# LSTM autoencoder for Anomaly detection

Sofia Noemi Crobeddu and Sarker Modan Mohan

Master of Biometrics and Intelligent Vision
Université Paris-Est Cretéil (UPEC)
Course of Machine Learning II
Professor Delphine Maugars

UPEC UNIVERSITÉ PARIS-EST CRÉTEIL | FACULTÉ DES SCIENCES ET TECHNOLOGIE

December 11, 2024

## Contents

## Abstract

This project explores the anomaly detection problem in water quality data using a combination of **Long Short-Term Memory (LSTM)** networks and **autoencoder** architectures. LSTM networks are a type of recurrent neural networks (RNNs), and are designed to handle time-series data by capturing long-term dependencies, and in general for sequential data. Through the integration of autoencoders, we introduce a mechanism for unsupervised learning that compresses the input data into a lower-dimensional latent representation before the reconstruction. The efficiency of this approach was validated by an analysis on the loss function and on reconstruction errors.

This project was performed by students Sofia Noemi Crobeddu and Sarker Modan Mohan for the course of Machine Learning II taught by Professor Delphine Maugars.

## 1 Dataset information

The source of this dataset is the GECCO Challenge 2018, *Internet of Things: Online Anomaly Detection for Drinking Water Quality* (see [1]). For this specific edition in 2018, the industrial partner was a water company in Germany, Thuringer Fernwasserversorgung (TFW), and the topic of the challenge was the prediction of any changes in a time series of drinking water data.

The dataset is composed by 11 variables, that are the following ones:

- **Time**: time of measurement, given in following format: `yyyy-mm-dd HH:MM:SS`;
- **Tp**: temperature of the water, given in Celtius;
- **Cl**: amount of chlorine dioxide in the water, given in $\frac{mg}{L}$;
- **pH**: pH value of the water;
- **Redox**: redox potential, given in mV;
- **Leit**: electric conductivity of the water, given in $\frac{\mu S}{cm}$;
- **Trueb**: turbidity of the water, given in NTU;
- **Cl_2**: amount of chlorine dioxide in the water, given in $\frac{mg}{L}$;
- **Fm**: flow rate at water line 1, given in $\frac{m^3}{h}$;
- **Fm_2**: flow rate at water line 2, given in $\frac{m^3}{h}$;
- **EVENT**: it is given in boolean and it is marker when this entry should be considered as a remarkable change response event.

The first important thing to underline is that for this project, since we are working on **unsupervised machine learning for anomaly detection**, we will not use the column *EVENT*.

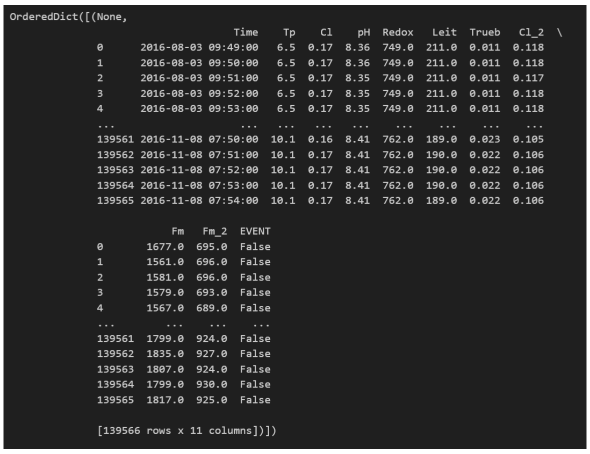
Another highlight is that the dataset covers water quality measurements from August 2016 to November 2016, collected at minute-level intervals.

## 2 Exploratory data analysis

Before applying the model and enter the principal part of the project, we performed an exploratory data analysis, in order to look at the structure and the content of the data.

The file of data is in *.RDS* format, so we read it using *pyreadr* library. We save the dictionary into the variable *data*. The code is shown below and the output is in Figure 1.

```python
import pyreadr

#File path
path = r"C:\Users\sofyc\OneDrive\Desktop\UPEC\ML II\project LSTM\ResourcePackage2018\
    ResourcePackage\source\R\Framework\Data\waterDataTraining.RDS"

#Loading
data = pyreadr.read_r(path) #dictionary

data
```

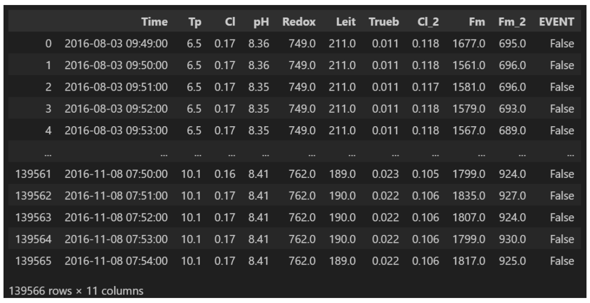


```
OrderedDict([(None,
                           Time    Tp    Cl    pH  Redox   Leit  Trueb   Cl_2  \
        0      2016-08-03 09:49:00   6.5  0.17  8.36  749.0  211.0  0.011  0.118
        1      2016-08-03 09:50:00   6.5  0.17  8.36  749.0  211.0  0.011  0.118
        2      2016-08-03 09:51:00   6.5  0.17  8.35  749.0  211.0  0.011  0.117
        3      2016-08-03 09:52:00   6.5  0.17  8.35  749.0  211.0  0.011  0.118
        4      2016-08-03 09:53:00   6.5  0.17  8.35  749.0  211.0  0.011  0.118
        ...                    ...   ...   ...   ...    ...    ...    ...    ...
        139561 2016-11-08 07:50:00  10.1  0.16  8.41  762.0  189.0  0.023  0.105
        139562 2016-11-08 07:51:00  10.1  0.17  8.41  762.0  190.0  0.022  0.106
        139563 2016-11-08 07:52:00  10.1  0.17  8.41  762.0  190.0  0.022  0.106
        139564 2016-11-08 07:53:00  10.1  0.17  8.41  762.0  190.0  0.022  0.106
        139565 2016-11-08 07:54:00  10.1  0.17  8.41  762.0  189.0  0.022  0.106

                   Fm   Fm_2  EVENT
        0       1677.0  695.0  False
        1       1561.0  696.0  False
        2       1581.0  696.0  False
        3       1579.0  693.0  False
        4       1567.0  689.0  False
        ...        ...    ...    ...
        139561  1799.0  924.0  False
        139562  1835.0  927.0  False
        139563  1807.0  924.0  False
        139564  1799.0  930.0  False
        139565  1817.0  925.0  False

        [139566 rows x 11 columns])])
```

