Deusto
Universidad de Deusto

DEEP LEARNING
3rd ASSIGNMENT

ANOMALY DETECTION IN STOCK PRICES

https://github.com/InigoPena/Anomaly_Detection_in_Stock_Prices

MATEO PÉREZ SUÁREZ - 79439806Z
IÑIGO PEÑA DE LAS HERAS - 20983346D

CONTENTS

INTRODUCTION AND OBJECTIVE

In this third deliverable we are focused on anomaly detection in time series task. Our goal is to find outliers in stock values, as this anomalys not labeled in our data, we decided to detect anomalies by training the models to reconstruct the time series and after it compare them with the real function flagging points as anomalies when the reconstruction error exceeds a defined threshold.

THE DATASET

We used taken a sample of historical time series data (2005–2025) from Yahoo Finance, which is a public platform for financial market data in order to take the datasets we are using to solve our tasks.

Our data separated into 5 different datasets covering Apple, Tesla, Nvidia, Google, Microsoft and Salesforce for our anomaly detection deep learning project.

Date	Open	High	Low	Close	Volume	Dividends	Stock Splits
2005-05-23 00:00:00-04:00	17.905448490126158	18.13500547101993	17.905448490126158	17.98196792602539	75421100	0.0	0.0
2005-05-24 00:00:00-04:00	17.94718203017148	18.00283215558272	17.891531904760235	17.91240119934082	61287700	0.0	0.0
2005-05-25 00:00:00-04:00	17.86370120767081	17.9263077425372	17.738488137938035	17.88456916809082	35749000	0.0	0.0
2005-05-26 00:00:00-04:00	17.912404589301808	18.08631142997464	17.898491723615045	18.016748428344727	50579200	0.0	0.0
2005-05-27 00:00:00-04:00	17.968044196831375	18.148907364370434	17.95413133929688	18.134994506835938	54978000	0.0	0.0

The dataset contains daily stock market data for each company, structured with columns:

- **Date**: The trading day timestamp.
- **Open**: The price at which the stock opened on that day.
- **High**: The highest price reached during the trading session.
- **Low**: The lowest price reached during the trading session.
- **Close**: The final trading price of the stock for that day.
- **Volume**: The number of shares traded during the day.
- **Dividends**: Cash dividends issued on that date (mostly 0 in our data).
- **Stock Splits**: Split ratio applied to the stock (also mostly 0).

DATA PREPROCESSING

To prepare the raw financial time series data for deep learning, we applied a preprocessing pipeline designed to transform the original daily stock values into normalized, fixed-length input sequences suitable for model training.

Each stock dataset (Apple, Tesla, Nvidia, Google, Microsoft, and Salesforce) was first normalized using Min-Max normalization. We selected five key numerical features per day:

- Close_Normalized
- Open_Normalized
- High_Normalized
- Low_Normalized
- Volume_Normalized

These normalized features were used to build multivariate sequences using a sliding window approach. Each input sample consists of 30 consecutive trading days (window_size = 30), forming a tensor of shape (30, 5) per sequence.

The motivation behind this transformation is to:

- Provide the model with a temporal context of one month for each prediction
- Ensure uniform input shape for batch processing.
- Reduce variability between different stocks by working with normalized values.

The sequences were then flattened and saved into CSV files, one per company, to be loaded efficiently during model training.

Subsequently, during training, we load these preprocessed CSVs for each ticker and reshape them back into 3D tensors of shape (num_sequences, 30, 5), where 30 is the time window length.and 5 is the number of input features per day.

LTSM Autoencoder MODEL (ARCHITECTURE)

Our LSTM Autoencoder is designed to compress multivariate time series data (30 days × 5 features) into a lower-dimensional representation and then reconstruct it. As an autoencoder, the model has two main components:

1. Encoder

- Input: Sequences of shape (64, 30, 5), meaning 5 normalized features over 30 days separated in batches of 64 days.
- LSTM Layer: Converts the input into a hidden state of size 64, capturing dependencies.
- Linear Layer: Projects the last hidden state to a compact latent vector of size 32 (latent_size), serving as the learned representation.

