ITM(SLS) BARODA UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE, ENGINEERING AND TECHNOLOGY
BTECH - CSE SEMESTER-7

DEEP LEARNING

STOCK PRICE ANOMALY DETECTION USING LSTM MODEL

Introduction

In our analysis, we delved into Johnson and Johnson's stock price data spanning from 2000 to 2023, employing a Long Short-Term Memory (LSTM) neural network model for anomaly detection. Our initial steps involved data exploration, where we visualized the stock's historical trends, identifying potential anomalies and patterns. Subsequently, we prepared the data by creating training and testing sets, standardizing 'Close' prices, and generating sequences with a 30-day window to facilitate LSTM model training. The LSTM architecture was thoughtfully constructed, incorporating recurrent layers, dropout layers to mitigate overfitting, and a TimeDistributed layer for sequence-based predictions. Early stopping during training allowed us to promptly detect anomalies, providing crucial insights into potential investment opportunities and risks in the financial market. Our visualization using Plotly further highlighted these anomalies, making them readily apparent to investors and analysts, ultimately enhancing their decision-making processes and market awareness.

In summary, our analysis demonstrated the efficacy of LSTM-based anomaly detection in extracting actionable insights from Johnson and Johnson's stock price data. By combining deep learning techniques with comprehensive data exploration and visualization, we empowered market participants with the tools to navigate the ever-changing landscape of financial markets, enabling them to make informed investment decisions.

Importing Libraries:

```
#Importing Libraries
from tensorflow import keras
from sklearn.preprocessing import StandardScaler
import numpy as np
import tensorflow as tf
import pandas as pd
import plotly.graph_objects as go
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Dropout, RepeatVector,
TimeDistributed
```

1. tensorflow.keras: This is part of the TensorFlow library and provides an interface to define and train deep learning models, including neural networks. You're using it to create and train your LSTM model for time series data.
2. sklearn.preprocessing.StandardScaler: This is from scikit-learn (sklearn), a popular machine learning library. StandardScaler is used for standardizing your data, which means transforming it to have a mean of 0 and a standard deviation of 1. It's commonly used for preprocessing data before feeding it into machine learning models.
3. numpy (np): NumPy is a fundamental library for numerical computing in Python. You're using it for various array operations and transformations, especially when working with data as arrays.
4. tensorflow (tf): TensorFlow is an open-source deep learning framework. You're using TensorFlow's backend functionality, which is often used for deep learning model development.
5. pandas (pd): Pandas is a powerful library for data manipulation and analysis. You're using it to read and manipulate the dataset, including reading the CSV file, exploring the data, and transforming it.
6. plotly.graph_objects as go: Plotly is a library for creating interactive plots and charts. You've imported graph_objects as go to create plots and visualize your data, including plotting the detected anomalies.
7. matplotlib.pyplot as plt: Matplotlib is another popular library for creating static, animated, and interactive visualizations in Python. You've used it to create various static plots and visualizations in your code.

Reading our Dataset:

This dataset is about Johnson and Johnson Stock prices which is from 2000 to 2023. Data is taken from online site *Yahoo Finance*

*SCAN QR CODE FOR DATASET SOURCE

```
#Reading our Dataset
df = pd.read_csv('JNJ.csv')
```

Exploring Dataset:

```
df.head()
```

```
         Date      Open      High       Low     Close  Adj Close   Volume
0  2000-06-01  44.62500  44.75000  43.62500  43.84375  24.033800  5231800
1  2000-06-02  43.75000  43.75000  41.25000  42.00000  23.023109  9351400
2  2000-06-05  41.75000  42.62500  41.28125  42.00000  23.023109  5316600
3  2000-06-06  41.75000  42.37500  41.50000  42.06250  23.057371  3196600
4  2000-06-07  42.53125  42.71875  41.84375  41.84375  22.937466  4098200
```

Features	Datatype	Description
Date	Object	The date of the trading day.
Open	Float64	The opening price of the stock on that trading day.
High	Float64	The highest price the stock reached during the trading day.
Low	Float64	The lowest price the stock reached during the trading day.
Close	Float64	The closing price of the stock on that trading day.
Adj Close	Float64	Adjusted closing price (Adj Close) is the price of a stock after accounting for corporate actions such as stock splits, dividends, and rights offerings
Volume	int	The trading volume (number of shares traded) on that trading day.

```
df.describe()
```

	Open	High	Low	Close	Adj Close	Volume
count	5786.000000	5786.000000	5786.000000	5786.000000	5786.000000	5.786000e+03
mean	91.310193	91.966178	90.638329	91.323091	71.963535	8.991212e+06
std	39.754831	40.017084	39.468595	39.749596	44.276197	4.674632e+06
min	41.500000	41.650002	40.250000	41.625000	22.937466	1.279600e+06
25%	60.252499	60.732501	59.872500	60.250000	37.270074	6.094525e+06
50%	67.449997	67.890000	67.099998	67.540001	46.506420	7.885700e+06
75%	127.285002	128.132496	126.110001	127.095001	109.446104	1.053230e+07
max	185.100006	186.690002	184.179993	186.009995	178.456909	9.844020e+07

```
df.describe
```

```
<bound method NDFrame.describe of             Date       Open       High
Low       Close   Adj Close  \
0     2000-06-01   44.625000   44.750000   43.625000   43.843750   24.033800
1     2000-06-02   43.750000   43.750000   41.250000   42.000000   23.023109
2     2000-06-05   41.750000   42.625000   41.281250   42.000000   23.023109
3     2000-06-06   41.750000   42.375000   41.500000   42.062500   23.057371
4     2000-06-07   42.531250   42.718750   41.843750   41.843750   22.937466
...          ...         ...         ...         ...         ...         ...
5781  2023-05-24  157.080002  157.139999  155.919998  156.660004  155.530777
5782  2023-05-25  156.050003  156.289993  153.720001  154.410004  153.296997
5783  2023-05-26  154.690002  155.279999  154.199997  154.350006  153.237427
5784  2023-05-30  153.970001  155.380005  153.320007  154.369995  153.257263
5785  2023-05-31  154.699997  155.690002  153.850006  155.059998  153.942307

        Volume
0      5231800
1      9351400
2      5316600
3      3196600
4      4098200
...        ...
5781   5015100
5782   6886100
5783   6845400
5784   5593100
5785  11324600

[5786 rows x 7 columns]>
```