**Fig. 1**: Output of *data*.

After this, we extract the object from the dictionary, and we assign it to the dataframe *df* in order to have an easier manipulation. The code is shown below and Figure 2 is the resulting dataset.

```
#Extracting the object from the dictionary data
df = data[None]  #RDS file format usually saves a single anonimous object

#Take a look
df
```



| | Time | Tp | Cl | pH | Redox | Leit | Trueb | Cl_2 | Fm | Fm_2 | EVENT |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2016-08-03 09:49:00 | 6.5 | 0.17 | 8.36 | 749.0 | 211.0 | 0.011 | 0.118 | 1677.0 | 695.0 | False |
| 1 | 2016-08-03 09:50:00 | 6.5 | 0.17 | 8.36 | 749.0 | 211.0 | 0.011 | 0.118 | 1561.0 | 696.0 | False |
| 2 | 2016-08-03 09:51:00 | 6.5 | 0.17 | 8.35 | 749.0 | 211.0 | 0.011 | 0.117 | 1581.0 | 696.0 | False |
| 3 | 2016-08-03 09:52:00 | 6.5 | 0.17 | 8.35 | 749.0 | 211.0 | 0.011 | 0.118 | 1579.0 | 693.0 | False |
| 4 | 2016-08-03 09:53:00 | 6.5 | 0.17 | 8.35 | 749.0 | 211.0 | 0.011 | 0.118 | 1567.0 | 689.0 | False |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 139561 | 2016-11-08 07:50:00 | 10.1 | 0.16 | 8.41 | 762.0 | 189.0 | 0.023 | 0.105 | 1799.0 | 924.0 | False |
| 139562 | 2016-11-08 07:51:00 | 10.1 | 0.17 | 8.41 | 762.0 | 190.0 | 0.022 | 0.106 | 1835.0 | 927.0 | False |
| 139563 | 2016-11-08 07:52:00 | 10.1 | 0.17 | 8.41 | 762.0 | 190.0 | 0.022 | 0.106 | 1807.0 | 924.0 | False |
| 139564 | 2016-11-08 07:53:00 | 10.1 | 0.17 | 8.41 | 762.0 | 190.0 | 0.022 | 0.106 | 1799.0 | 930.0 | False |
| 139565 | 2016-11-08 07:54:00 | 10.1 | 0.17 | 8.41 | 762.0 | 189.0 | 0.022 | 0.106 | 1817.0 | 925.0 | False |

139566 rows × 11 columns

**Fig. 2**: Output of *df*.

To have an initial idea of our data, we calculate the basic descriptive statistics for the quantitative columns, that will be the ones considered in LSTM model application in next section. In particular, we calculate the mean, the standard deviation, the minimum and the maximum, the quantiles in the following way:

```
import pandas as pd

#Columns names
columns = ['Tp', 'Cl', 'pH', 'Redox', 'Leit', 'Trueb', 'Cl_2', 'Fm', 'Fm_2']

for i in columns:
    display(pd.DataFrame(df[i].describe()).transpose())
```

Table 1 shows the output of the statistics. As we can see, there are some variables like *Fm* and *Fm_2* with a very high standard deviation.

We can now look at the time series plots of the group of variables considered before (the quantitative ones). In this case we also plot the anomalies of the column *EVENT*, i.e. when it is equal to TRUE for each specific time series variable plotted.

```
import matplotlib.pyplot as plt

#Number of subplots excluding 'Time'
```

| Variable | Count | Mean | Std | Min | 25% | 50% | 75% | Max |
|---|---|---|---|---|---|---|---|---|
| Tp | 138522.0 | 8.521406 | 1.281314 | 0.0 | 7.5 | 8.4 | 9.5 | 11.8 |
| Cl | 138521 | 0.165482 | 0.010207 | 0.0 | 0.16 | 0.17 | 0.17 | 0.8 |
| pH | 138522 | 8.366416 | 0.101624 | 4.0 | 8.34 | 8.37 | 8.39 | 8.936228 |
| Redox | 138522 | 752.899009 | 12.945284 | 300.0 | 751.0 | 754.0 | 756.0 | 895.0 |
| Leit | 138522 | 209.520998 | 7.747497 | 0.0 | 209.0 | 211.0 | 211.0 | 646.0 |
| Trueb | 138522 | 0.019778 | 0.005492 | 0.0 | 0.016 | 0.018 | 0.023 | 0.254 |
| Cl_2 | 138522 | 0.106099 | 0.007132 | 0.0 | 0.103 | 0.106 | 0.11 | 0.462 |
| Fm | 138522 | 1534.208891 | 208.171011 | 0.0 | 1388.0 | 1512.0 | 1650.0 | 3923.0 |
| Fm_2 | 138522 | 927.728014 | 147.44743 | 0.0 | 847.0 | 925.0 | 1000.0 | 2592.0 |

**Table 1**: Descriptive Statistics of the variables.

```
num_cols = len(columns) - 1
rows = (num_cols + 2) // 3  #3 subplots per row

fig, axes = plt.subplots(rows, 3, figsize=(20, 4 * rows), constrained_layout=True)
axes = axes.flatten()

for i, col in enumerate(columns):
    ax = axes[i]

    #Time series of the specific variable
    ax.plot(df['Time'], df[col], label=col, color='forestgreen')

    #Anomalies where 'EVENT'=True
    anomaly_mask = df['EVENT'] == True
    ax.plot(df['Time'][anomaly_mask], df[col][anomaly_mask], '.', label='Anomaly', color='red',
     markersize=3)

    ax.set_xlabel('Time')
    ax.set_ylabel(col)
    ax.grid(True)
    ax.legend()
plt.show()
```

As we can see from Figure 3, the plot of *Fm* and *Fm_2* reflect the high standard deviation.