2. Decoder

- Linear Layer: Expands the latent_size (32) back to hidden_size (64).
- Repeat: The expanded vector is repeated over 30 timesteps to match the length of input sequence because even if it is the same vector, as the model has memorie, it interpretes it well.
- LSTM Layer: Decodes the repeated sequence, modeling temporal structure in reconstruction.
- Output Layer: Final linear layer maps back from hidden_size (64) to the original input_size (5) for every timestep.

```python
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, latent_size):
        super().__init__()

        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.linear = nn.Linear(hidden_size, latent_size)

    def forward(self, x):
        _, (h_n, _) = self.lstm(x)
        latent = self.linear(h_n[-1])
        return latent
```

```python
class Decoder(nn.Module):
    def __init__(self, latent_size, hidden_size, output_size, seq_len):
        super().__init__()
        self.seq_len = seq_len
        self.linear = nn.Linear(latent_size, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.output_layer = nn.Linear(hidden_size, output_size)

    def forward(self, latent):
        hidden = self.linear(latent)
        repeated = hidden.unsqueeze(1).repeat(1, self.seq_len, 1)
        decoded, _ = self.lstm(repeated)
        out = self.output_layer(decoded)
        return out
```

LTSM Autoencoder MODEL (TRAINING PROCESS)

We trained a separate LSTM Autoencoder for each stock (Apple, Tesla, Nvidia, Google, Microsoft) using normalized time series data. The training process involved the following choices:

Data Handling

- Each stock was reshaped to multivariate sequences of shape 30 timesteps × 5 features.
- Datasets were loaded using PyTorch DataLoader with a batch size of 64.

Optimizer

- Adam optimizer was selected with a learning rate of 0.001.
- Adam is well-suited for time series tasks because it adapts the learning rate individually for each parameter, keeping astable and efficient convergence.

Loss Function

- MSELoss (Mean Squared Error) was used to minimize the difference between the input and the reconstructed output.
- MSE is suitable for reconstruction-based anomaly detection, as high error indicates deviations from learned patterns.

Training Settings

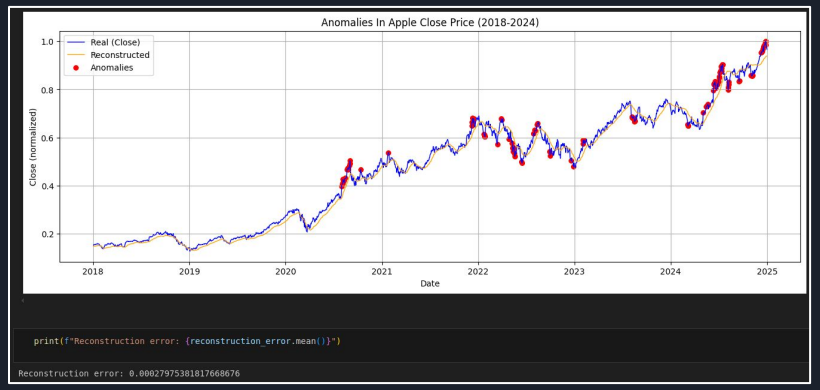
- 10 epochs per stock balanced overfitting.
- Models trained on GPU when available to accelerate learning.
- Loss per epoch was recorded and saved as plots.

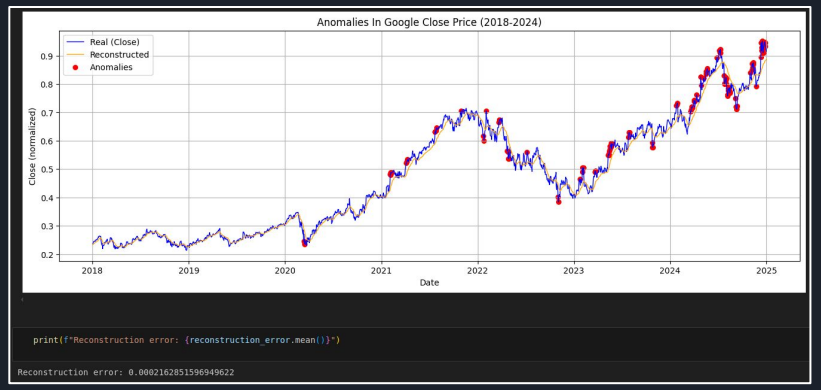
LTSM Autoencoder MODEL (2 RESULTS)

We compared the reconstructed Close price to the actual value. Anomalies were identified using a threshold based on the 97th percentile of the reconstruction error.