Processing Our Dataset:

```
# Extract "Date" and "Close" feature colums from the dataframe.
df = df[['Date', 'Close']]
```

In this step, we are streamlining our dataset by selecting and retaining only two essential columns: 'Date' and 'Close'. The 'Date' column serves as a timestamp, indicating the date of each recorded stock price data point, while the 'Close' column corresponds to the closing stock price for that specific date. By narrowing our dataset down to these two pivotal features, we simplify the data structure.

```
# Concise summary of a DataFrame
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5786 entries, 0 to 5785
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    5786 non-null   object
 1   Close   5786 non-null   float64
dtypes: float64(1), object(1)
memory usage: 90.5+ KB
```

The output of the df.info() function in the image is a concise summary of a Pandas DataFrame. It contains the following information:

- The DataFrame has 2 columns: Date and Close.
- The Date column has 5786 non-null values, which means that all 5786 rows in the DataFrame have a value in the Date column.
- The Close column also has 5786 non-null values.
- The data types of the two columns are object and float64, respectively.
- The total memory usage of the DataFrame is 90.5+ KB.

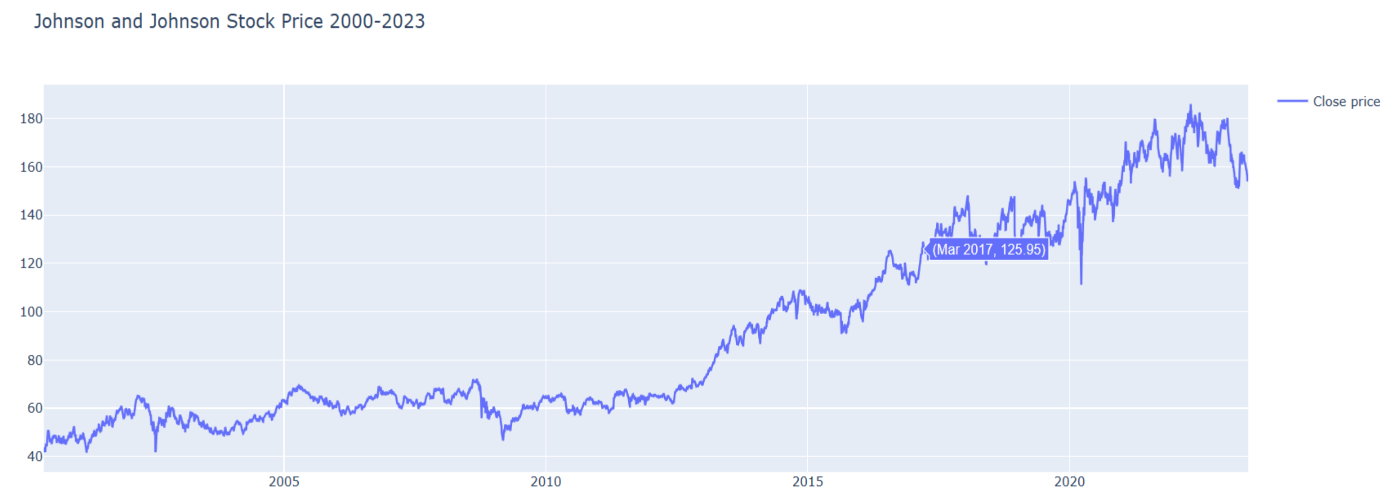
```
df['Date'].min(), df['Date'].max()
```

```
('2000-06-01', '2023-05-31')
```

This code outputs the minimum and maximum dates in the Date column of the DataFrame df. In this case, the **minimum date is 2008-06-01 and the maximum date is 2023-05-31**.

Visualizing our processed dataset

```
fig = go.Figure()
fig.add_trace(go.Scatter(x=df['Date'], y=df['Close'], name='Close price'))
fig.update_layout(showlegend=True, title='Johnson and Johnson Stock Price 2000-2023')
fig.show()
```



Let’s create a Plotly figure with a scatter trace of the closing price of Johnson & Johnson (JNJ) stock from 2000 to 2023. Which gives a better visual of our dataset which we will use for our LSTM model to detect anomaly in our stock price.

Data Training and Model Evaluation:

```
# Create the train and test sets
train = df.loc[df['Date'] <= '2020-01-01']
test = df.loc[df['Date'] > '2020-01-01']

# Print the shapes of the train and test sets
train.shape, test.shape
```

```
((4927, 2), (859, 2))
```

