CSE 676 B: Deep Learning, Spring 2024
Assignment 2
Autoencoder and Transformer Architecture

Name: Bhavesh Tharlapally
UBIT: bhavesht
UB Person Number: 50541076

Part II: Autoencoders for Anomaly Detection

1)The dataset selected for building autoencoders for anomaly detection is "occupancy_6005.csv. It is a timeseries data for real-word traffic data. We are encountering time-series data against values. The dataset comprises of 2380 rows and 2 columns. The dataset contains the following relevant information:

- Timestamp
- Value for a rea traffic dataset

Selecting a real traffic dataset with timestamp and value features is ideal for anomaly detection due to its temporal nature, relevance to real-world scenarios, and enriched feature space.
This dataset offers valuable contextual information and enable capturing temporal dependencies, making them well-suited for practical applications like traffic management. Developers can efficiently benchmark and compare anomaly detection algorithms, leading to the development of robust solutions tailored to time-series analysis.
Dataset shape:

```
# Shape of the dataset
df.shape
```

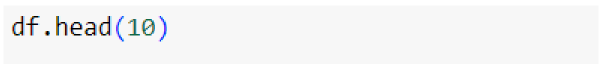
```
(2380, 2)
```

Main Statistics:

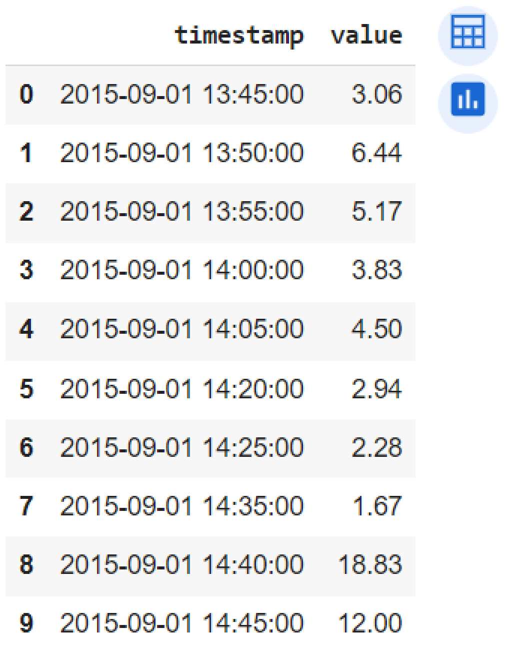
```
# Main statistics of the dataset
df.describe()
```

	value
count	2380.000000
mean	4.495147
std	3.404555
min	0.000000
25%	1.940000
50%	3.830000
75%	6.170000
max	22.280000

Top 10 rows:

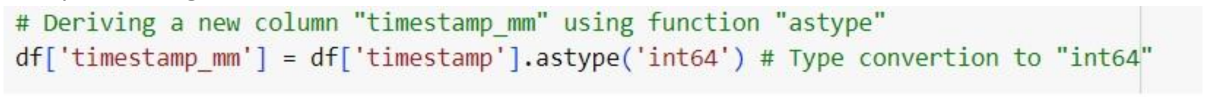


```
df.head(10)
```

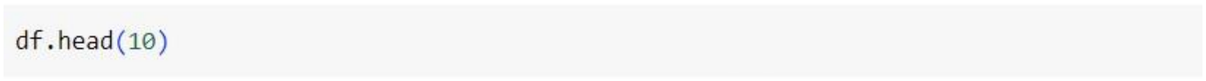


	timestamp	value
0	2015-09-01 13:45:00	3.06
1	2015-09-01 13:50:00	6.44
2	2015-09-01 13:55:00	5.17
3	2015-09-01 14:00:00	3.83
4	2015-09-01 14:05:00	4.50
5	2015-09-01 14:20:00	2.94
6	2015-09-01 14:25:00	2.28
7	2015-09-01 14:35:00	1.67
8	2015-09-01 14:40:00	18.83
9	2015-09-01 14:45:00	12.00

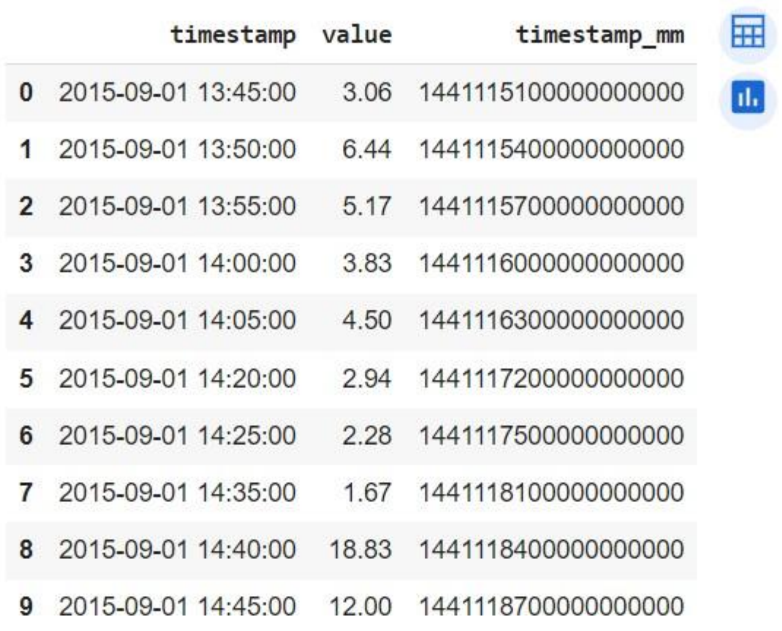
Preprocessing:



```
# Deriving a new column "timestamp_mm" using function "astype"
df['timestamp_mm'] = df['timestamp'].astype('int64') # Type convertion to "int64"
```



```
df.head(10)
```



	timestamp	value	timestamp_mm
0	2015-09-01 13:45:00	3.06	1441115100000000000
1	2015-09-01 13:50:00	6.44	1441115400000000000
2	2015-09-01 13:55:00	5.17	1441115700000000000
3	2015-09-01 14:00:00	3.83	1441116000000000000
4	2015-09-01 14:05:00	4.50	1441116300000000000
5	2015-09-01 14:20:00	2.94	1441117200000000000
6	2015-09-01 14:25:00	2.28	1441117500000000000
7	2015-09-01 14:35:00	1.67	1441118100000000000
8	2015-09-01 14:40:00	18.83	1441118400000000000
9	2015-09-01 14:45:00	12.00	1441118700000000000

Dataset info:

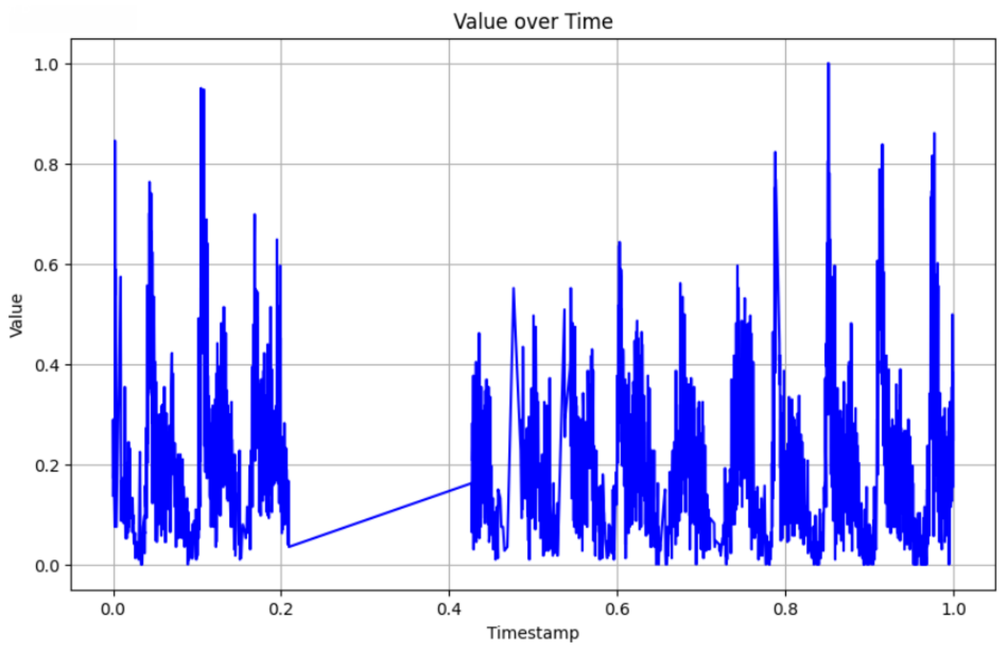
```
# Displaying preprocessed columns and its datatypes
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2380 entries, 0 to 2379
Data columns (total 2 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   value         2380 non-null   float64
 1   timestamp_mm  2380 non-null   int64  
dtypes: float64(1), int64(1)
memory usage: 37.3 KB
```

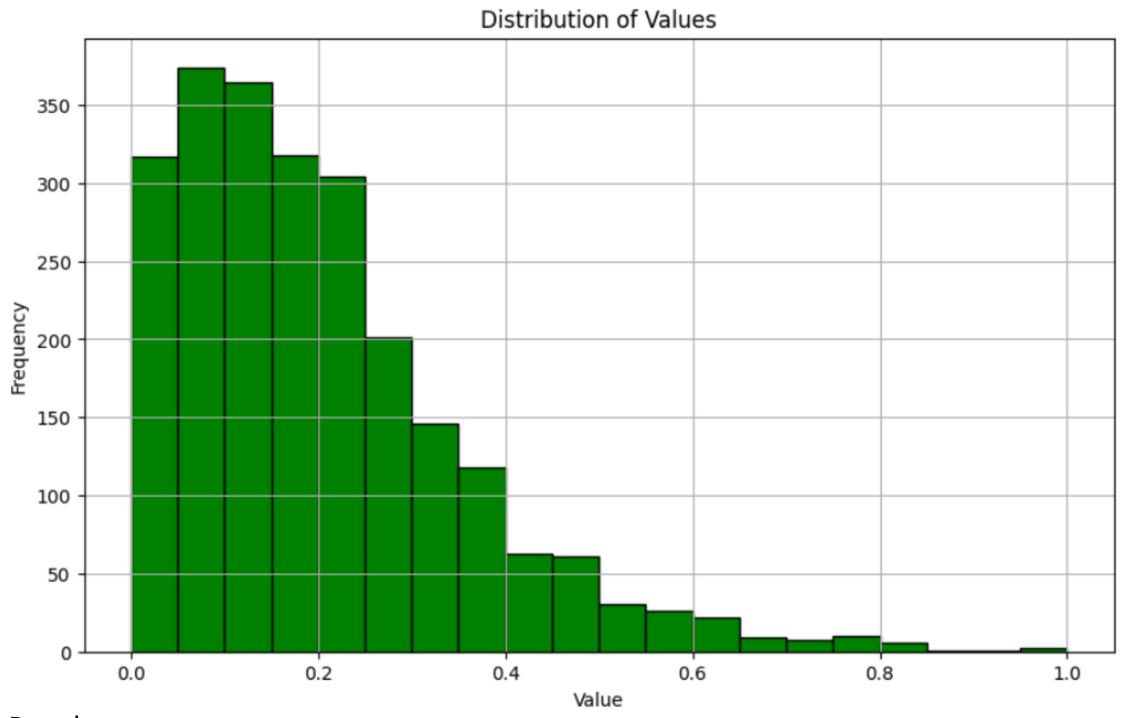
Normalization:

```
# Normalization of features
import pandas as pd

min_timestamp = df['timestamp_mm'].min()
max_timestamp = df['timestamp_mm'].max()

df['timestamp_mm'] = (df['timestamp_mm'] - min_timestamp) / (max_timestamp - min_timestamp)

min_value = df['value'].min()
max_value = df['value'].max()

df['value'] = (df['value'] - min_value) / (max_value - min_value)
```

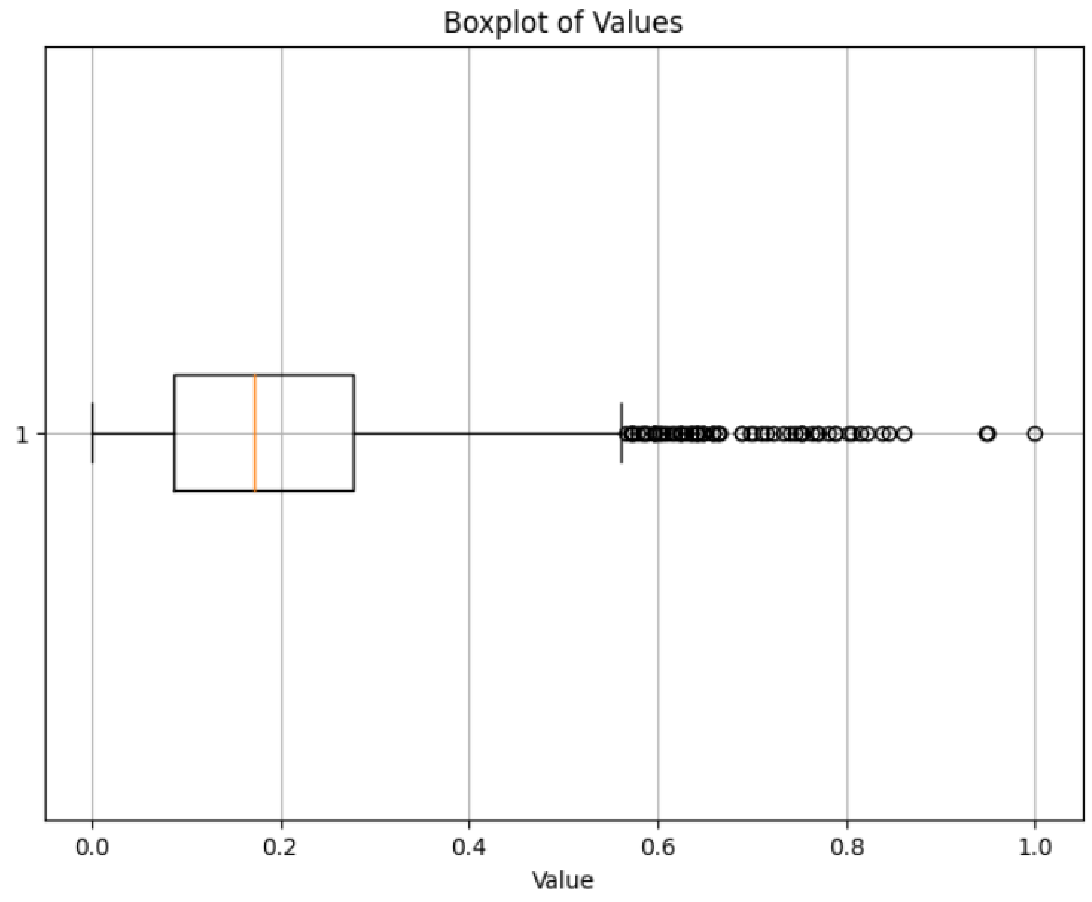
Data Visualization:



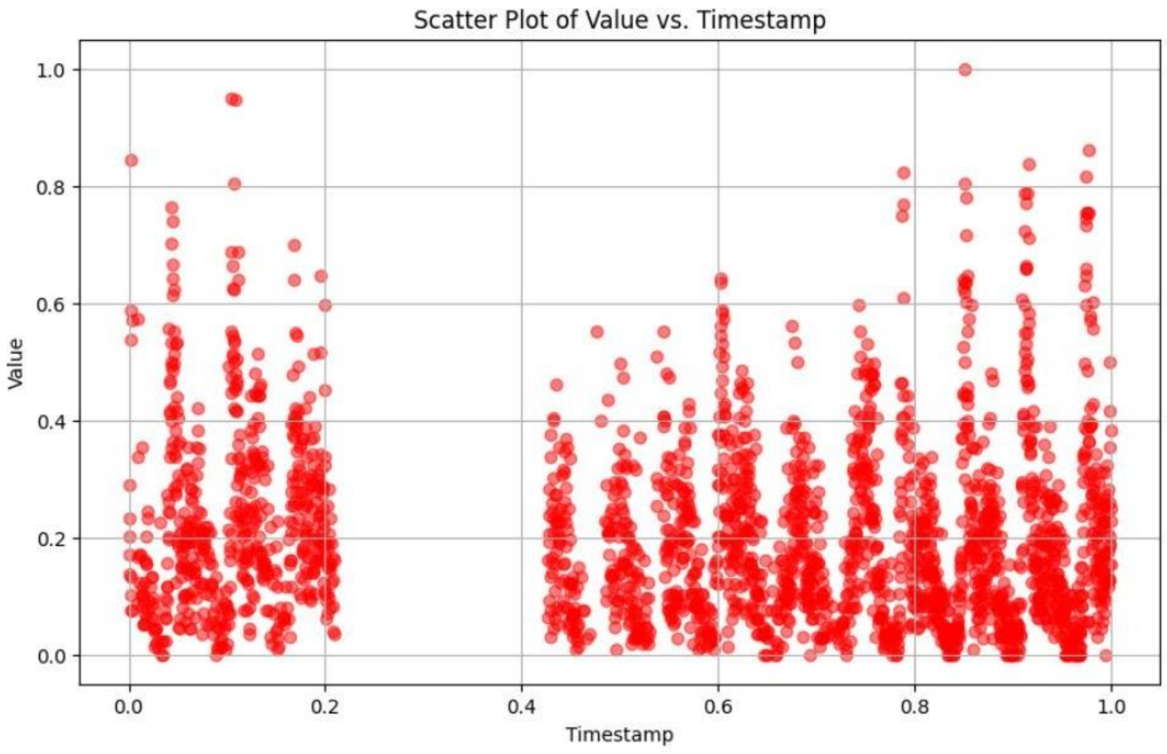
Histogram Plot:



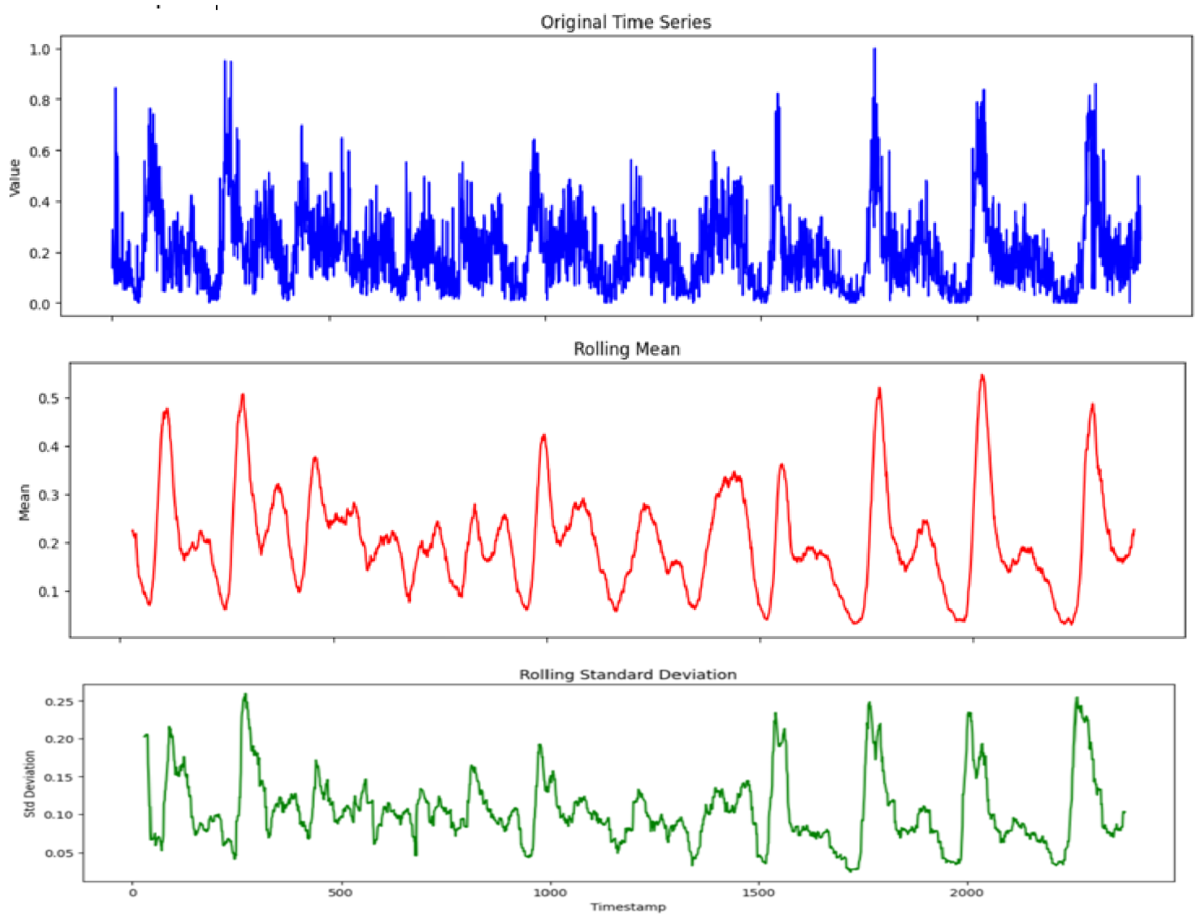
Boxplot:



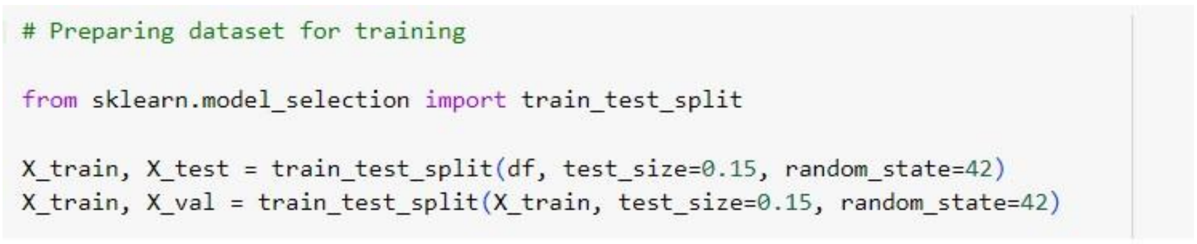
Scatter Plot:



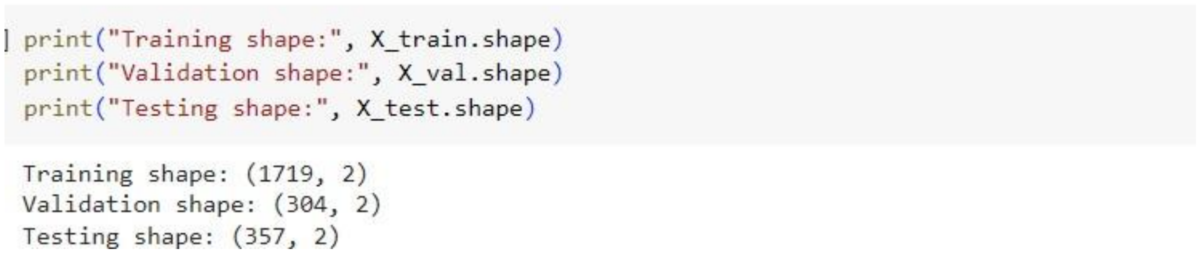
Time Series Plot:



Train Test Validation Split:



```
# Preparing dataset for training

from sklearn.model_selection import train_test_split

X_train, X_test = train_test_split(df, test_size=0.15, random_state=42)
X_train, X_val = train_test_split(X_train, test_size=0.15, random_state=42)
```

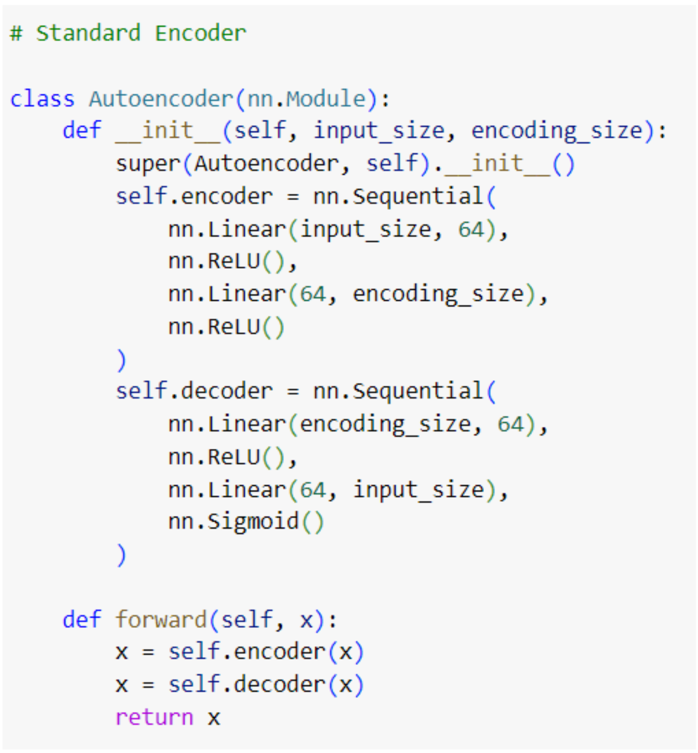


```
print("Training shape:", X_train.shape)
print("Validation shape:", X_val.shape)
print("Testing shape:", X_test.shape)
```

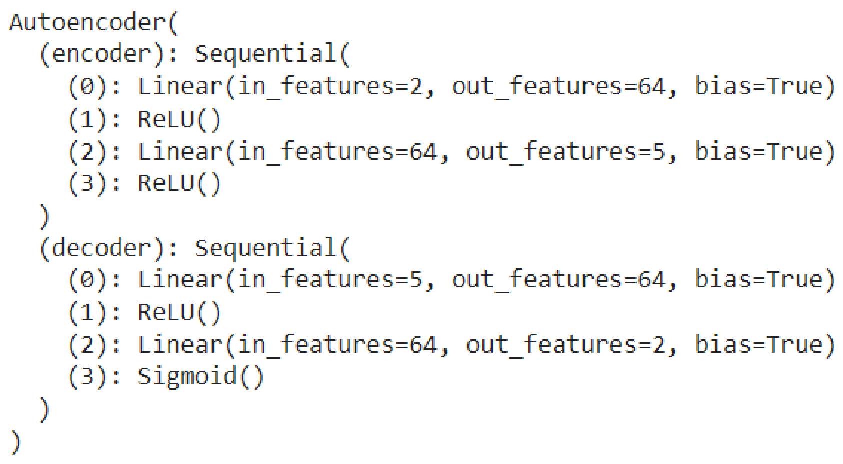
```
Training shape: (1719, 2)
Validation shape: (304, 2)
Testing shape: (357, 2)
```

Details of autoencoder:

Standard Autoencoder:



```
# Standard Encoder

class Autoencoder(nn.Module):
    def __init__(self, input_size, encoding_size):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 64),
            nn.ReLU(),
            nn.Linear(64, encoding_size),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(encoding_size, 64),
            nn.ReLU(),
            nn.Linear(64, input_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```



```
Autoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=2, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=5, bias=True)
    (3): ReLU()
  )
  (decoder): Sequential(
    (0): Linear(in_features=5, out_features=64, bias=True)
    (1): ReLU()
    (2): Linear(in_features=64, out_features=2, bias=True)
    (3): Sigmoid()
  )
)
```

The input_size and encoding_size parameters are used by the constructor (__init__) to initialize the autoencoder. The encoder and decoder components of the autoencoder are defined inside the constructor using nn.Sequential.

ReLU activation functions come after two completely connected (linear) layers that make up the encoder.

A size 64 layer with ReLU activation is the result of the first linear layer, which receives an input of size input_size.

Next, with ReLU activation, the second linear layer further decreases the dimensionality to encoding_size.

The decoder attempts to recover the original input by mirroring the encoder's structure in reverse.

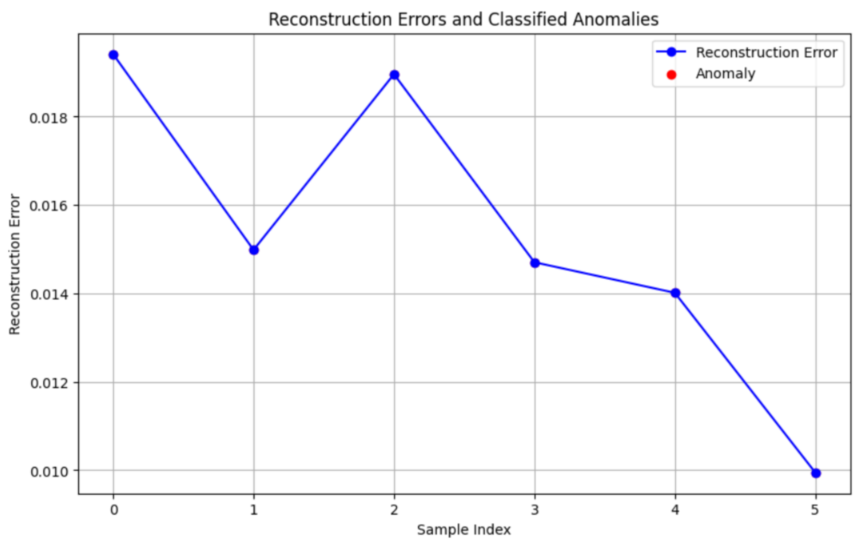
Using ReLU activation, it stretches the encoded representation of size, encoding_size, back to 64 dimensions.

With a sigmoid activation function, the second linear layer recreates the initial input size, guaranteeing values between 0 and 1.

Training and Validation Loss:

```
Epoch 1/10, Training Loss: 0.1112, Validation Loss: 0.0962
Epoch 2/10, Training Loss: 0.0838, Validation Loss: 0.0686
Epoch 3/10, Training Loss: 0.0602, Validation Loss: 0.0534
Epoch 4/10, Training Loss: 0.0508, Validation Loss: 0.0469
Epoch 5/10, Training Loss: 0.0439, Validation Loss: 0.0391
Epoch 6/10, Training Loss: 0.0360, Validation Loss: 0.0310
Epoch 7/10, Training Loss: 0.0282, Validation Loss: 0.0235
Epoch 8/10, Training Loss: 0.0214, Validation Loss: 0.0176
Epoch 9/10, Training Loss: 0.0169, Validation Loss: 0.0143
Epoch 10/10, Training Loss: 0.0146, Validation Loss: 0.0127
```

Reconstruction Errors and Classified Anomalies:



Performance on validation set:

```
val_losses = []
with torch.no_grad():
    for batch_data in val_loader:
        inputs = batch_data[0].to(device)
        outputs = autoencoder(inputs)
        loss = criterion(outputs, inputs)
        val_losses.append(loss.item())

average_val_loss = sum(val_losses) / len(val_losses)

print(f"Average Validation Loss: {average_val_loss:.4f}")
```

```
Average Validation Loss: 0.0126
```

LSTM Architecture:

```
# LSTM Architecture
class LSTMAutoencoder(nn.Module):
    def __init__(self, input_size, hidden_size, encoding_size):
        super(LSTMAutoencoder, self).__init__()
        self.encoder = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.decoder = nn.LSTM(hidden_size, input_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, encoding_size)

    def forward(self, x):
        _, (hidden, _) = self.encoder(x)
        encoded = self.fc(hidden.squeeze(0))
        decoded, _ = self.decoder(encoded.unsqueeze(1))
        return decoded
```

All neural network modules in PyTorch are derived from nn.Module, which is the base class from which the LSTMAutoencoder class comes.

The autoencoder is initialized with input_size, hidden_size, and encoding_size in the constructor (__init__).

Two LSTM layers are defined inside the constructor: an encoder LSTM and a decoder LSTM, both of which are configured to function on batches as the first dimension.

Sequences of input size input_size are fed into the encoder LSTM, which converts them into hidden representations of size hidden_size.

These hidden representations are then decoded back into sequences with the original input size by the decoder LSTM.

Furthermore, to further compress the encoded representation to the required encoding_size, a fully connected layer (fc) is defined.

During the forward pass, the forward technique establishes the data flow over the network.

The input sequences are represented by an input tensor x.

The encoder LSTM processes the input sequences.

To obtain the encoded representation (encoded), the fully connected layer (fc) passes the hidden state (hidden) that was obtained from the encoder.

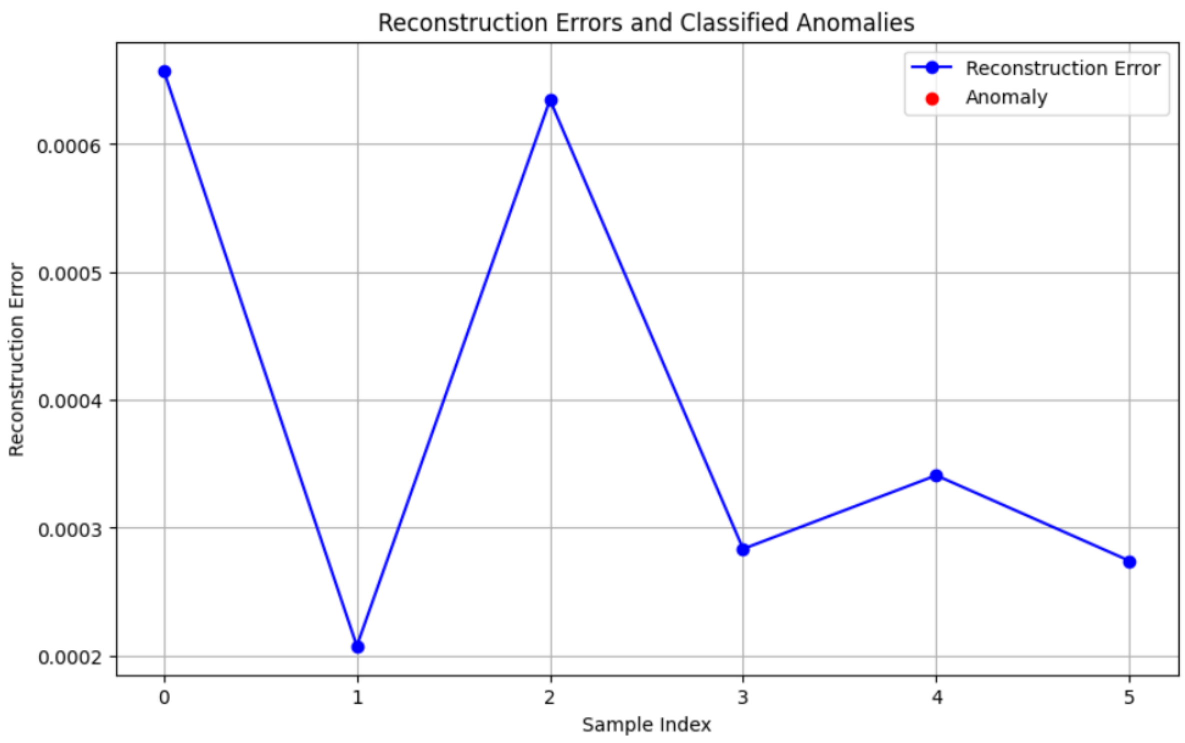
In order to recreate the input sequences, the encoded representation is subsequently run through an LSTM decoder.

Sequences that have been rebuilt are given back.

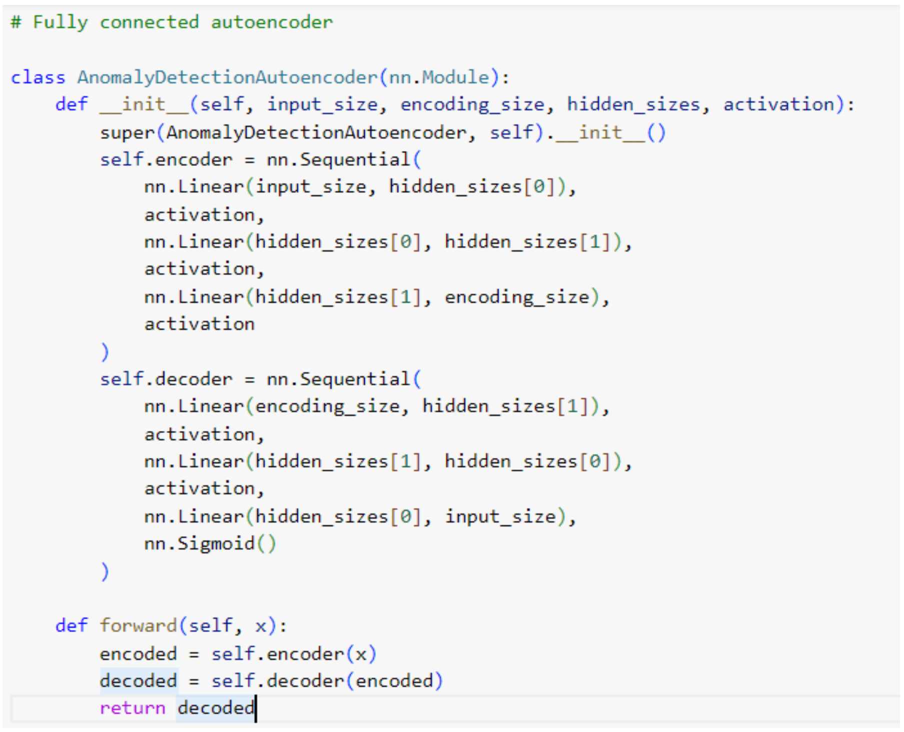
Training and Validation Loss:

```
Epoch [1/10], Train Loss: 0.0276, Val Loss: 0.0268
Epoch [2/10], Train Loss: 0.0219, Val Loss: 0.0231
Epoch [3/10], Train Loss: 0.0202, Val Loss: 0.0211
Epoch [4/10], Train Loss: 0.0176, Val Loss: 0.0173
Epoch [5/10], Train Loss: 0.0137, Val Loss: 0.0118
Epoch [6/10], Train Loss: 0.0083, Val Loss: 0.0057
Epoch [7/10], Train Loss: 0.0034, Val Loss: 0.0018
Epoch [8/10], Train Loss: 0.0011, Val Loss: 0.0006
Epoch [9/10], Train Loss: 0.0005, Val Loss: 0.0004
Epoch [10/10], Train Loss: 0.0005, Val Loss: 0.0004
```

Reconstruction Errors and Classified Anomalies:



Fully connected autoencoder:



```python
# Fully connected autoencoder

class AnomalyDetectionAutoencoder(nn.Module):
    def __init__(self, input_size, encoding_size, hidden_sizes, activation):
        super(AnomalyDetectionAutoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_size, hidden_sizes[0]),
            activation,
            nn.Linear(hidden_sizes[0], hidden_sizes[1]),
            activation,
            nn.Linear(hidden_sizes[1], encoding_size),
            activation
        )
        self.decoder = nn.Sequential(
            nn.Linear(encoding_size, hidden_sizes[1]),
            activation,
            nn.Linear(hidden_sizes[1], hidden_sizes[0]),
            activation,
            nn.Linear(hidden_sizes[0], input_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

The fundamental class for all neural network modules in PyTorch, nn.Module, is what the AnomalyDetectionAutoencoder class derives from.

The autoencoder is initialized with input_size, encoding_size, hidden_sizes, and activation in the constructor (__init__).

The nn.Sequential module, a container for sequentially running a sequence of layers, is used to define the encoder portion of the autoencoder.

It is composed of activation functions layered after a sequence of fully connected (linear) layers.

The hidden_sizes option controls the quantity and dimensions of hidden layers.

The input is compressed into the required encoding_size in the last layer.

The nn.Sequential module is also used to define the autoencoder's decoder portion.

It has fully connected layers that are followed by activation functions; its structure is similar to the encoder's, but it is reversed.

In order to guarantee that the output values are within the range [0, 1], the last layer employs the sigmoid activation function.

During the forward pass, the forward technique establishes the data flow over the network.

The input data is represented by an input tensor x.

To obtain the encoded representation (encoded), the input data is transmitted through the encoder.

After that, the decoder processes the encoded representation to rebuild the input data (decoded).

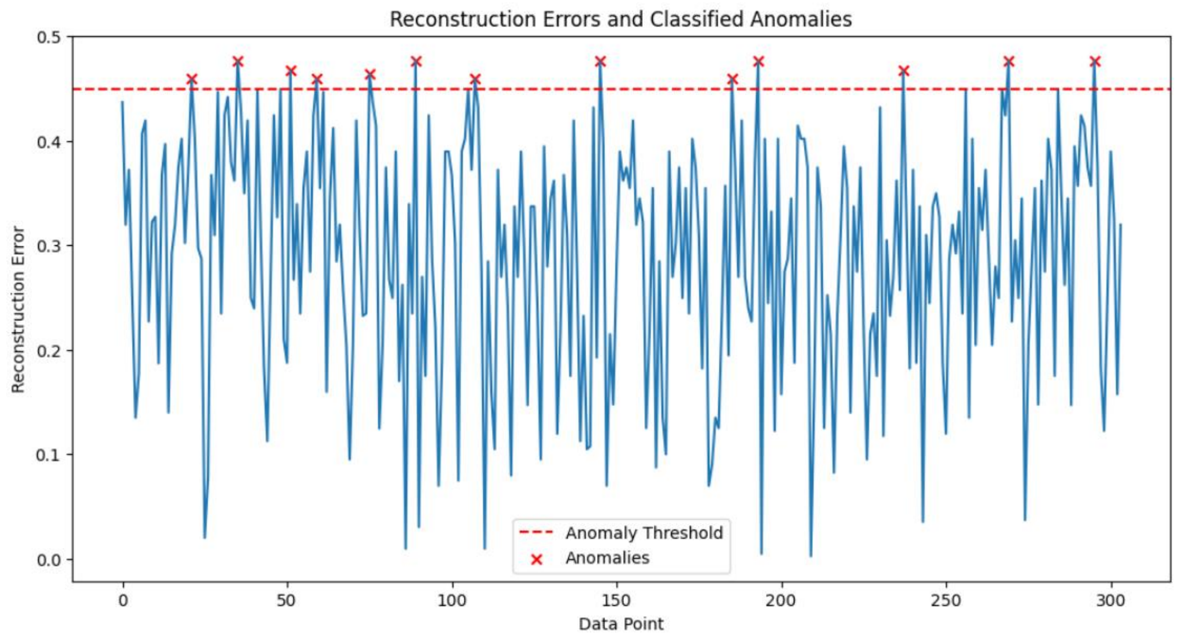
The data that was rebuilt is given back.

```
AnomalyDetectionAutoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=1, out_features=16, bias=True)
    (1): ReLU()
    (2): Linear(in_features=16, out_features=8, bias=True)
    (3): ReLU()
    (4): Linear(in_features=8, out_features=8, bias=True)
    (5): ReLU()
  )
  (decoder): Sequential(
    (0): Linear(in_features=8, out_features=8, bias=True)
    (1): ReLU()
    (2): Linear(in_features=8, out_features=16, bias=True)
    (3): ReLU()
    (4): Linear(in_features=16, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```

Training and Validation Losses:

```
Epoch [1/10], Training Loss: 0.0996, Validation Loss: 0.0968
Epoch [2/10], Training Loss: 0.0997, Validation Loss: 0.0968
Epoch [3/10], Training Loss: 0.0997, Validation Loss: 0.0968
Epoch [4/10], Training Loss: 0.0997, Validation Loss: 0.0968
Epoch [5/10], Training Loss: 0.0996, Validation Loss: 0.0968
Epoch [6/10], Training Loss: 0.0997, Validation Loss: 0.0968
Epoch [7/10], Training Loss: 0.0998, Validation Loss: 0.0968
Epoch [8/10], Training Loss: 0.0997, Validation Loss: 0.0968
Epoch [9/10], Training Loss: 0.0997, Validation Loss: 0.0968
Epoch [10/10], Training Loss: 0.0997, Validation Loss: 0.0968
```

Reconstruction Errors and Classified Anomalies:



Performance on validation set:

```
# Performance on validation set

fc_autoencoder.eval()
val_loss = 0

with torch.no_grad():
    for data in val_loader:
        inputs = data[0].to(device)
        outputs = fc_autoencoder(inputs)
        loss = criterion(outputs, inputs)
        val_loss += loss.item() * inputs.size(0)

average_val_loss = val_loss / len(val_loader.dataset)
print(f'Average Validation Loss: {average_val_loss:.4f}')
```

```
Average Validation Loss: 0.0964
```

FinalAutoencoder:

```
class FinalAutoencoder(nn.Module):
    def __init__(self, input_size, encoding_size, hidden_sizes, activations):
        super(FinalAutoencoder, self).__init__()
        assert len(hidden_sizes) == len(activations), "Length of hidden_sizes must match length of activations"

        encoder_layers = []
        decoder_layers = []

        # Encoder layers
        prev_hidden_size = input_size
        for i in range(len(hidden_sizes)):
            encoder_layers.append(nn.Linear(prev_hidden_size, hidden_sizes[i]))
            encoder_layers.append(activations[i])
            prev_hidden_size = hidden_sizes[i]

        encoder_layers.append(nn.Linear(hidden_sizes[-1], encoding_size))
        self.encoder = nn.Sequential(*encoder_layers)

        # Decoder layers
        decoder_layers.append(nn.Linear(encoding_size, hidden_sizes[-1]))
        decoder_layers.append(activations[-1])

        for i in range(len(hidden_sizes) - 1, 0, -1):
            decoder_layers.append(nn.Linear(hidden_sizes[i], hidden_sizes[i-1]))
            decoder_layers.append(activations[i-1])

        decoder_layers.append(nn.Linear(hidden_sizes[0], input_size))
        decoder_layers.append(nn.Sigmoid())
        self.decoder = nn.Sequential(*decoder_layers)

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

Class Definition: A class called FinalAutoencoder has been defined. It derives from nn.Module, which is the base class of all PyTorch neural network modules.

__init__ (Constructor Method):

The autoencoder is initialized with the following parameters by the constructor:
input_size: The amount of data that was entered.

encoding_size: The compressed representation's size during encoding.

hidden_sizes: A list that describes the encoder and decoder's hidden layer sizes.

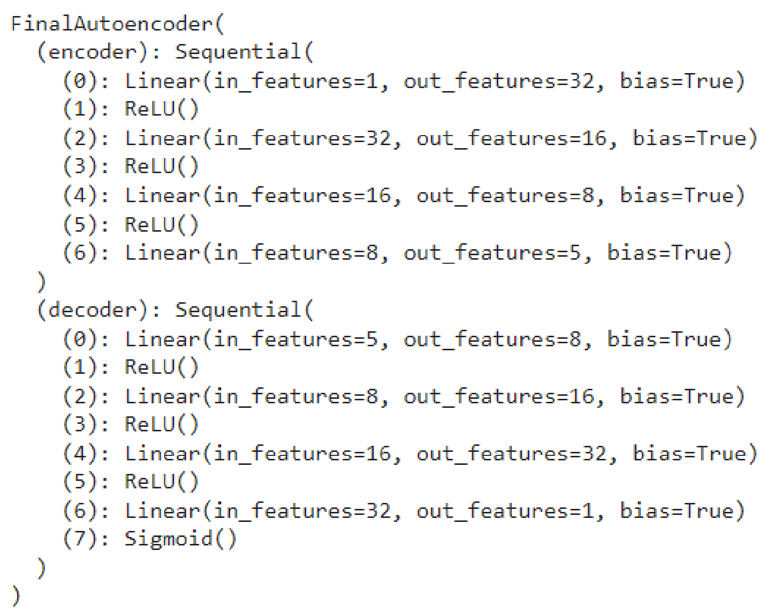
activations: A list that details the interlayer activation functions to be applied.

Inside the builder:

The encoder is described as a sequential module (nn.Sequential) with linear layers and activation functions in order of precedence.

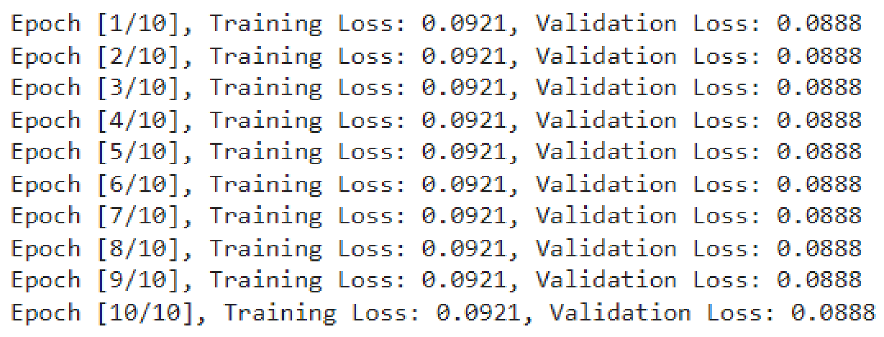
Another definition of the decoder is a sequential module that has the same structure as the encoder but a different order.

The sigmoid activation function is used in the decoder's last layer to compress the output values into the interval [0, 1].



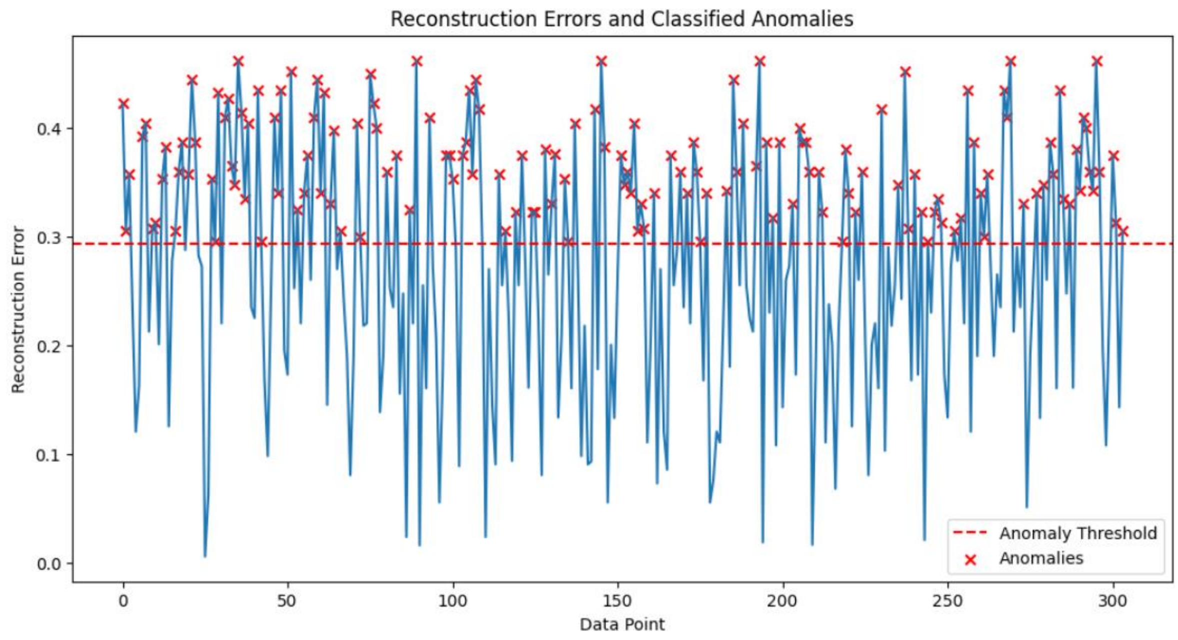
```
FinalAutoencoder(
  (encoder): Sequential(
    (0): Linear(in_features=1, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=16, bias=True)
    (3): ReLU()
    (4): Linear(in_features=16, out_features=8, bias=True)
    (5): ReLU()
    (6): Linear(in_features=8, out_features=5, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=5, out_features=8, bias=True)
    (1): ReLU()
    (2): Linear(in_features=8, out_features=16, bias=True)
    (3): ReLU()
    (4): Linear(in_features=16, out_features=32, bias=True)
    (5): ReLU()
    (6): Linear(in_features=32, out_features=1, bias=True)
    (7): Sigmoid()
  )
)
```