Apple (AAPL)

- Reconstruction closely follows the real price curve from 2018 to 2024.
- Many anomalies detected during market volatility (e.g., COVID-19 crash, 2022 tech correction).
- Average reconstruction error: *very low*, indicating strong model fit.





Google (GOOGL)

- Reconstruction is slightly less aligned in highly volatile regions.
- Anomalies appear in different time windows compared to AAPL.
- Slightly higher reconstruction error, meaning there might be more complex price patterns.

LTSM Autoencoder MODEL (2 RESULTS)

- The curve is monotonic and consistent, indicating a clean and efficient convergence.
- Suggests that Apple's price data is easier to model (less volatility or noise).

- Also shows a steep initial drop, but from epoch 6 onward, the loss curve flattens and oscillates slightly.
- Small fluctuations appear in the last epochs (minor increase at epoch 9).
- This might reflect a more complex or volatile price structure, making reconstruction harder.

Gated Recurrent Unit (GRU) Autoencoder

We implemented and trained a GRU-based Autoencoder to detect anomalies in general multivariate stock price sequences.

Architecture

- Encoder: 2-layer GRU (input_size=5, hidden_size=64), captures temporal patterns.
- Decoder: 2-layer GRU with identical hidden_size=64, reconstructs the original input.
- Output Layer: Linear projection to input_size=5 (Open, High, Low, Close, Volume).
- Input format: sequences of shape (30 timesteps × 5 features).

```python
class Encoder(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, batch_first=True, dropout=0.2):
        super(Encoder, self).__init__()
        self.hidden_size = hidden_size
        self.gru = nn.GRU(input_size, hidden_size, num_layers=num_layers, batch_first=batch_first, dropout=dropout)

    def forward(self, x):
        output, hidden = self.gru(x)
        return output, hidden
```

```python
class Decoder(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, batch_first=True, dropout=0.2):
        super(Decoder, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size, num_layers=num_layers, batch_first=batch_first, dropout=dropout)

    def forward(self, x, hidden):
        output, hidden = self.gru(x, hidden)
        return output, hidden
```

Training Setup

- Trained on a combined dataset of AAPL, TSLA, GOOGL, MSFT, and NVDA.
- MSELoss, reconstruction-based anomaly detection.
- Adam optimizer (lr=0.0005), chosen for stable convergence in time series.
- 10 Epochs and batch size of 64
- Training loss saved and visualized (log-scale).

```python
print(f"Training model...")

loss_history = []

for epoch in range(10):
    model.train()
    total_loss = 0
    for batch in dataloader:
        batch = batch[0].to(device)
        output = model(batch)
        loss = loss_fn(output, batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    epoch_loss = total_loss / len(dataloader)
    loss_history.append(epoch_loss)
    print(f"Epoch {epoch+1}/10, Loss: {total_loss/len(dataloader):.6f}")

current_dir = Path(__file__).resolve().parent
performance_dir = current_dir / "model_performances"
```

Gated Recurrent Unit (GRU) Autoencoder

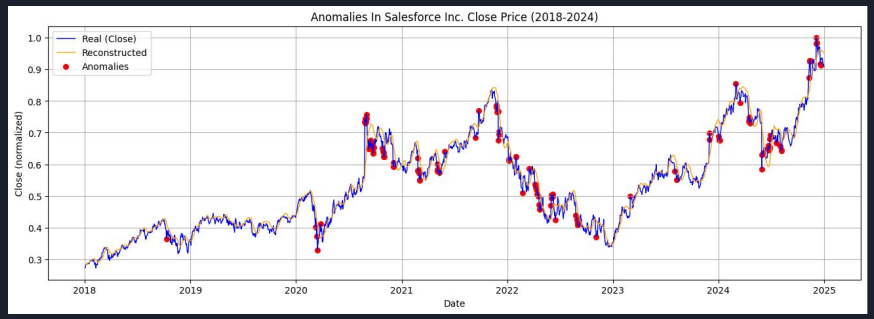
We evaluated the general GRU Autoencoder on Salesforce (CRM) stock data and compared its performance before and after fine-tuning it with this stocks data.