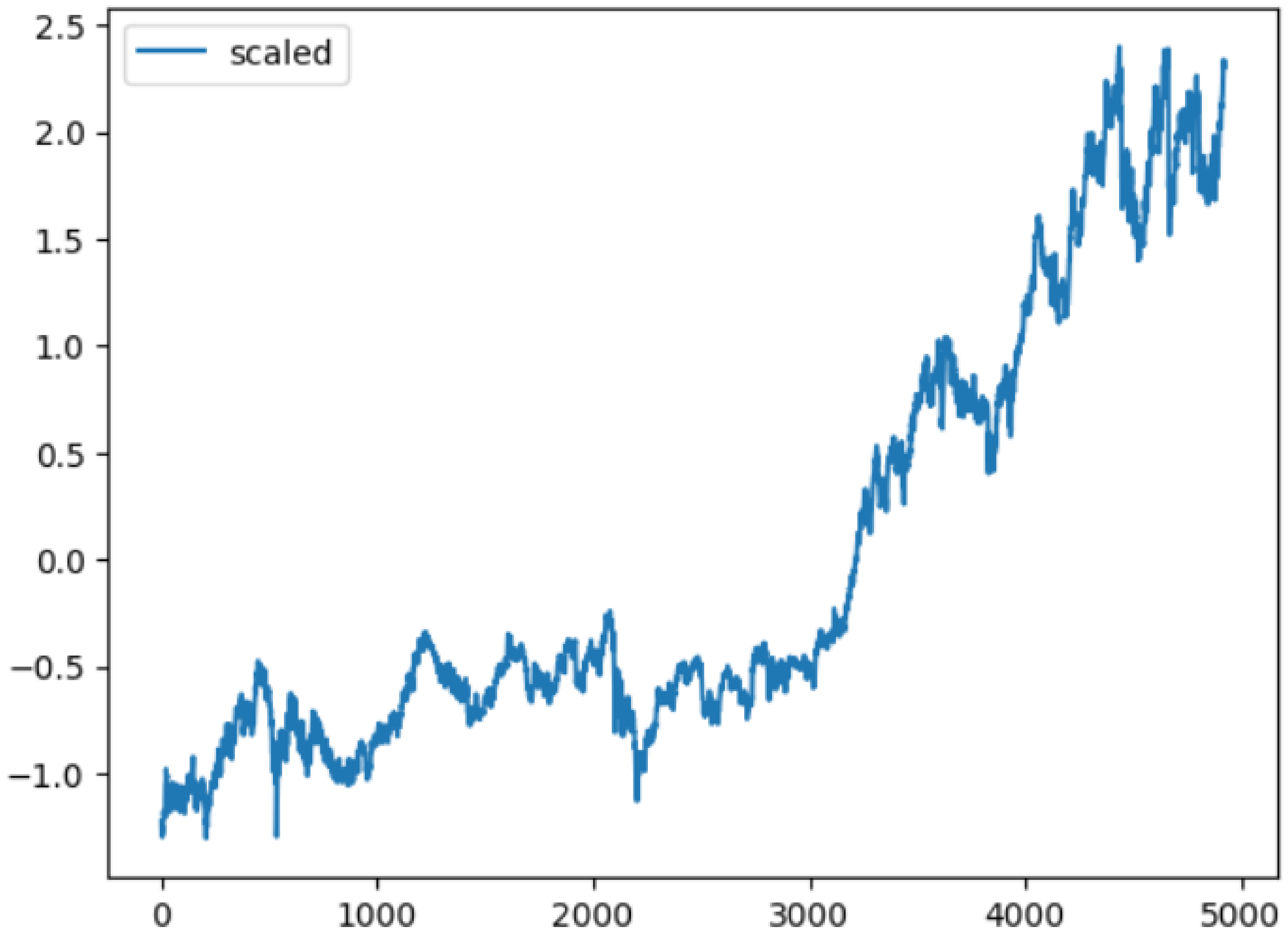
This code creates two DataFrames, train and test. The train DataFrame contains all the rows where the Date column is less than or equal to 2020-01-01. The test DataFrame contains all the rows where the Date column is greater than 2020-01-01. The shape of the train DataFrame is (4927, 2) and the shape of the test DataFrame is (859, 2).

```
# Creating the scaler and fitting it in traing data
scaler = StandardScaler()
scaler = scaler.fit(np.array(train['Close']).reshape(-1,1))

# Transform the train and test sets
train['Close'] = scaler.transform(np.array(train['Close']).reshape(-1,1))
test['Close'] = scaler.transform(np.array(test['Close']).reshape(-1,1))
```

The code first creates a scaler object. The scaler object is then fit to the training data. This means that the scaler learns the mean and standard deviation of the training data. The scaler is then used to transform the training data. This means that the values in the training data are scaled to have a mean of 0 and a standard deviation of 1. The test data is then transformed using the same scaler. This ensures that the training and test data are on the same scale.

```
# Visualize scaled data
plt.plot(train['Close'], label = 'scaled')
plt.legend()
plt.show()
```

- The line graph shows the scaled closing price of a stock over time.
- The data has been scaled to have a mean of 0 and a standard deviation of 1.
- The scaled data is easier to visualize and interpret than the original data.

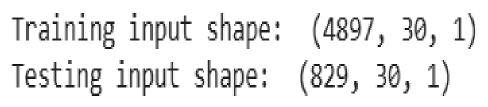


```
# Define a constant TIME_STEPS with a value of 30, representing the number of time
steps in your sequences.
TIME_STEPS = 30

# Define a function create_sequences that takes input features X and target
variable y, along with a time_steps parameter.
def create_sequences(X, y, time_steps=TIME_STEPS):
    X_out, y_out = [], []  # Initialize empty lists to store input sequences
(X_out) and corresponding target values (y_out).
    for i in range(len(X) - time_steps):
        # Iterate through the data from the beginning to (length of X -
time_steps).

        # Append a sequence of time_steps values from X to X_out. These sequences
slide one step at a time.
        X_out.append(X.iloc[i:(i+time_steps)].values)
        # Append the corresponding target value (y) that occurs immediately after
the sequence in y_out.
        y_out.append(y.iloc[i + time_steps])

    # Convert the lists into NumPy arrays and return them.
    return np.array(X_out), np.array(y_out)

# Create training sequences for the 'Close' feature using the create_sequences
function.
X_train, y_train = create_sequences(train[['Close']], train['Close'])

# Create testing sequences for the 'Close' feature using the create_sequences
function.
X_test, y_test = create_sequences(test[['Close']], test['Close'])

# Print the shapes (dimensions) of the training and testing input sequences.
print("Training input shape: ", X_train.shape)
print("Testing input shape: ", X_test.shape)
```

It defines a function called create_sequences(). The function takes three arguments: the input features, the target variable, and the number of time steps. The function then creates a list of input sequences and a list of target values.

