

## CSE 676-B: Deep Learning - Assignment 2 Report

Names:

Epuru Sai Muralidhar

Revanth Balineni

UBID

saimural

rbalinen

equal contribution

equal contribution

### Part -2

#### 1) Datasets chosen -> HardDisk

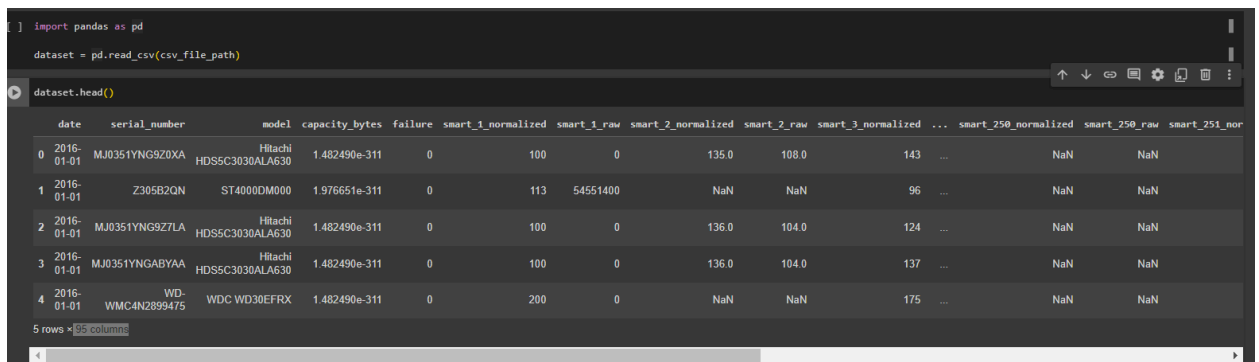
[Hard Drive Test Data \(kaggle.com\)](https://www.kaggle.com/datasets/ucsd/hard-drive-test-data)

This dataset is all about keeping an eye on hard drives to catch any signs they might fail using something called S.M.A.R.T data.

Each row is like a daily health check-up of a hard drive, listing out important stats that can give clues on how well its doing.

With 90 different stats spread across columns, plus info on the hard drives model, capacity, and whether it failed or not, theres a lot to dig into.

The goal here is to use this data to predict when a hard drive might be on the brink of failing, making it a crucial tool for preventing data loss and keeping systems running smoothly.