Before Fine-Tuning

- The model, trained on other tech stocks, already reconstructed Salesforce data well.
- Most sequences had low reconstruction error.
- Anomalies detected using the 97th percentile threshold mostly aligned with high-volatility events.
- It had a low and stable reconstruction loss, indicating that the model was already well generalizing unseen data
- Reconstruction Error Mean: 0.00029920

After Fine-Tuning

- We applied fine-tuning on Salesforce data using:
  - 5 additional epochs
  - Lower learning rate (0.0001)
  - ReduceLROnPlateau scheduler
- The improvement was marginal, showing slightly more consistent reconstruction.
- The anomaly pattern remained largely the same.
- Reconstruction Error Mean: ≈ 0.00175



Potential Improvements on LSTM & GRU Autoencoders

- While both the LSTM and GRU Autoencoders gave us solid results, there's definitely room to improve. One idea would be to try attention mechanisms or even bidirectional RNNs, which could help the model better understand which parts of the sequence matter most. We could also experiment with Transformer models, especially since they're known for handling long-term patterns really well.

- On the training side, we might get better performance by training a bit longer, but using early stopping to avoid overfitting. It could also help to adjust the learning rate dynamically (something like Cosine Annealing) and apply gradient clipping to keep things stable during training.

- Last, it would be interesting to expand the dataset with stocks from other sectors to make the model more general. We could also try injecting artificial anomalies to see how the model reacts in more extreme situations. And if we had labeled data, we'd be able to measure things like precision and recall to get a clearer picture of how well the models are actually working.

- Overall, working with autoencoders has shown us how powerful they can be for spotting unusual patterns in time series data — even without labeled examples. They're flexible, relatively lightweight, and once trained, they do a good job of capturing what "normal" looks like for a given stock. That said, they're not perfect out of the box. Their performance really depends on the quality of the input data and how well the model architecture fits the specific behavior of the stock. Still, they offer a strong foundation for building more advanced anomaly detection systems, and there's a lot of potential to keep improving them with smarter training strategies and better evaluation techniques.

TRANSFORMERS AUTOENCODER MODEL (ARCHITECTURE)

1. Input Projection: The first step into this architecture is a single linear layer which projects the 5-Dimensional input into a higher dimensional feature space to prepare it for the next transformer layers.
   a. We have chosen a size of 64 as the dataset isn't that big and bigger could end up in model overfitting.
2. Positional Encoding: As transformers have no inherent notion of order in sequences, we include learnable positional encodings. Which add to the projected inputs to inject information about the position of each timestep.
   a. We considered using sinusoidal encodings, but we find out that learnable encodings give the model more flexibility to optimize positional information specifically for our task.

```python
class TransformerAutoencoder(nn.Module):  5 usages  MateoPerezSuarez *
    def __init__(self, seq_len=30, d_model=64, nhead=4, num_layers=2, input_dim=5):  MateoPerezSuare
        super(TransformerAutoencoder, self).__init__()
        self.seq_len = seq_len
        self.d_model = d_model

        self.input_projection = nn.Linear(input_dim, d_model)

        self.positional_encoding = nn.Parameter(torch.randn(1, seq_len, d_model))

        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=nhead, batch_first=True)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)

        decoder_layer = nn.TransformerDecoderLayer(d_model=d_model, nhead=nhead, batch_first=True)
        self.decoder = nn.TransformerDecoder(decoder_layer, num_layers=num_layers)

        self.output_projection = nn.Linear(d_model, input_dim)
```

```python
def forward(self, src):  MateoPerezSuarez *
    x = self.input_projection(src)
    x = x + self.positional_encoding[:, :x.size(1), :]

    memory = self.encoder(x)
    out = self.decoder(x, memory)

    out = self.output_projection(out)
    return out
```

TRANSFORMERS MODEL (ARCHITECTURE)

1. Transformer Encoder: We build it with two layers of multi-head self-attention and feed-forward networks, by using this stacked structure allows the model to progressively capture more abstract and complex patterns across the input sequence.
    a. Multi-head self-attention layer: Each time step can attend to all others in the sequence, enabling the model to capture both short- and long-range dependencies. We used 4 attention heads, allowing the model to focus on different representation subspaces in parallel, which improves its ability to recognize diverse patterns.
    b. Feed-forward network: After the attention mechanism, we decided to implement a position-wise feed-forward network applies non-linear transformations independently at each time step which we considered could enrich the learned features and increases model expressiveness.
2. Transformer Decoder: Also consists of two layers, each containing mechanisms to process and reconstruct the input sequence by referencing both the original input and the encoder's output:
    a. Self-attention layer: While our decoder receives the same input as the encoder, including self-attention allows each position to integrate information from all others
    b. Cross-attention allows the decoder to leverage the encoder's latent representation, guiding accurate reconstruction.
3. Output Projection: We set up a final linear layer that maps the decoder output (64 dimensions) back to the original 5-dimensional feature space, with it producing the reconstructed input sequence.