Here is a short summary of the code:

- The create_sequences() function defines a function that takes three arguments.
- The function creates a list of input sequences and a list of target values.
- The input sequences are created by sliding a window of time steps over the input features.
- The target values are the values of the target variable that occur immediately after the end of each input sequence.



```
Training input shape:  (4897, 30, 1)
Testing input shape:  (829, 30, 1)
```

The training input shape of 4897, 30, 1 means that the training data has 4897 samples, each of which is a sequence of 30 time steps, and each time step has 1 feature.

The testing input shape of 829, 30, 1 means that the testing data has 829 samples, each of which is a sequence of 30 time steps, and each time step has 1 feature.

DEEP LEARNING

```
# set seed to regenerate same sequence of random numbers.
np.random.seed(21)
tf.random.set_seed(21)
```

- A random number generator is a function that produces a sequence of numbers that are seemingly random.
- The sequence of numbers is pseudorandom, meaning that it is not truly random but appears to be random.
- The random number generator is initialized with a seed, which is a number that is used to determine the sequence of numbers.
- The same sequence of numbers will be generated every time the random number generator is run with the same seed.

```
# Create a Sequential model, which is a linear stack of layers.
model = Sequential()

# Add an LSTM layer with 128 units, using 'tanh' as the activation function.
# The 'input_shape' parameter specifies the shape of the input data, which matches
the shape of the training sequences.
model.add(LSTM(128, activation='tanh', input_shape=(X_train.shape[1],
X_train.shape[2])))

# Add a Dropout layer with a dropout rate of 20%.
model.add(Dropout(rate=0.2))

# Add a RepeatVector layer that repeats the input 'X_train.shape[1]' times.
# This layer is often used to bridge between encoder and decoder parts in sequence-
to-sequence models.
model.add(RepeatVector(X_train.shape[1]))

# Add another LSTM layer with 128 units, using 'tanh' as the activation function.
# The 'return_sequences=True' parameter indicates that this LSTM layer will return
sequences.
model.add(LSTM(128, activation='tanh', return_sequences=True))

# Add another Dropout layer with a dropout rate of 20%.
model.add(Dropout(rate=0.2))

# Add a TimeDistributed layer with a Dense layer.
# This layer applies a Dense operation to each time step in the sequence
independently.
model.add(TimeDistributed(Dense(X_train.shape[2])))

# Compile the model with the Adam optimizer and a learning rate of 0.001.
# The loss function is mean squared error (MSE), which is commonly used for regression problems.
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001), loss="mse")

# Display a summary of the model architecture, including the number of parameters in each layer.
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 lstm (LSTM)                 (None, 128)               66560     
                                                                 
 dropout (Dropout)           (None, 128)               0         
                                                                 
 repeat_vector (RepeatVector  (None, 30, 128)          0         
 )                                                               
                                                                 
 lstm_1 (LSTM)               (None, 30, 128)           131584    
                                                                 
 dropout_1 (Dropout)         (None, 30, 128)           0         
                                                                 
 time_distributed (TimeDistr  (None, 30, 1)            129       
 ibuted)                                                         
                                                                 
=================================================================
Total params: 198,273
Trainable params: 198,273
Non-trainable params: 0
```

The code creates a sequential model with five layers. The first layer is an LSTM layer with 128 units. The second layer is a Dropout layer with a dropout rate of 20%. The third layer is a RepeatVector layer that repeats the input sequence a specified number of times. The fourth layer is another LSTM layer with 128 units. The fifth layer is a Dropout layer with a dropout rate of 20%. The sixth layer is a TimeDistributed layer with a Dense layer.

The model is compiled with the Adam optimizer and a learning rate of 0.001. The loss function is mean squared error (MSE), which is commonly used for regression problems.