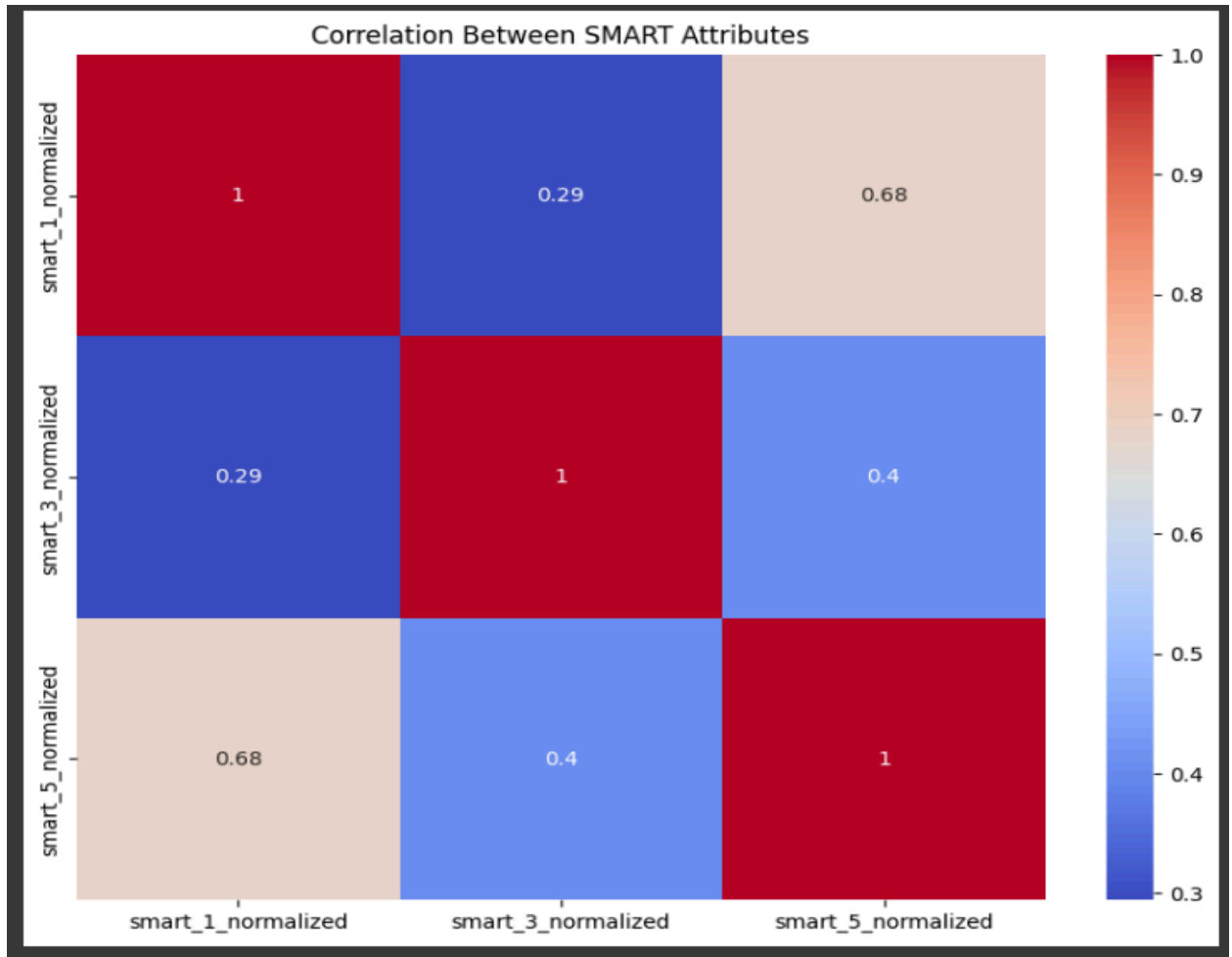
```
import pandas as pd
dataset = pd.read_csv(csv_file_path)
dataset.head()
```

	date	serial_number	model	capacity_bytes	failure	smart_1_normalized	smart_1_raw	smart_2_normalized	smart_2_raw	smart_3_normalized	...	smart_250_normalized	smart_250_raw	smart_251_normalized
0	2016-01-01	MJ0351YNG9Z0XA	Hitachi HDS5C3030ALA630	1.482490e-311	0	100	0	135.0	108.0	143	...	NaN	NaN	NaN
1	2016-01-01	Z305B2QN	ST4000DM000	1.976651e-311	0	113	54551400	NaN	NaN	96	...	NaN	NaN	NaN
2	2016-01-01	MJ0351YNG9Z7LA	Hitachi HDS5C3030ALA630	1.482490e-311	0	100	0	136.0	104.0	124	...	NaN	NaN	NaN
3	2016-01-01	MJ0351YNGABYAA	Hitachi HDS5C3030ALA630	1.482490e-311	0	100	0	136.0	104.0	137	...	NaN	NaN	NaN
4	2016-01-01	WDC WMC4N2899475	WDC WD30EFRX	1.482490e-311	0	200	0	NaN	NaN	175	...	NaN	NaN	NaN