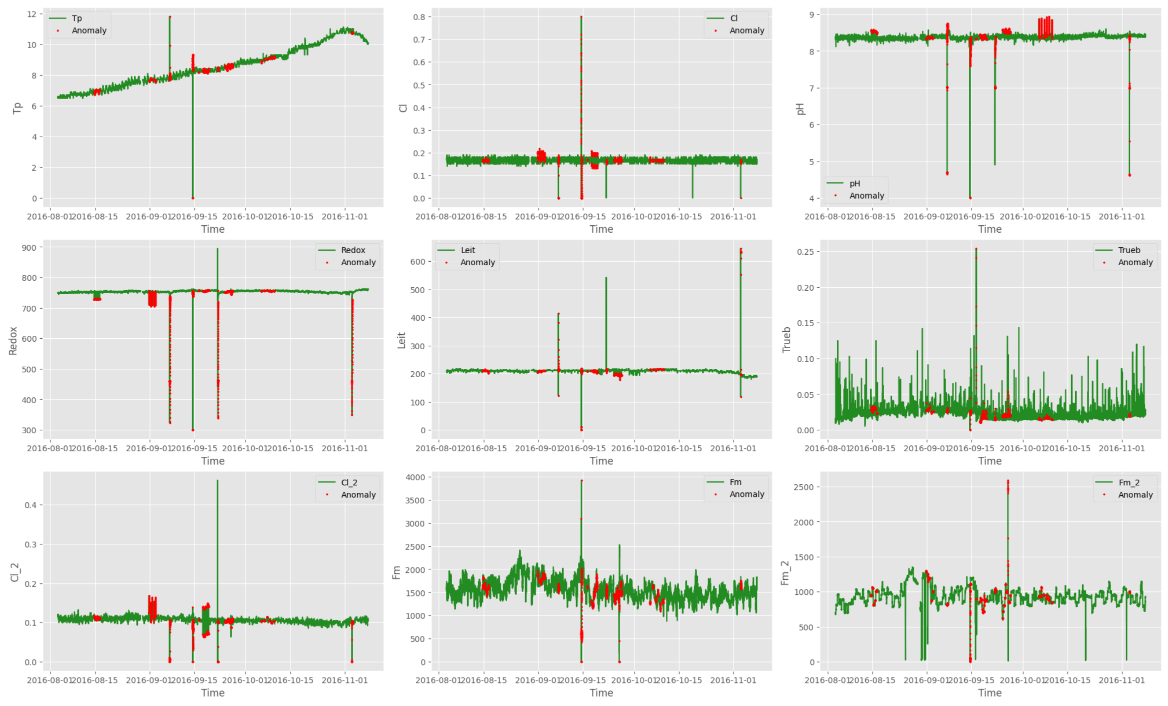


**Fig. 3**: Time series plots of the variables with anomalies of column *EVENT*.

Now we can look at the seasonality , as our final step of the EDA.
In general, the seasonality refers to repeated variations in specific temporal intervals, for example hourly, daily, monthly. This analysis could help in evaluating recurrent patterns that can influence the anomalies. To do this, we aggregate data based on month, day and hour, creating three new columns and of course saving the year 2016 in another separated one.

```python
#Just an initial checkl for the datetime format
df['Time'] = pd.to_datetime(df['Time'])

#Extracting the year, month, the day and the hour.
df.loc[:, 'year'] = df['Time'].dt.year
df.loc[:, 'month'] = df['Time'].dt.month
df.loc[:, 'day'] = df['Time'].dt.day
df.loc[:, 'hour'] = df['Time'].dt.hour

#Columns to plot are the same defined before

#Groupifying data by month
df_plot_monthly = (
    df.groupby(['month'])[columns].mean()
    .reset_index()
    .sort_values(by='month')
)

#Groupifying data by day
df.loc[:, 'day_of_month'] = df['Time'].dt.day
df_plot_daily = (
    df.groupby(['day_of_month'])[columns].mean()
    .reset_index()
    .sort_values(by='day_of_month')
)

#Groupifying data by hour
df_plot_hourly = (
    df.groupby(['hour'])[columns].mean()
    .reset_index()
    .sort_values(by='hour')
)

#Plot settings
color = ['gold', 'darkorange', 'firebrick', 'forestgreen', 'dodgerblue',
         'royalblue', 'black', 'purple', 'orchid'] #Colors
plt.style.use('ggplot') #Style

fig, axes = plt.subplots(1, 3, figsize=(20, 5))

#Plot for monthly seasonality
for i, col in enumerate(columns):
    axes[0].plot(df_plot_monthly['month'], df_plot_monthly[col],
                 label=col, color=color[i], linewidth=2)
axes[0].set_title('Monthly Seasonal Plot', fontsize=15)
axes[0].set_xlabel('Month', fontsize=10)
axes[0].set_ylabel('Average Value', fontsize=10)
axes[0].set_xticks([8, 9, 10, 11])
axes[0].set_xticklabels(['Aug', 'Sep', 'Oct', 'Nov'], fontsize=9)
axes[0].legend(title='Variables', fontsize=8, loc='upper left', bbox_to_anchor=(1, 1))

#Plot for daily seasonality
for i, col in enumerate(columns):
    axes[1].plot(df_plot_daily['day_of_month'], df_plot_daily[col],
                 label=col, color=color[i], linewidth=2)
axes[1].set_title('Daily Seasonal Plot', fontsize=15)
axes[1].set_xlabel('Day of Month', fontsize=10)
axes[1].set_ylabel('Average Value', fontsize=10)
axes[1].legend(title='Variables', fontsize=8, loc='upper left', bbox_to_anchor=(1, 1))

#Plot for hourly seasonality
for i, col in enumerate(columns):
    axes[2].plot(df_plot_hourly['hour'], df_plot_hourly[col],
                 label=col, color=color[i], linewidth=2)
axes[2].set_title('Hourly Seasonal Plot', fontsize=15)
axes[2].set_xlabel('Hour', fontsize=10)
axes[2].set_ylabel('Average Value', fontsize=10)
axes[2].legend(title='Variables', fontsize=8, loc='upper left', bbox_to_anchor=(1, 1))

plt.tight_layout()
plt.show()
```