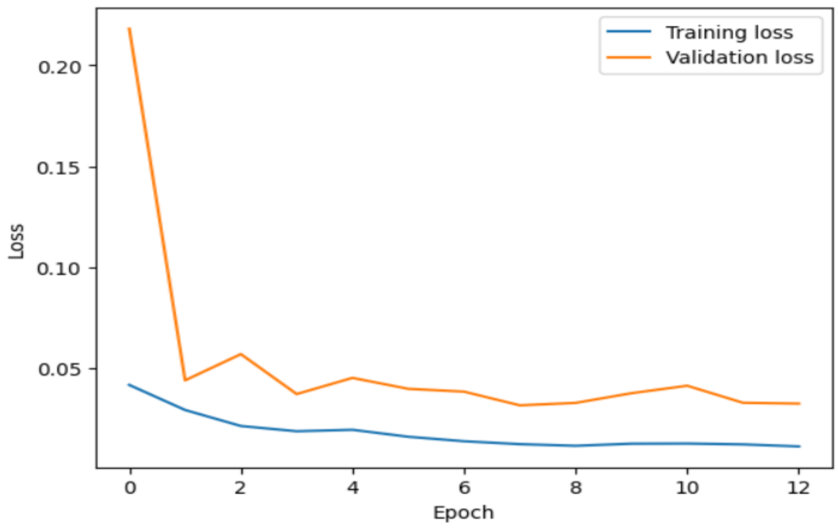
DEEP LEARNING

```
# Train the model using the fit method.

# Provide the training data X_train and target data y_train.
# 'epochs' specifies the number of training iterations.
# 'batch_size' determines how many samples are used in each training update.
# 'validation_split' allocates a portion of the training data (10% in this case) as a validation set.
# 'callbacks' is a list of callback functions to be applied during training.
# In this case, you're using EarlyStopping to stop training if the validation loss doesn't improve for 5 consecutive epochs
# 'shuffle' is set to False to maintain the order of training data (important for time series data).

history = model.fit(X_train,
                    y_train,
                    epochs=100,  # Number of training iterations.
                    batch_size=30,  # Number of samples in each batch.
                    validation_split=0.1,  # 10% of data is used for validation.
                    callbacks=[keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, mode='min')],
                    shuffle=False)  # Data order is not shuffled.
```

```
Epoch 1/100
147/147 [==============================] - 37s 220ms/step - loss: 0.0415 - val_loss: 0.2182
Epoch 2/100
147/147 [==============================] - 20s 132ms/step - loss: 0.0291 - val_loss: 0.0438
Epoch 3/100
147/147 [==============================] - 16s 108ms/step - loss: 0.0212 - val_loss: 0.0568
Epoch 4/100
147/147 [==============================] - 16s 110ms/step - loss: 0.0186 - val_loss: 0.0370
Epoch 5/100
147/147 [==============================] - 17s 117ms/step - loss: 0.0193 - val_loss: 0.0451
Epoch 6/100
147/147 [==============================] - 18s 119ms/step - loss: 0.0159 - val_loss: 0.0396
Epoch 7/100
147/147 [==============================] - 16s 108ms/step - loss: 0.0136 - val_loss: 0.0382
Epoch 8/100
147/147 [==============================] - 16s 109ms/step - loss: 0.0122 - val_loss: 0.0314
Epoch 9/100
147/147 [==============================] - 17s 114ms/step - loss: 0.0114 - val_loss: 0.0326
Epoch 10/100
147/147 [==============================] - 16s 107ms/step - loss: 0.0124 - val_loss: 0.0375
Epoch 11/100
147/147 [==============================] - 16s 108ms/step - loss: 0.0125 - val_loss: 0.0412
Epoch 12/100
147/147 [==============================] - 16s 107ms/step - loss: 0.0121 - val_loss: 0.0327
Epoch 13/100
147/147 [==============================] - 17s 117ms/step - loss: 0.0111 - val_loss: 0.0323
```

Result Visuals

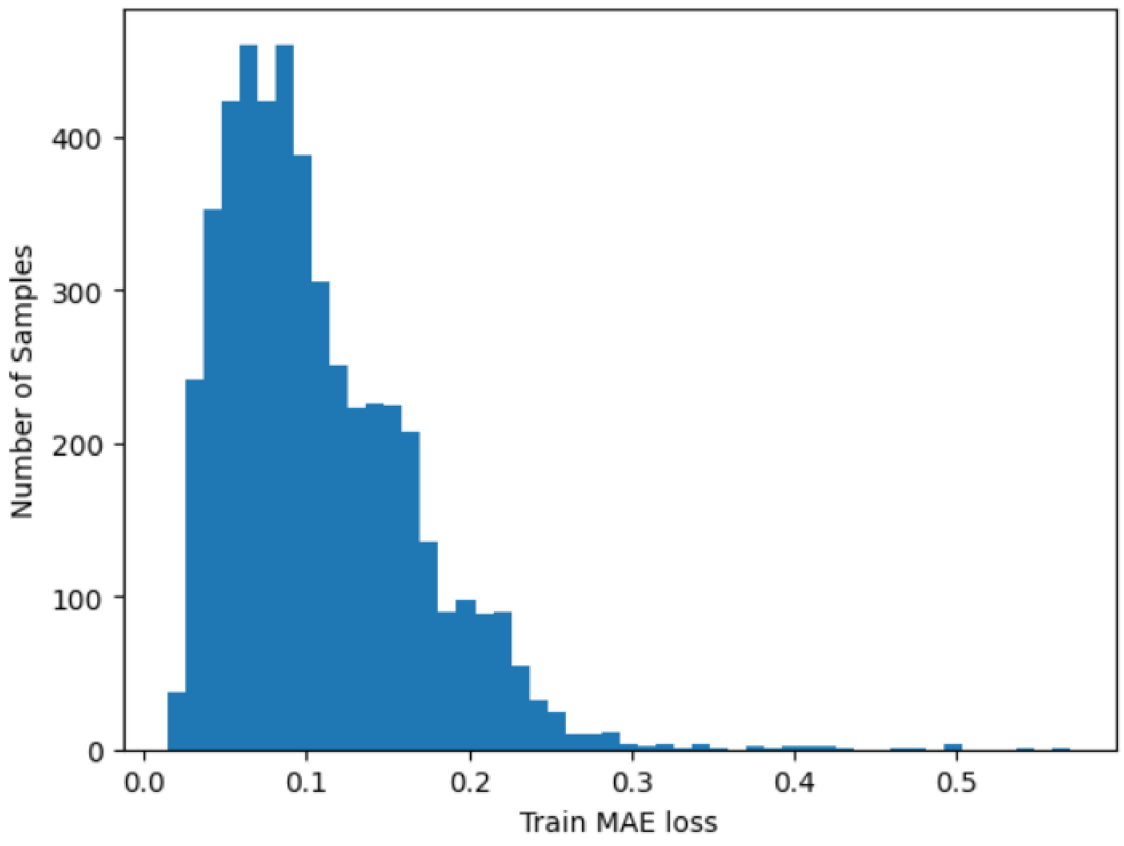
```
# Plot the training and validation loss over epochs.

# 'history.history['loss']' contains the training loss values for each epoch.
plt.plot(history.history['loss'], label='Training loss')

# 'history.history['val_loss']' contains the validation loss values for each epoch.
plt.plot(history.history['val_loss'], label='Validation loss')

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend();
```