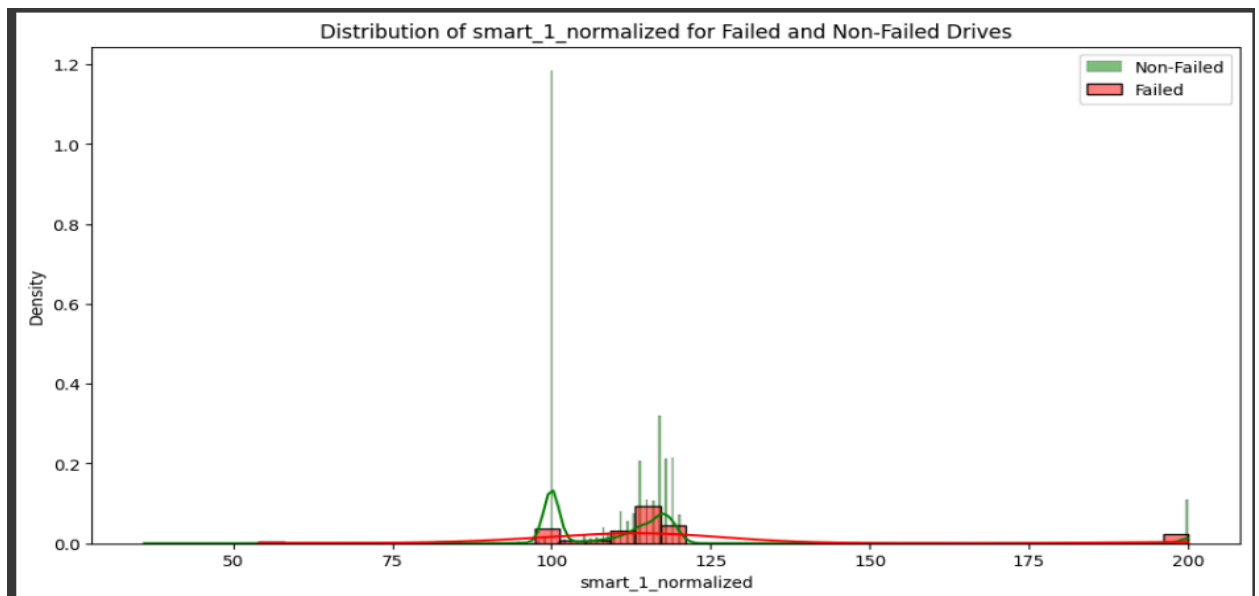
5 rows x 15 columns

#### Visualization graphs:

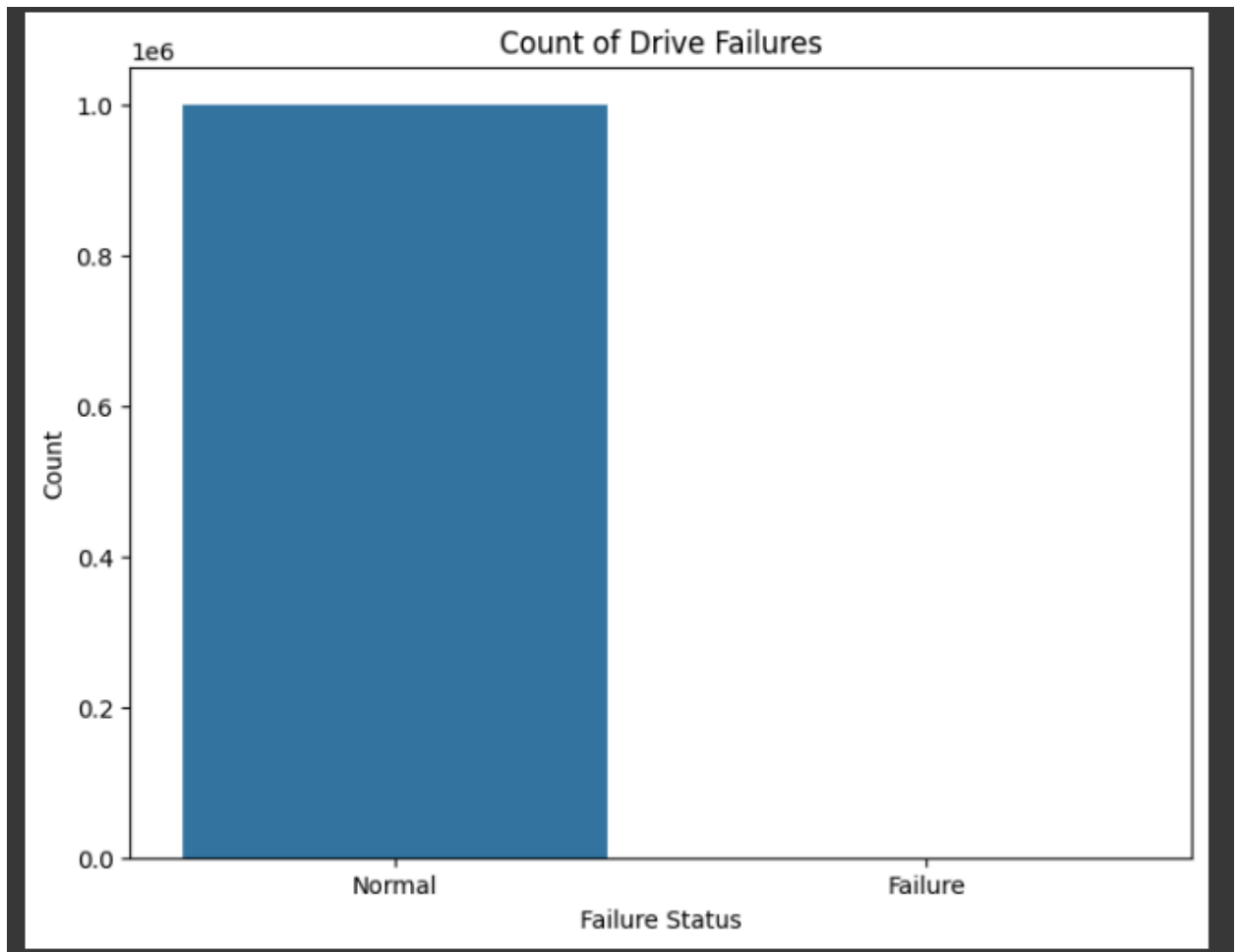
SMART attributes for simplicity



Distribution of smart\_attribute for Failed and Non-Failed Drives



Count of Drive Failures



## 2) Simple Autoencoder Model:

This model is designed as a straightforward autoencoder without convolutional /recurrent layers, making it suitable for dense input data.

```

import torch
import torch.nn as nn
import torch.optim
from sklearn.metrics import mean_squared_error, accuracy_score
import numpy as np

class SimpleAutoencoder(nn.Module):
    def __init__(self, input_size):
        super(SimpleAutoencoder, self).__init__()
        self.encoder = self.create_encoder(input_size)
        self.decoder = self.create_decoder(input_size)

    def create_encoder(self, input_size):
        layers = [
            nn.Linear(input_size, 128), nn.ReLU(True),
            nn.Linear(128, 64), nn.ReLU(True),
            nn.Linear(64, 12), nn.ReLU(True),
            nn.Linear(12, 3)
        ]
        return nn.Sequential(*layers)

    def create_decoder(self, input_size):
        layers = [
            nn.Linear(3, 12), nn.ReLU(True),
            nn.Linear(12, 64), nn.ReLU(True),
            nn.Linear(64, 128), nn.ReLU(True),
            nn.Linear(128, input_size), nn.Sigmoid()
        ]
        return nn.Sequential(*layers)

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

```

### Encoder:

The encoder compresses data via linear layers that decrease in size: from the input dimension to 128, then 64, 12, and finally 3 units. Each layer, except the last, uses a ReLU activation to add non-linearity, enhancing the model's ability to capture complex patterns in a compact representation.

### Decoder:

The decoder reconstructs data from the compressed form, inversely mirroring the encoder: from 3 units to the original dimension. Layers expand from 3 to 128, using

ReLU activations, except the final layer, which employs a sigmoid to normalize outputs to [0, 1], assuming pre-normalized input data.