TRANSFORMERS MODEL (TRAINING PROCESS)

In order to train this model, we followed a relatively a similar process as for LTSM as the final task an input are the same we thought that using the same that worked before could be perfectly functional:

Data Handling

- Each stock was reshaped to multivariate sequences of shape 30 timesteps × 5 features.
- We loaded data using batch size of 64, with this we allowed a more efficient mini-batch training

Optimizer

- Adam optimizer was selected with a learning rate of 0.001.
- We used this as it adapts learning rates per parameter, which could promote stable convergence and we research that could be interesting with deep architecture like transformers

Loss Function

- We decided to use MSELoss as we considered it should help minimizing the reconstruction error between the input and output.
- Also looking into different forums we found is an ideal loss for reconstruction tasks as ours.

Training Settings

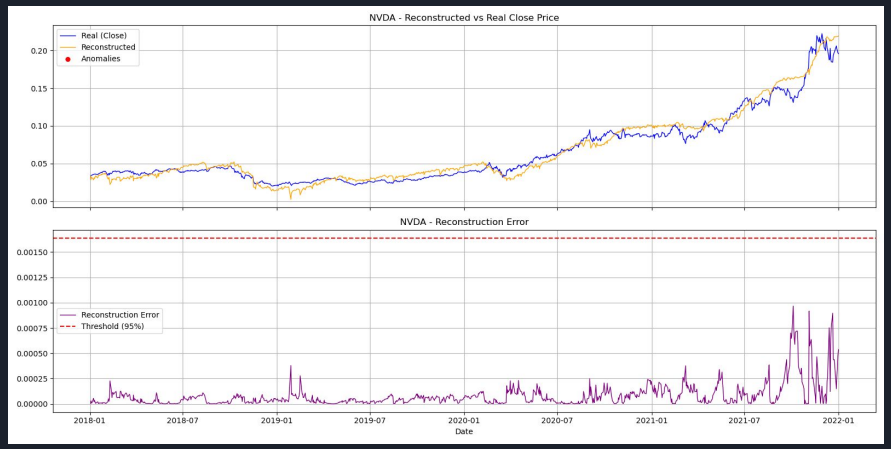
- 10 epochs per stock balanced overfitting even if the results aren't great, we started with 20 which resulted in weird results probably caused by overfitting
- Models trained on GPU when available to accelerate learning.

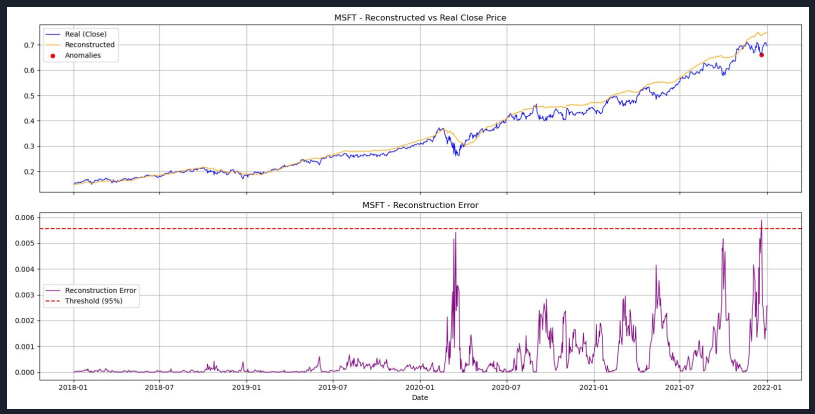
TRANSFORMERS MODEL (2 RESULTS)

For the comparison anomalies were identified using a threshold based of .95 as we considered that to be an anomaly the difference between the reconstructed function and the real should be very high.