As we can notice from Figure 4, also here the two variables *Fm* and *Fm_2* show a seasonal effect, while the other remain more stable. In particular, *Fm* shows regular peaks in the central hours of the day (between 7 am and 6 pm more or less), and this could suggest a higher use of the water during working/common routine hours.

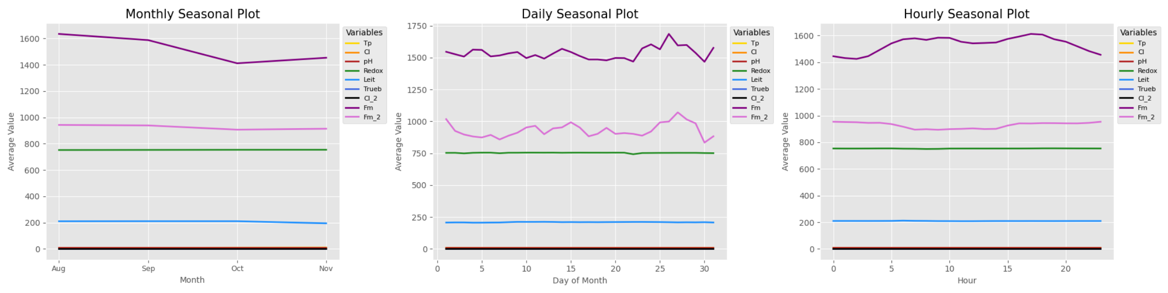
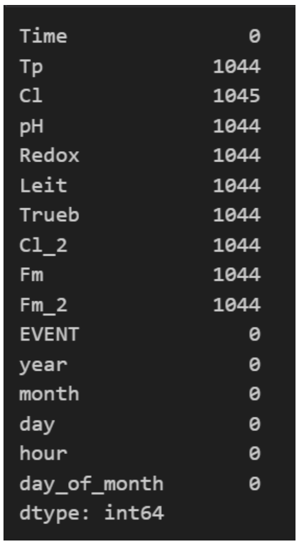


**Fig. 4**: Seasonality plots of the variables.

## 3 LSTM autoencoder model

Before building the model, we need to look at the NAs values in the columns of interest. We decided to fill these values with the mean of the column, in order to not influence the analysis. The code for this preliminary pre-processing is shown below with the corresponding images of the outputs (Figures 5 and 6).

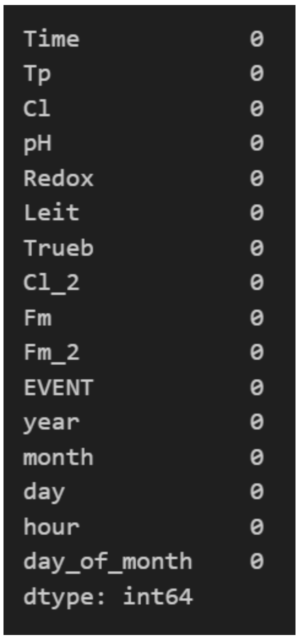
```
#Verifying if there are missing values
print(df.isna().sum())
```



```
Time              0
Tp             1044
Cl             1045
pH             1044
Redox          1044
Leit           1044
Trueb          1044
Cl_2           1044
Fm             1044
Fm_2           1044
EVENT             0
year              0
month             0
day               0
hour              0
day_of_month      0
dtype: int64
```

**Fig. 5**: Initial quantity of NA values in *df*.

```
#Filling missing values with the column mean
data_cleaned = df.fillna(df.mean())

#Verifying if missing values are handled
print(data_cleaned.isna().sum())  #Should print all zeros
```

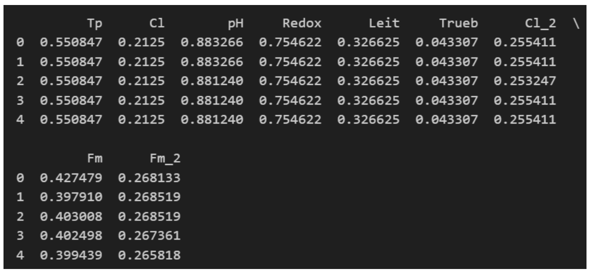


```
Time            0
Tp              0
Cl              0
pH              0
Redox           0
Leit            0
Trueb           0
Cl_2            0
Fm              0
Fm_2            0
EVENT           0
year            0
month           0
day             0
hour            0
day_of_month    0
dtype: int64
```

**Fig. 6**: Check of the remotion of NA values in *df*.

Since LSTM model needs normal data to converge, i.e. it is sensitive to the scale of the variables, we need to standardize or normalize the quantitative data. We chose to apply the **Normalization** through *MinMaxScaler*: it scales all the values between the range [0,1]. Normalized data are shown in Figure 7.

```python
from sklearn.preprocessing import MinMaxScaler

#Normalizing the data using MinMaxScaler
scaler = MinMaxScaler()
data_cleaned = data_cleaned[columns]
data_normalized = pd.DataFrame(scaler.fit_transform(data_cleaned), columns=columns)

#Check normalized data
print(data_normalized.head())
```



```
         Tp      Cl        pH     Redox      Leit     Trueb      Cl_2  \
0  0.550847  0.2125  0.883266  0.754622  0.326625  0.043307  0.255411
1  0.550847  0.2125  0.883266  0.754622  0.326625  0.043307  0.255411
2  0.550847  0.2125  0.881240  0.754622  0.326625  0.043307  0.253247
3  0.550847  0.2125  0.881240  0.754622  0.326625  0.043307  0.255411
4  0.550847  0.2125  0.881240  0.754622  0.326625  0.043307  0.255411

         Fm      Fm_2
0  0.427479  0.268133
1  0.397910  0.268519
2  0.403008  0.268519
3  0.402498  0.267361
4  0.399439  0.265818
```