### LSTM Autoencoder model:

```
class TemporalAutoencoder(nn.Module):
    def __init__(self, feature_dim, sequence_length, num_features):
        super(TemporalAutoencoder, self).__init__()
        self.sequence_length = sequence_length
        self.num_features = num_features
        self.lstm_hidden_dim = 128 # Hidden dimension for LSTM layer
        self.lstm_layers = 1 # Number of layers in LSTM

        self.lstm_layer = nn.LSTM(
            input_size=num_features,
            hidden_size=self.lstm_hidden_dim,
            num_layers=self.lstm_layers,
            batch_first=True
        )

        self.feature_encoder = self._build_encoder(sequence_length * self.lstm_hidden_dim)
        self.feature_decoder = self._build_decoder(sequence_length * num_features)

    def _build_encoder(self, input_dim):
        return nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(True),
            nn.Linear(128, 64),
            nn.ReLU(True),
            nn.Linear(64, 12),
            nn.ReLU(True),
            nn.Linear(12, 3)
        )

    def _build_decoder(self, output_dim):
        return nn.Sequential(
            nn.Linear(3, 12),
            nn.ReLU(True),
            nn.Linear(12, 64),
            nn.ReLU(True),
            nn.Linear(64, 128),
            nn.ReLU(True),
            nn.Linear(128, output_dim),
            nn.Sigmoid()
        )
```

### Configuration:

Input Size: Number of features in each sequence (num\_features).

Hidden Size: 128 units, defining the dimensionality of the LSTM output and hidden state.

Number of Layers: 1, indicating a single LSTM layer is used.

Batch First: True, which means the input tensor is expected to have a batch size as the first dimension.

#### **Encoder:**

The encoder compresses sequences into a compact form via linear layers, shrinking dimensions from the LSTM's output down to 128, 64, 12, and finally a 3-unit bottleneck. ReLU activations follow each layer, promoting non-linear learning, with the bottleneck capturing the essence of the data in a reduced representation.

#### **Decoder:**

The decoder reconstructs sequences from a 3-unit bottleneck, expanding through layers of 12, 64, and 128 units with ReLU activations, then uses a sigmoid in the final layer to normalize outputs to  $[0, 1]$ . This structure mirrors the encoder inversely, restoring the original sequence size from the compressed representation.

#### **Compact Deep AutoEncoder:**

##### **Encoder:**

The encoder compresses input data to a compact form through layers reducing dimensions from the original size to 256, then progressively to 128, 64, 32, 16, and finally a 3-unit bottleneck. ReLU activation functions are used at each step to ensure non-linear learning and prevent gradient issues.