NVIDIA

- Reconstruction closely follows the real price curve from 2018 to 2022.
- Using transformers model, the reconstruction didn't find any anomaly as result.
- Average reconstruction error: *low*, even if at the end it grows exponentially. It never surpases to the threshold





MICROSOFT (MSFT)

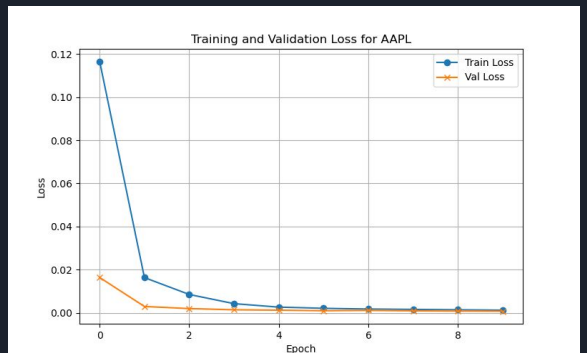
- Reconstruction is slightly less aligned in highly volatile regions.
- The graph is close to the anomaly in more periods of time than in NVIDIA
- For this company we obtained slightly higher reconstruction error, this means that something external and more complex could have affected the price

TRANSFORMERS MODEL (IMPROVEMENT ATTEMPTS)

In order to improve different aspects of the model, we tried two apply two main things:

SHAPE IT FOR AVOID OVERFITTING

- We started training models with 20 epochs which didn't work at all, as the function from epoch 10 forwards started to stop reducing, so we considered that reducing epoch we could avoid overfitting from the model.
- Another possible improvement we tested was the addition of dropout which didn't work as expected, the loss function showed exactly the same results so we interpret that it wasn't necessary.



TRY ANOTHER TRANSFORMER ARCHITECTURE

- We included different features like the use of sinusoidal functions
- Using a 1-Dimensional input layer into a higher dimensional feature space to prepare it for the next transformer layers.
- We wanted to try as we didnt work with transformers in other submissions even if the results were not as good as expected

MODEL COMPARISONS & CONCLUSION

As a final conclusion and comparison comparing the results of 3 of them both GRU and LSTM models show a high-quality reconstruction of the real data, with the predicted (yellow) line closely following the actual prices. This results in a sparse and accurate identification of anomalies, typically occurring around sharp changes or irregular movements. In contrast, the Transformer model displays a less accurate reconstruction, especially in earlier segments of the series, leading to an excessive number of possible anomalies generally ignored. With this we concluded that maybe Transformer architecture struggles to capture short-term, local patterns that are critical in financial time series.

LTSM

TRANS

GRU

After research we find this can be due to Transformers, while excellent at capturing long-range dependencies in fields like natural language processing, lack the inherent temporal structure of GRU and LSTM networks. Transformers process all time steps simultaneously using attention mechanisms, which can overlook the subtle sequential patterns essential for precise anomaly detection. Furthermore, they require more data to train effectively and are highly sensitive to the design of positional encodings.

In summary, even if trying to explore different architectures one ended up being near useless for task. But we are satisfied with the accurate and interpretable results GRU and LSTM architecture performed.

ANOMALY ANALYSIS

We wanted to do a final feature in order to analyse and contrast that the results have coherence. For this we have selected the LTSM results for tesla as we think are the best ones and the one whose prices vary more.

- 2020-21: Global supply chains were disrupted following the COVID-19 pandemic, leading to shortages in semiconductors and raw materials. Tesla production delays due to chip shortages, affecting vehicle deliveries and causing volatility in its stock price.
- 2022: Russia-Ukraine War & Global Inflation. Increased material costs and geopolitical instability pressured Tesla's margins and investor sentiment.
- 2023: Financial Sector Turmoil & Climate Effects(el niño storm,...). This affected tesla as investors feared over economic slowdown and inflation volatility influenced Tesla's valuation and market confidence.
- 2024end–2025: U.S. Presidential Election and Elon Musk as Trumps supporter. The impact on Tesla was that investors speculated on how the outcome might affect Tesla’s regulatory environment, leading to heightened stock volatility and potential anomalies in price."

REFERENCES

1. Enable transformers for anomaly detection in Multivariate Time Series Data | by Dr. -Ing. Moussab Orabi | Medium
2. [2201.07284] TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data
3. arunbaruah/Anomaly_Detection_Transformer: Anomaly detection from OS logs using Transformers implemented with Pytorch.
4. Demystifying Neural Networks: Anomaly Detection with AutoEncoder | by Dagang Wei | Medium
5. Anomaly Detection with Autoencoder .ipynb - Colab
6. Using Autoencoders for Anomaly Detection: A Practical Guide - Toxigon
7. Link to repository: https://github.com/InigoPena/Anomaly_Detection_in_Stock_Prices