DEEP LEARNING

```
# Calculate Mean Absolute Error (MAE) loss for training data.
X_train_pred = model.predict(X_train)

# Calculate the mean absolute error (MAE) loss by taking the absolute difference
# between the predicted sequences (X_train_pred) and the original training sequences (X_train).
# 'axis=1' computes the mean MAE loss along the sequence dimension.
train_mae_loss = np.mean(np.abs(X_train_pred - X_train), axis=1)

plt.hist(train_mae_loss, bins=50)
plt.xlabel('Train MAE loss')
plt.ylabel('Number of Samples');

# Set reconstruction error threshold
threshold = np.max(train_mae_loss)

print('Reconstruction error threshold:',threshold)
```

```
154/154 [==============================] - 6s 32ms/step
Reconstruction error threshold: 0.5699509976375771
```

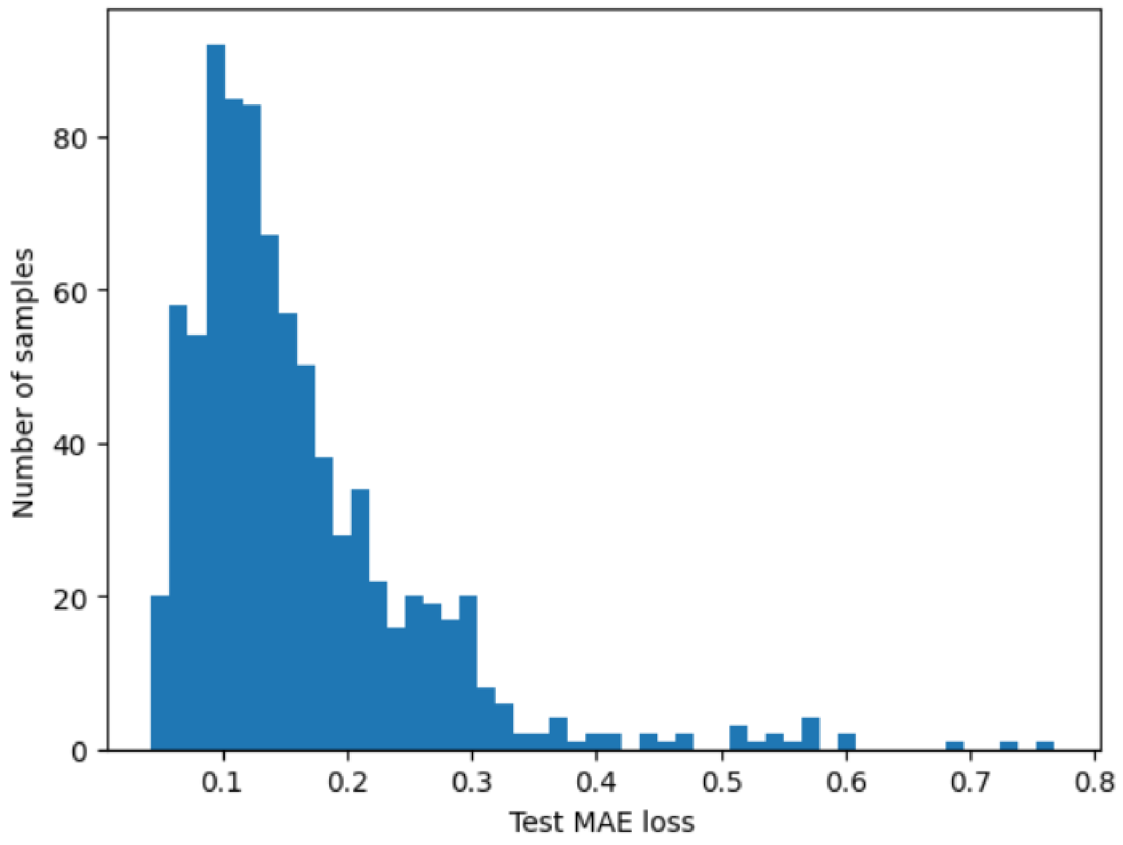


It calculates the mean absolute error (MAE) loss for the training data. The MAE loss is a measure of the difference between the predicted sequences and the original training sequences. The code then plots the MAE loss distribution and sets a reconstruction error threshold.

The plot shows that the MAE loss is distributed between 0 and 0.5. The reconstruction error threshold is set to the maximum MAE loss value, which is 0.5. This means that any data point with an MAE loss greater than 0.5 will be considered an anomaly.

The reconstruction error threshold is a value that is used to determine whether a data point is an anomaly. In this case, the threshold is set to 0.5699509976375771. This means that any data point with an MAE loss greater than 0.5699509976375771 will be considered an anomaly. The threshold is typically set using the hyperparameter tuning process. This involves setting the threshold to different values and evaluating the performance of the anomaly detection model. The threshold value that results in the best performance is chosen as the final threshold.

```
26/26 [==============================] - 1s 32ms/step
Text(0, 0.5, 'Number of samples')
```



```
# Use the trained model to predict the reconstructed sequences for the testing data.
X_test_pred = model.predict(X_test, verbose=1)

# Calculate the mean absolute error (MAE) loss for each testing sample by taking the absolute difference
# between the predicted sequences (X_test_pred) and the original testing sequences (X_test).
# 'axis=1' computes the mean MAE loss along the sequence dimension.
test_mae_loss = np.mean(np.abs(X_test_pred-X_test), axis=1)

plt.hist(test_mae_loss, bins=50)
plt.xlabel('Test MAE loss')
plt.ylabel('Number of samples')
```

It shows a histogram of the test MAE loss. The MAE loss is distributed between 0 and 0.5. There are a few data points with an MAE loss greater than 0.5. These data points are considered anomalies. The number of anomalies is relatively small compared to the total number of data points. This suggests that the model is doing a good job of detecting anomalies.