**Fig. 7**: Output of data normalized.

Now, we create temporal sequences with a temporal window, i.e. *timesteps* set at 30 min. In this way, each sequence will have the format

$$(30, \text{num\_features})$$

where num_features is the number of variables considered and num_samples is the length of the dataset. Our input shape of the output in the code below, is in fact (139537, 30, 9), where 9 is the number of features and 139537 indicates the num_samples, i.e. the number of sequences created.

```python
#Function to create time-series sequences
def create_sequences(data, timesteps):
    sequences = []
    for i in range(len(data) - timesteps + 1):
        seq = data.iloc[i : i + timesteps].values
        sequences.append(seq)
    return np.array(sequences)

#Setting time steps
timesteps = 30

#Creating sequences
input_data = create_sequences(data_normalized, timesteps)
print(f"Input shape: {input_data.shape}")  #Example: (num_samples, timesteps, num_features)
```

Now we split the sequential data into train and validation sets, and more specifically into 80% and 20%, as shown below.

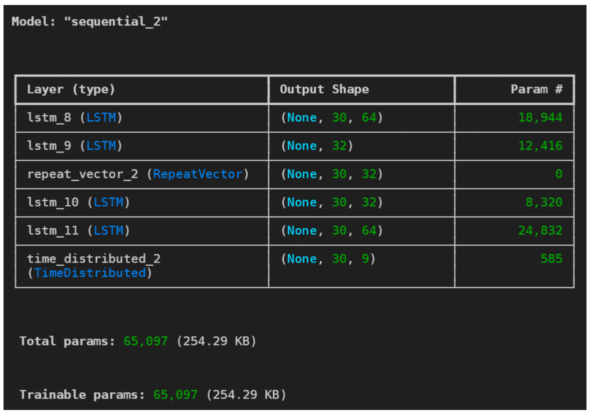
```python
from sklearn.model_selection import train_test_split

#Splitting the data into training and validation sets
X_train, X_val = train_test_split(input_data, test_size=0.2, random_state=42)
print(f"Training shape: {X_train.shape}, Validation shape: {X_val.shape}")
```

The shapes from the output are:

- (111629, 30, 9) for the training set;
- (27908, 30, 9) for the validation set.

In general, LSTM autoencoder model has an architecture with encoder, a RepeatVector layer and decoder. Also for our model we followed this structure. The **encoder**, in fact, compresses the input sequences into a smaller latent representation. It is composed by two LSTM layers: the first one with 64 neurons to capture complex temporal dependencies and the second one with 32. Moreover, in both the activation function used is *ReLU* function for non-linearity, and the second LSTM layer serves as the bottleneck, capturing the most essential features of the input sequence. More specifically, the bottleneck is the output of the encoder, i.e. a single vector that represents the entire input sequence in a reduced-dimensional form. In this way we are prioritizing key features.
The following layer is built with *RepeatVector* function. It unrolls the bottleneck vector into a repeated sequence of length equal to the number of timesteps. Basically, it expands the compact representation. After this, the **decoder** reconstructs the input sequence from the compressed bottleneck vector, through two LSTM layers (with 32 and 64 neurons respectively). It gradually reconstructs sequences of the original shape.
The final *TimeDistributed(Dense)* layer applies a fully connected layer to each timestep independently. In the model compilation we used the Mean Squared Error (MSE) between the input and the reconstructed sequences, in order to measure the reconstruction error (the goal is to minimize the loss function). In this way we are training the model to identify normal sequences.
The optimizer used is **Adam**, with a learning rate of 0.001. We also tried with a higher learning rate at 0.01, but the loss function was not good. At the same time, even if a lower learning rate of 0.0001 was giving a good performance, it was too slow and we decided to use the most suitable and efficient value.
Going on with the description, we also added the **Gradient clipping**, that prevents the gradients from exploding, i.e. from vanishing, during backpropagation, ensuring stable training.
The code for the model implementation is shown below, and the output of the summary of the model is in Figure 8.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, RepeatVector, TimeDistributed
from tensorflow.keras.optimizers import Adam

#Define the LSTM Autoencoder
model = Sequential([
    LSTM(64, activation='relu', input_shape=(timesteps, input_data.shape[2]), return_sequences=
     True),
    LSTM(32, activation='relu', return_sequences=False),
    RepeatVector(timesteps),
    LSTM(32, activation='relu', return_sequences=True),
    LSTM(64, activation='relu', return_sequences=True),
    TimeDistributed(Dense(input_data.shape[2]))
])

#Compile the model with a low learning rate and gradient clipping
model.compile(optimizer=Adam(learning_rate=0.001, clipvalue=1.0), loss='mse')
model.summary()
```

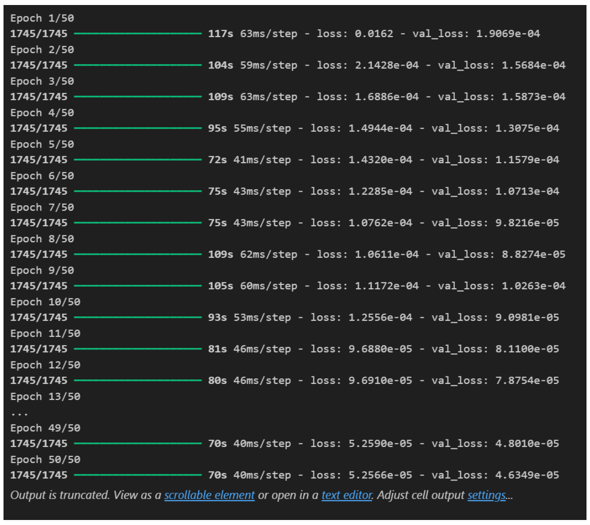