Training and Validation Losses:



```
Epoch [1/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [2/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [3/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [4/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [5/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [6/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [7/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [8/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [9/10], Training Loss: 0.0921, Validation Loss: 0.0888
Epoch [10/10], Training Loss: 0.0921, Validation Loss: 0.0888
```

Reconstruction Errors and Classified Anomalies



Evaluation on validation set

```
# Evaluation on validation set

final_autoencoder.eval()

val_losses = []

with torch.no_grad():
    for batch_data in val_loader:
        inputs = batch_data[0].to(device)
        outputs = final_autoencoder(inputs)
        loss = criterion(outputs, inputs)
        val_losses.append(loss.item())

average_val_loss = np.mean(val_losses)
print(f'Average Validation Loss: {average_val_loss:.4f}')
```

```
Average Validation Loss: 0.0893
```

Final Best Model Result:

Training and Validation losses over 10 epochs:

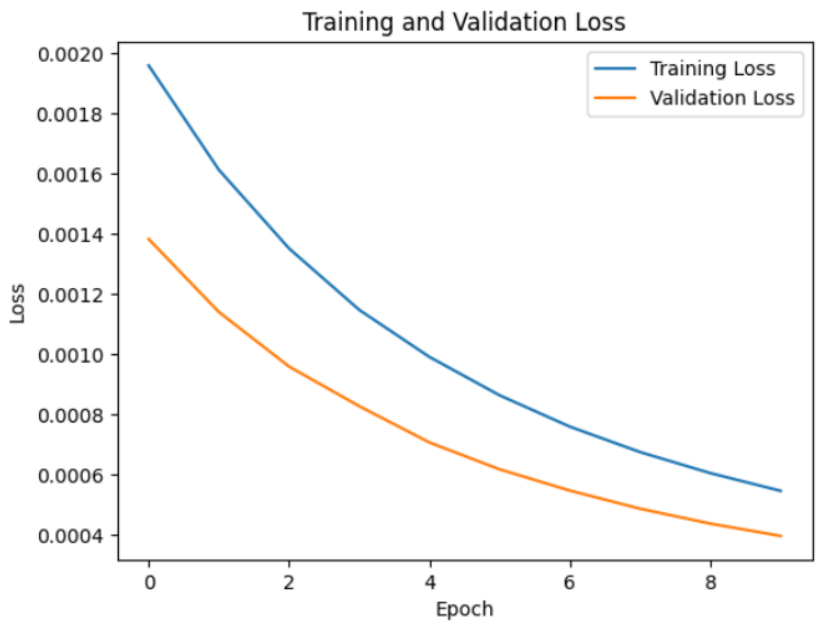
```
Epoch 1/10, Training Loss: 0.0020, Validation Loss: 0.0014
Epoch 2/10, Training Loss: 0.0016, Validation Loss: 0.0011
Epoch 3/10, Training Loss: 0.0014, Validation Loss: 0.0010
Epoch 4/10, Training Loss: 0.0011, Validation Loss: 0.0008
Epoch 5/10, Training Loss: 0.0010, Validation Loss: 0.0007
Epoch 6/10, Training Loss: 0.0009, Validation Loss: 0.0006
Epoch 7/10, Training Loss: 0.0008, Validation Loss: 0.0005
Epoch 8/10, Training Loss: 0.0007, Validation Loss: 0.0005
Epoch 9/10, Training Loss: 0.0006, Validation Loss: 0.0004
Epoch 10/10, Training Loss: 0.0005, Validation Loss: 0.0004
```

Testing set loss:

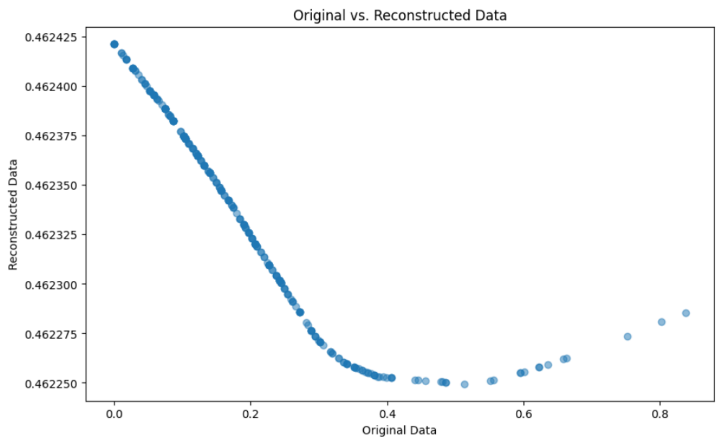
```
# Testing loss
autoencoder.eval()
test_loss = 0.0

with torch.no_grad():
    for batch_data in test_loader:
        inputs = batch_data[0].to(device)
        outputs = autoencoder(inputs)
        loss = criterion(outputs, inputs)
        test_loss += loss.item() * inputs.size(0)

# Calculating average testing loss
test_loss /= len(test_loader.dataset)

# Print testing loss
print(f'Testing Loss: {test_loss:.4f}')
```

```
Testing Loss: 0.0004
```

Training and validation losses:



Original vs. Reconstructed Data



Because autoencoders can learn representations of normal data and identify deviations from this norm, they have become increasingly popular for anomaly identification. They do, however, have advantages and disadvantages like any other method:

Strengths:

- Unsupervised Learning: Labeled instances of anomalies are not necessary for autoencoders to acquire the ability to represent data. This makes them appropriate for situations in which it is difficult or costly to get labeled abnormalities.

- Non-linear Transformations: Autoencoders are able to effectively model non-linear patterns and anomalies because they are able to capture complicated correlations in the data.

- Adaptability: By modifying the architecture, loss function, and training approach, autoencoders may be made to work with a variety of data formats and anomaly detection applications.

- Feature Learning: Autoencoders can acquire meaningful features that are applicable to tasks beyond anomaly detection by compressing data into a lower-dimensional latent space.

Limitations:

- Normality Assumption: Because autoencoders are trained on normal data, they could find it difficult to discern between typical distributional fluctuations and uncommon anomalies. They might mark as anomalies odd but harmless patterns.

- Sensitivity to Training Data: Anomalies that differ from the training data may go unnoticed by autoencoders because of their susceptibility to the distribution of training data.

- Limited Generalization: When faced with data distributions that deviate greatly from the training set or unknown anomalies, autoencoders might not be able to adapt well. They might not be able to identify new kinds of anomalies that weren't there during training.

- Interpretability: It can be hard to grasp why some cases are marked as anomalies because of the latent space that autoencoders have learned.

When there is a lack of labeled anomaly data, autoencoders present a viable method for anomaly detection. However, a number of variables, like the representativeness and quality of the training data, the architecture and hyperparameter selection, and the type of anomalies being identified, affect how effective they are. It's critical to thoroughly assess autoencoder-based anomaly detection systems and take into account both their advantages and disadvantages in particular application scenarios.