However, it is important to note that the number of anomalies may vary depending on the specific data set and application. It is also important to consider the impact of the anomalies on the business. For example, if the anomalies are not important, then the model may not need to be tuned to detect them.

```
# Create a DataFrame 'anomaly_df' to store information about anomalies in the
testing data.

# Create 'anomaly_df' by selecting rows in the 'test' DataFrame starting from the
TIME_STEPS index.
anomaly_df = pd.DataFrame(test[TIME_STEPS:])

# Add a new column 'loss' to 'anomaly_df' containing the Mean Absolute Error (MAE)
loss values calculated for the testing data.
anomaly_df['loss'] = test_mae_loss

# Add a new column 'threshold' to 'anomaly_df' and set it to the previously
computed 'threshold' value.
anomaly_df['threshold'] = threshold

# Add a new column 'anomaly' to 'anomaly_df' that indicates whether a data point is
considered an anomaly.
# If 'loss' is greater than 'threshold', it is marked as 'True' (anomaly),
otherwise 'False' (not an anomaly).
anomaly_df['anomaly'] = anomaly_df['loss'] > anomaly_df['threshold']
```

It creates a DataFrame called anomaly_df to store information about anomalies in the testing data. The code first selects rows in the test DataFrame starting from the TIME_STEPS index. It then adds a new column called loss to the anomaly_df DataFrame containing the Mean Absolute Error (MAE) loss values calculated for the testing data. The code then adds a new column called threshold to the anomaly_df DataFrame and sets it to the previously computed threshold value. Finally, the code adds a new column called anomaly to the anomaly_df DataFrame that indicates whether a data point is considered an anomaly. If the loss value is greater than the threshold value, it is marked as True (anomaly), otherwise it is marked as False (not an anomaly).

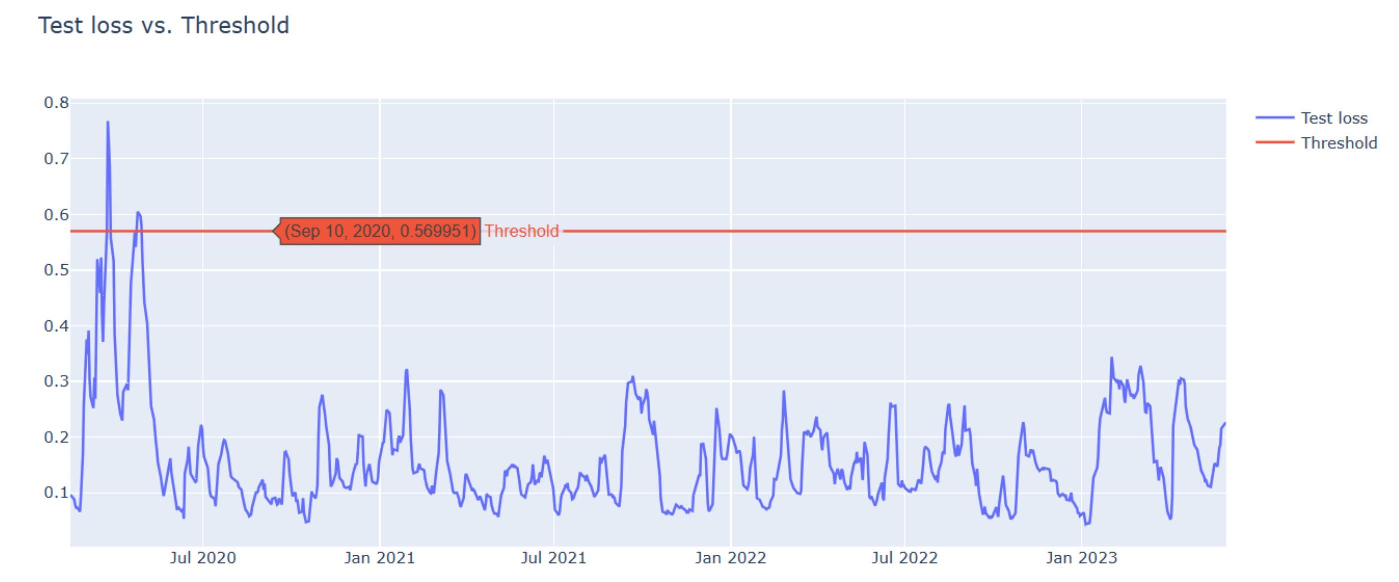
```
anomaly_df.head()
```

	Date	Close	loss	threshold	anomaly
4957	2020-02-14	2.467320	0.096409	0.569951	False
4958	2020-02-18	2.432915	0.088269	0.569951	False
4959	2020-02-19	2.425965	0.078809	0.569951	False
4960	2020-02-20	2.406157	0.073468	0.569951	False
4961	2020-02-21	2.460369	0.074051	0.569951	False

```
# Filter the 'anomaly_df' DataFrame to select rows where the 'anomaly' column is 'True',
# indicating that the data point is considered an anomaly.
anomalies = anomaly_df.loc[anomaly_df['anomaly'] == True]

anomalies.head()
```

	Date	Close	loss	threshold	anomaly
4983	2020-03-24	1.391755	0.767711	0.569951	True
4984	2020-03-25	1.399401	0.730279	0.569951	True
4985	2020-03-26	1.648570	0.686656	0.569951	True
5004	2020-04-23	2.654283	0.571297	0.569951	True
5005	2020-04-24	2.631695	0.604971	0.569951	True