```
Model: "sequential_2"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ lstm_8 (LSTM)                   │ (None, 30, 64)         │        18,944 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ lstm_9 (LSTM)                   │ (None, 32)             │        12,416 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ repeat_vector_2 (RepeatVector)  │ (None, 30, 32)         │             0 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ lstm_10 (LSTM)                  │ (None, 30, 32)         │         8,320 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ lstm_11 (LSTM)                  │ (None, 30, 64)         │        24,832 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ time_distributed_2              │ (None, 30, 9)          │           585 │
│ (TimeDistributed)               │                        │               │
└─────────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 65,097 (254.29 KB)

 Trainable params: 65,097 (254.29 KB)
```

**Fig. 8**: Summary of the model.

The model is now trained to reconstruct the non-anomalous data, using a *batch size* of 64 and 50 *epochs*.
We tried also with 30 and 40 epochs, but the loss function was totally unstable.

```
#Train the model
history = model.fit(
    X_train, X_train,
    epochs=50,
    batch_size=64,
    validation_data=(X_val, X_val),
    shuffle=True
)
```



```
Epoch 1/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 117s 63ms/step - loss: 0.0162 - val_loss: 1.9069e-04
Epoch 2/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 104s 59ms/step - loss: 2.1428e-04 - val_loss: 1.5684e-04
Epoch 3/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 109s 63ms/step - loss: 1.6886e-04 - val_loss: 1.5873e-04
Epoch 4/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 95s 55ms/step - loss: 1.4944e-04 - val_loss: 1.3075e-04
Epoch 5/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 72s 41ms/step - loss: 1.4320e-04 - val_loss: 1.1579e-04
Epoch 6/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 75s 43ms/step - loss: 1.2285e-04 - val_loss: 1.0713e-04
Epoch 7/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 75s 43ms/step - loss: 1.0762e-04 - val_loss: 9.8216e-05
Epoch 8/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 109s 62ms/step - loss: 1.0611e-04 - val_loss: 8.8274e-05
Epoch 9/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 105s 60ms/step - loss: 1.1172e-04 - val_loss: 1.0263e-04
Epoch 10/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 93s 53ms/step - loss: 1.2556e-04 - val_loss: 9.0981e-05
Epoch 11/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 81s 46ms/step - loss: 9.6880e-05 - val_loss: 8.1100e-05
Epoch 12/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 80s 46ms/step - loss: 9.6910e-05 - val_loss: 7.8754e-05
Epoch 13/50
...
Epoch 49/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 70s 40ms/step - loss: 5.2590e-05 - val_loss: 4.8010e-05
Epoch 50/50
1745/1745 ━━━━━━━━━━━━━━━━━━━━ 70s 40ms/step - loss: 5.2566e-05 - val_loss: 4.6349e-05
```

*Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...*

**Fig. 9**: Output of the epochs.

Figure 10 shows how the error decreases during the training. As we can see, the loss curve indicates a strong convergence. The drop at the beginning in the training loss shows the model rapidly minimizing its error, followed by a plateau. Hence, the model is learning effectively in early epochs.

The validation loss, instead, follows closely the training loss, which suggests that the model is not overfitting. Generally, since the final values of both training and validation loss are very low, we can say that the model has learned the patterns in the data very well, with minimal error.

```python
#Plot training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```

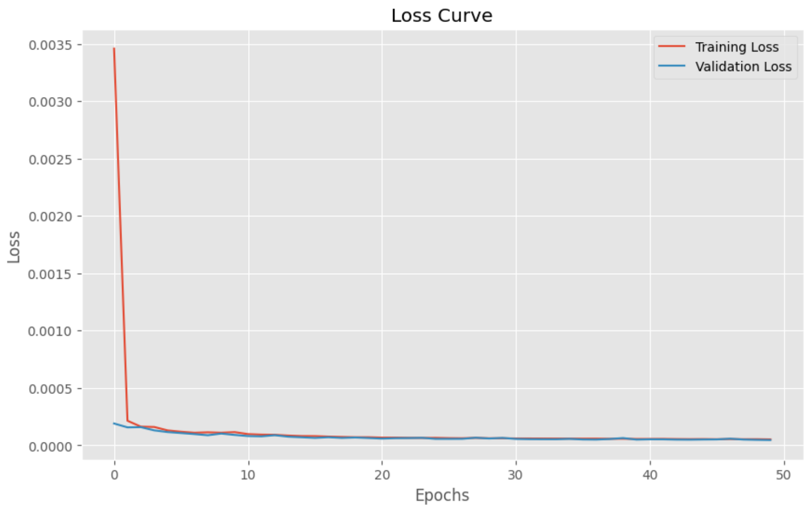
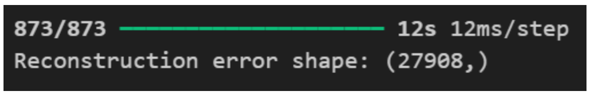


**Fig. 10**: Loss curves.

Here we look at the reconstruction error: it measures the difference between original data and the reconstructed ones. To interpret it, we need to know that high values indicates sequences of data that the model doesn't manage to interpret properly, i.e. potential anomalies.

```python
import numpy as np

#Prediction on validation data
reconstructed = model.predict(X_val)

#Calculating the reconstruction error
reconstruction_error = np.mean(np.power(X_val - reconstructed, 2), axis=(1, 2))
print(f"Reconstruction error shape: {reconstruction_error.shape}")
```



```
873/873 ━━━━━━━━━━━━━━━━━━━━ 12s 12ms/step
Reconstruction error shape: (27908,)
```

