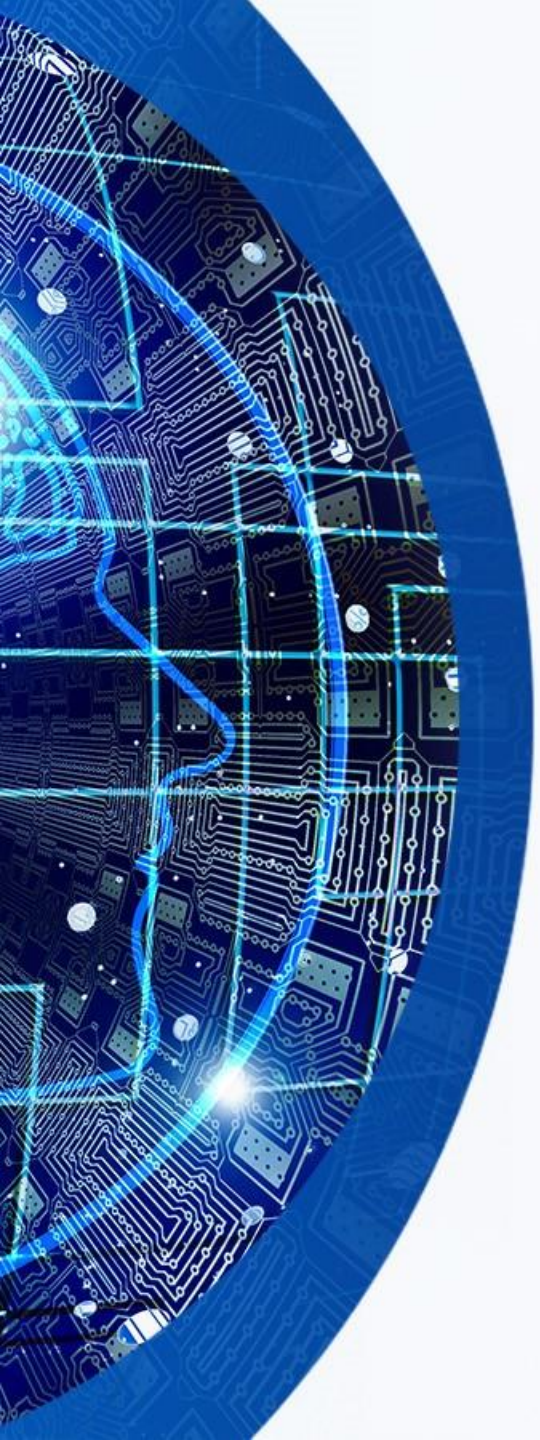
Patient: **3001**



Patient: **3006**



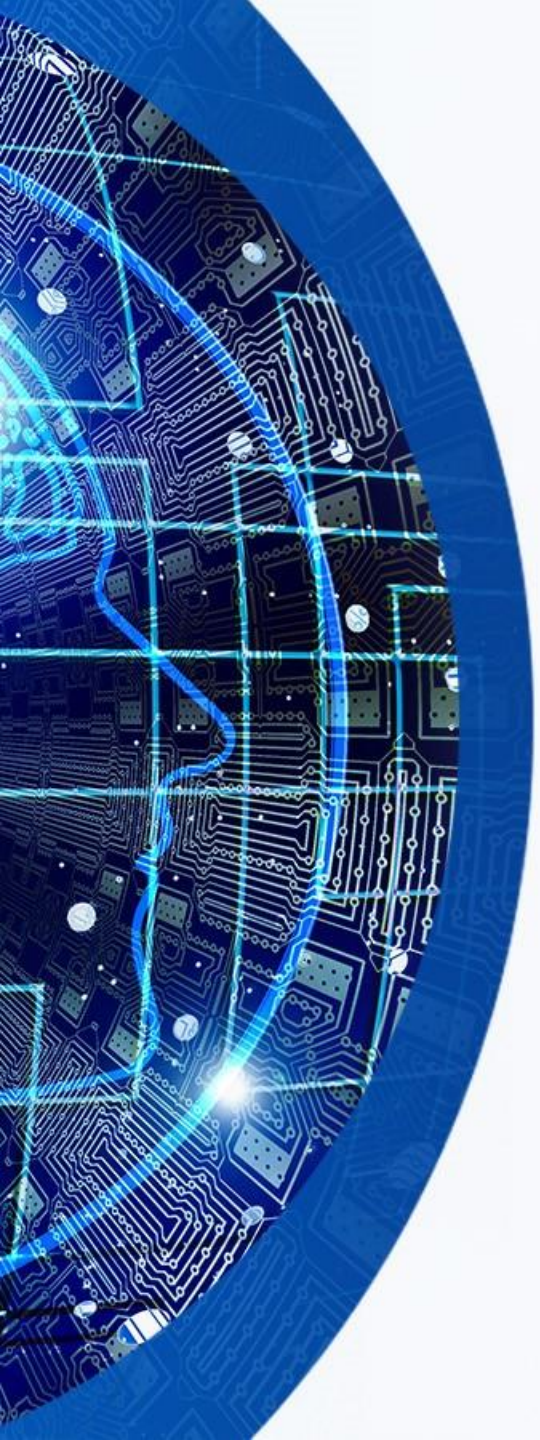
Patient: **4002**



Evaluation

Finally, we have created a csv file in which we write the **off period** of each patient. An off period is when the error of the relative sequence is higher than its threshold.

```
off_period = open("off_period.csv", "w")
off_period.write("patient, tsDate, off\n")
for pat in range(len(errors)):
    for i in range(len(errors[pat])):
      if errors[pat][i] > tr[pat]:
        off_period.write(str(df_copy[pat].iloc[i]['patient']) + "," + str(df_copy[pat].iloc[i]['tsDate']) + ", True\n")
off_period.close()
```



Conclusions

Also, in this task the results obtained are good as the autoencoder is able to reproduce the sequences provided in input. This allows us to identify the anomalies present in the input and therefore the off periods.

Thanks for the attention!