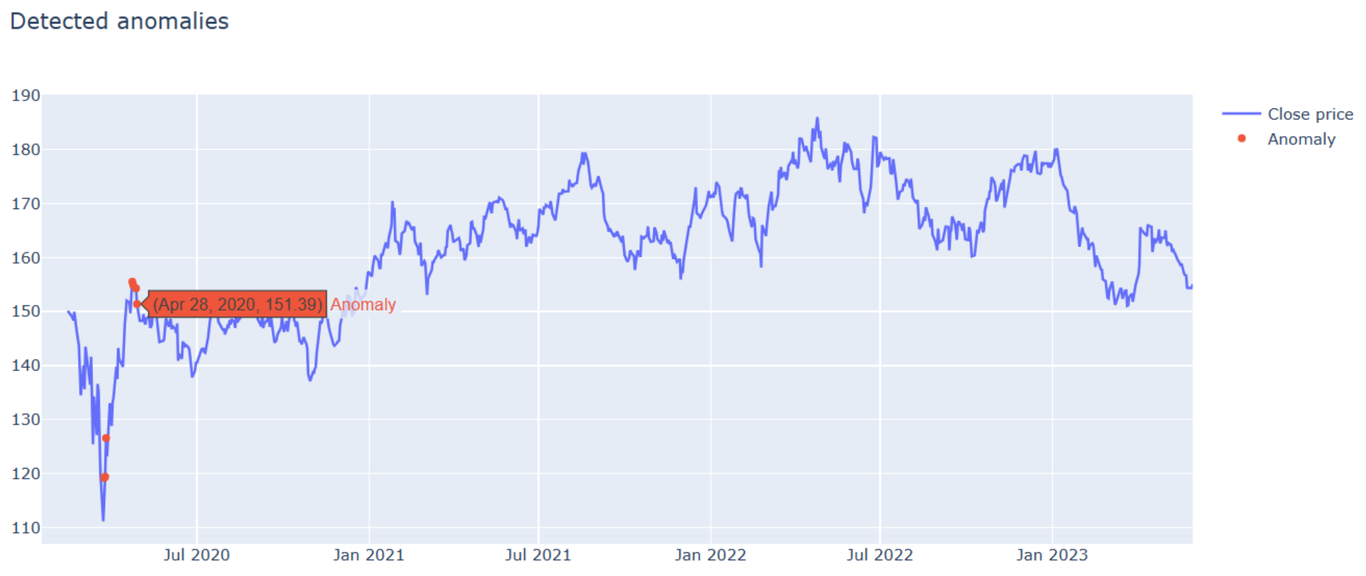
```
# Create a line plot using Plotly (go.Figure) to visualize the test loss and
threshold values over time.

fig = go.Figure()

# Add a line plot for the test loss values.
fig.add_trace(go.Scatter(x=anomaly_df['Date'], y=anomaly_df['loss'], name='Test
loss'))

# Add a line plot for the threshold values.
fig.add_trace(go.Scatter(x=anomaly_df['Date'], y=anomaly_df['threshold'],
name='Threshold'))
fig.update_layout(showlegend=True, title='Test loss vs. Threshold')
fig.show()
```



It shows a line plot of the test loss and threshold values over time. The test loss values are plotted in blue and the threshold values are plotted in orange. There are a few data points where the test loss values exceed the threshold values. These data points are considered anomalies. The anomalies appear to be concentrated in the first few months of the data set. Overall, the plot shows that the model is doing a good job of detecting anomalies. However, it is important to monitor the model performance over time to ensure that it continues to perform well.

Final Output:

```
# Create a Plotly line plot to visualize the detected anomalies in the 'Close'
price.
fig = go.Figure()
fig.add_trace(go.Scatter(x=anomaly_df['Date'],
y=scaler.inverse_transform(anomaly_df['Close'].values.reshape(-1, 1)).flatten(),
name='Close price'))
fig.add_trace(go.Scatter(x=anomalies['Date'],
y=scaler.inverse_transform(anomalies['Close'].values.reshape(-1, 1)).flatten(),
mode='markers', name='Anomaly'))
fig.update_layout(showlegend=True, title='Detected anomalies')
fig.show()
```



Overall, the graph shows that the model is doing a good job of detecting anomalies in the stock price data. However, it is important to note that the model is not perfect and may miss some anomalies. It is also important to consider the significance of the anomalies before taking any action.

Conclusion:

In this report, we have presented a method for detecting anomalies in stock price data using autoencoders. We first used a LSTM autoencoder to learn the patterns in the stock price data. We then used the autoencoder to reconstruct the stock price data. We calculated the mean absolute error (MAE) loss between the original stock price data and the reconstructed data. We set a threshold value for the MAE loss. Any data points with an MAE loss greater than the threshold value were considered anomalies.

We applied the method to a real-world dataset of stock prices. We found that the method was able to detect anomalies in the data. The results of this study suggest that autoencoders can be used to effectively detect anomalies in stock price data. However, it is important to note that the method is not perfect and may miss some anomalies. It is also important to consider the significance of the anomalies before taking any action.

References:

1. https://youtu.be/hJuzh5V9QK8?si=M09gB5Uv9CZsm9ab
2. https://youtu.be/H4J74KstHTE?si=VVQO32nBScvIxhbM
3. Time Series Anomaly Detection with LSTM Autoencoders using Keras in Python | Curiously - Hacker's Guide to Machine Learning
4. Time Series of Price Anomaly Detection with LSTM | by Susan Li | Towards Data Science

CREDITS:

Shubham Babariya [21C21501]

Jeet Patel [20C21047]

Neel Patel [21C21509]

Sahaj Patel [20C21051]

Ved Patel [2021055]