**Fig. 11**: Reconstruction error shape.

To calculate the reconstruction error we use the 95th percentile that defines the threshold, in order to detect the anomalies.
The code to calculate it is shown below, and the final output from the line code *print()* is of 1396 anomalies detected, while the threshold is $7.8 \cdot 10^{-5}$.
Figure 12 shows a graph with the reconstruction errors in blue and the threshold in red. As we can notice, the majority of the data points has a low reconstruction error, which is below the threshold. Hence, the autoencoder is accurately reconstructing most of the data. High peaks represent, instead, anomalies, as they points exceed the threshold and deviates from the norm. Finally, we could say

that the distribution of anomalies seems sparse that implies rare occurrences of anomalies, in fact we detected just 1396 anomalies that is a small number if we consider the total length of the dataset (i.e. 139566). In addiction, since we have also the column *EVENT*, we can compare our result with the anomalies detected a priori for the Gecco challenge. The column contains 1726 values equal to TRUE. This means that our model detected less anomalies compared to the ones indicated by the column, but the results are still very close. We need also to consider that, based on the rules of the challenge (see [1]), in *EVENT* there are both real anomalies and artificial ones created specifically for the challenge. Hence, we can conclude that our model has a good convergence.

```python
#Defining threshold (95th percentile of reconstruction error)
threshold = np.percentile(reconstruction_error, 95)
print(f"Anomaly detection threshold: {threshold}")

#Identifying anomalies
anomalies = reconstruction_error > threshold
print(f"Number of anomalies detected: {np.sum(anomalies)}")
```

```python
#Plot reconstruction errors
plt.figure(figsize=(10, 6))
plt.plot(reconstruction_error, label='Reconstruction Error', color='blue')
plt.axhline(y=threshold, color='r', linestyle='--', label='Threshold')
plt.legend()
plt.xlabel("Data Point")
plt.ylabel("Reconstruction Error")
plt.show()
```

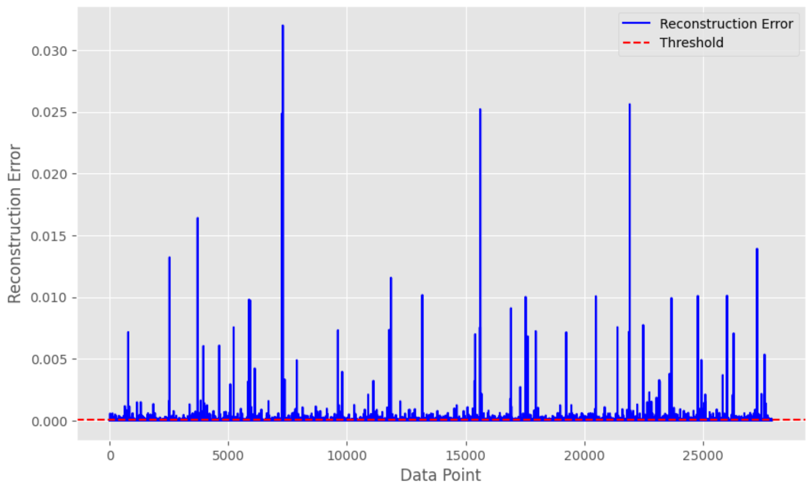
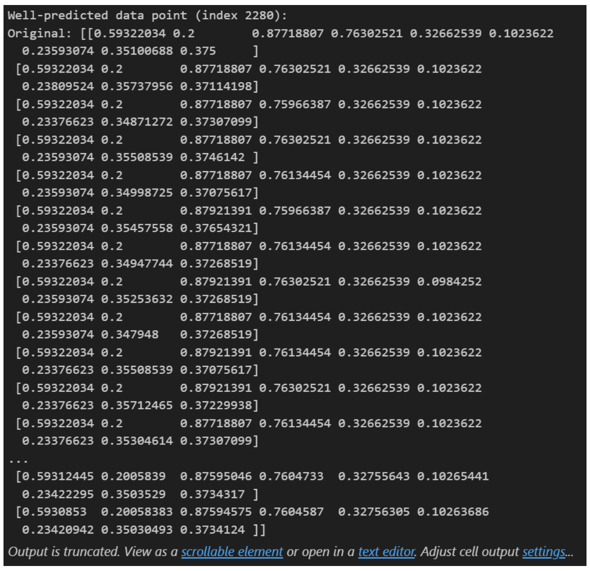


**Fig. 12**: Reconstruction error for validation data.

For the final step, we look at two examples: one with well-predicted data, and the other with poorly-predicted data. The codes are shown below and the outputs are in Figures 13 and 14.

```python
#Example of well-predicted data
well_predicted_index = np.argmin(reconstruction_error)
print(f"Well-predicted data point (index {well_predicted_index}):")
print(f"Original: {X_val[well_predicted_index]}")
print(f"Reconstructed: {reconstructed[well_predicted_index]}")
```

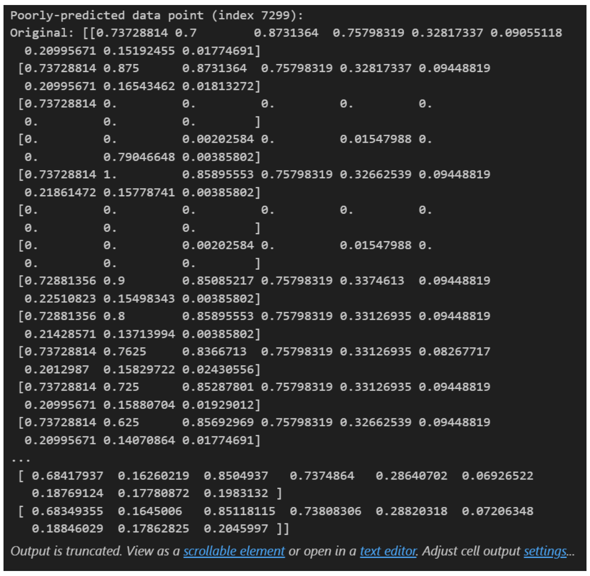


```
Well-predicted data point (index 2280):
Original: [[0.59322034 0.2        0.87718807 0.76302521 0.32662539 0.1023622
  0.23593074 0.35100688 0.375     ]
 [0.59322034 0.2        0.87718807 0.76302521 0.32662539 0.1023622
  0.23809524 0.35737956 0.37114198]
 [0.59322034 0.2        0.87718807 0.75966387 0.32662539 0.1023622
  0.23376623 0.34871272 0.37307099]
 [0.59322034 0.2        0.87718807 0.76302521 0.32662539 0.1023622
  0.23593074 0.35508539 0.3746142 ]
 [0.59322034 0.2        0.87718807 0.76134454 0.32662539 0.1023622
  0.23593074 0.34998725 0.37075617]
 [0.59322034 0.2        0.87921391 0.75966387 0.32662539 0.1023622
  0.23593074 0.35457558 0.37654321]
 [0.59322034 0.2        0.87718807 0.76134454 0.32662539 0.1023622
  0.23376623 0.34947744 0.37268519]
 [0.59322034 0.2        0.87921391 0.76302521 0.32662539 0.0984252
  0.23593074 0.35253632 0.37268519]
 [0.59322034 0.2        0.87718807 0.76134454 0.32662539 0.1023622
  0.23593074 0.347948   0.37268519]
 [0.59322034 0.2        0.87921391 0.76134454 0.32662539 0.1023622
  0.23376623 0.35508539 0.37075617]
 [0.59322034 0.2        0.87921391 0.76302521 0.32662539 0.1023622
  0.23376623 0.35712465 0.37229938]
 [0.59322034 0.2        0.87718807 0.76134454 0.32662539 0.1023622
  0.23376623 0.35304614 0.37307099]
...
 [0.59312445 0.2005839  0.87595046 0.7604733  0.32755643 0.10265441
  0.23422295 0.3503529  0.3734317 ]
 [0.5930853  0.20058383 0.87594575 0.7604587  0.32756305 0.10263686
  0.23420942 0.35030493 0.3734124 ]]
```

*Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...*

**Fig. 13**: Example of well-predicted data.

```python
#Example of poorly-predicted data
poorly_predicted_index = np.argmax(reconstruction_error)
print(f"Poorly-predicted data point (index {poorly_predicted_index}):")
print(f"Original: {X_val[poorly_predicted_index]}")
print(f"Reconstructed: {reconstructed[poorly_predicted_index]}")
```



```
Poorly-predicted data point (index 7299):
Original: [[0.73728814 0.7        0.8731364  0.75798319 0.32817337 0.09055118
  0.20995671 0.15192455 0.01774691]
 [0.73728814 0.875      0.8731364  0.75798319 0.32817337 0.09448819
  0.20995671 0.16543462 0.01813272]
 [0.73728814 0.         0.         0.         0.         0.
  0.         0.         0.        ]
 [0.         0.         0.00202584 0.         0.01547988 0.
  0.         0.79046648 0.00385802]
 [0.73728814 1.         0.85895553 0.75798319 0.32662539 0.09448819
  0.21861472 0.15778741 0.00385802]
 [0.         0.         0.         0.         0.         0.
  0.         0.         0.        ]
 [0.         0.         0.00202584 0.         0.01547988 0.
  0.         0.         0.        ]
 [0.72881356 0.9        0.85085217 0.75798319 0.3374613  0.09448819
  0.22510823 0.15498343 0.00385802]
 [0.72881356 0.8        0.85895553 0.75798319 0.33126935 0.09448819
  0.21428571 0.13713994 0.00385802]
 [0.73728814 0.7625     0.8366713  0.75798319 0.33126935 0.08267717
  0.2012987  0.15829722 0.02430556]
 [0.73728814 0.725      0.85287801 0.75798319 0.33126935 0.09448819
  0.20995671 0.15880704 0.01929012]
 [0.73728814 0.625      0.85692969 0.75798319 0.32662539 0.09448819
  0.20995671 0.14070864 0.01774691]
...
 [ 0.68417937  0.16260219  0.8504937   0.7374864   0.28640702  0.06926522
   0.18769124  0.17780872  0.1983132 ]
 [ 0.68349355  0.1645006   0.85118115  0.73808306  0.28820318  0.07206348
   0.18846029  0.17862825  0.2045997 ]]
```

*Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...*

**Fig. 14**: Example of poorly-predicted data.

## References

[1] Team, S.: GECCO Challenge 2018 (2018). https://www.spotseven.de/gecco/gecco-challenge/gecco-challenge-2018/

[2] Science, T.D.: Time Series of Price Anomaly Detection with LSTM (2021). https://towardsdatascience.com/time-series-of-price-anomaly-detection-with-lstm-11a12ba4f6d9

[3] ResearchGate: MSE and MAE curves of Auto-encoder (2020). https://www.researchgate.net/figure/MSE-and-MAE-curves-of-Auto-encoder_fig2_338742574

[4] Zhonghong: Anomaly Detection in Time Series Data Using LSTM Autoencoders (2020). https://medium.com/@zhonghong9998/anomaly-detection-in-time-series-data-using-lstm-autoencoders-51fd14946fa3

[5] Science, T.D.: Using LSTM Autoencoders on Multidimensional Time Series Data (2022). https://towardsdatascience.com/using-lstm-autoencoders-on-multidimensional-time-series-data-f5a7a51b29a1

[6] YouTube: LSTM Autoencoder for Anomaly Detection. YouTube video (2022). https://youtu.be/6S2v7G-OupA

[7] Maugars, D.: Panorama de l'IA. PDF document (2023-2024)

[8] Brett, K.: Time Series Forecasting: A Practical Guide to Exploratory Data Analysis (2020). https://towardsdatascience.com/time-series-forecasting-a-practical-guide-to-exploratory-data-analysis-a101dc5f85b1