##### **Decoder:**

The decoder reconstructs data from a 3-unit bottleneck, progressively expanding through layers to 16, 32, 64, 128, and 256 units, then back to the original dimension. It employs ReLU activations for intermediate layers and a Sigmoid in the final layer to normalize outputs between 0 and 1, matching pre-normalized input.

```
import torch
import torch.nn as nn
import numpy as np

class CompactDeepAutoencoder(nn.Module):
    def __init__(self, features_dim):
        super(CompactDeepAutoencoder, self).__init__()
        self.encode_layers = nn.Sequential(
            nn.Linear(features_dim, 256), nn.ReLU(True),
            nn.Linear(256, 128), nn.ReLU(True),
            nn.Linear(128, 64), nn.ReLU(True),
            nn.Linear(64, 32), nn.ReLU(True),
            nn.Linear(32, 16), nn.ReLU(True),
            nn.Linear(16, 3)
        )
        self.decode_layers = nn.Sequential(
            nn.Linear(3, 16), nn.ReLU(True),
            nn.Linear(16, 32), nn.ReLU(True),
            nn.Linear(32, 64), nn.ReLU(True),
            nn.Linear(64, 128), nn.ReLU(True),
            nn.Linear(128, 256), nn.ReLU(True),
            nn.Linear(256, features_dim), nn.Sigmoid()
        )

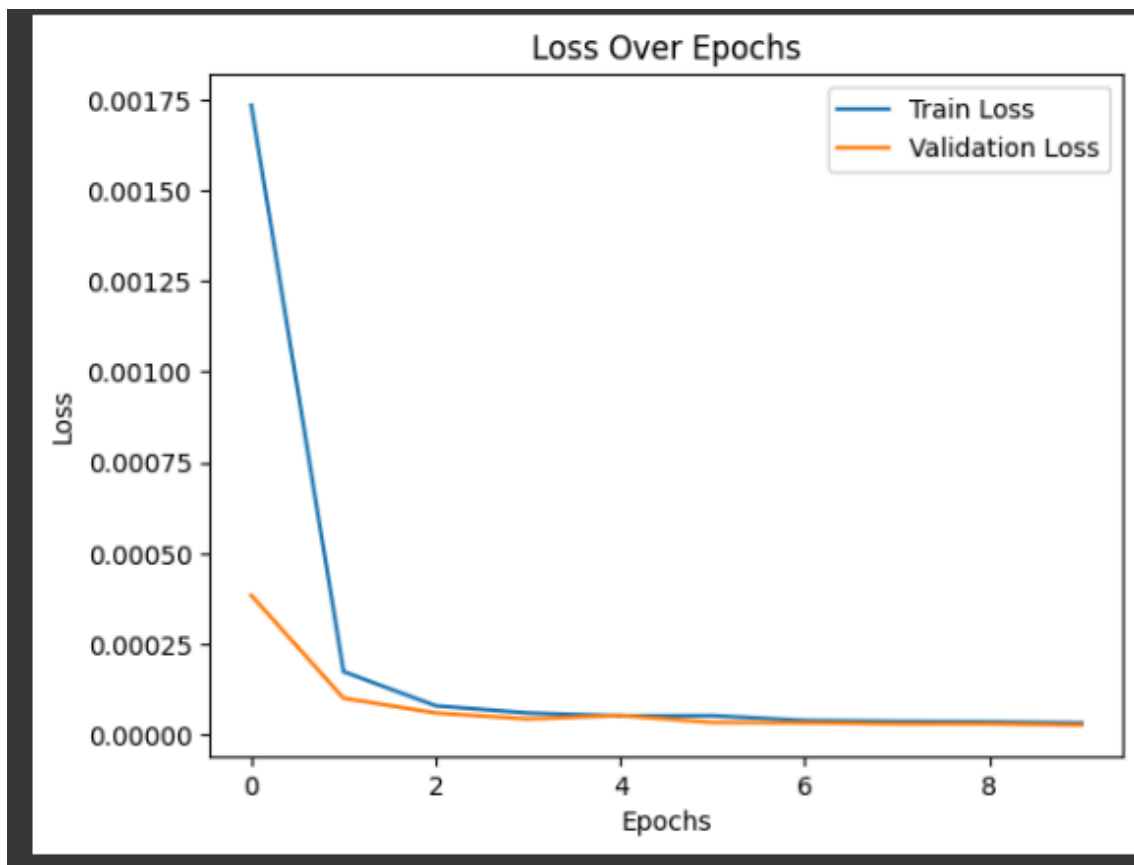
    def forward(self, inputs):
        encoded = self.encode_layers(inputs)
        decoded = self.decode_layers(encoded)
        return decoded
```

### 3) R2 Score, Reconstruction error and losses:

#### Simple Autoencoder Model:

Losses:

```
Epoch 1/10, Train Loss: 0.0017, Val Loss: 0.0004  
Epoch 2/10, Train Loss: 0.0002, Val Loss: 0.0001  
Epoch 3/10, Train Loss: 0.0001, Val Loss: 0.0001  
Epoch 4/10, Train Loss: 0.0001, Val Loss: 0.0000  
Epoch 5/10, Train Loss: 0.0001, Val Loss: 0.0001  
Epoch 6/10, Train Loss: 0.0001, Val Loss: 0.0000  
Epoch 7/10, Train Loss: 0.0000, Val Loss: 0.0000  
Epoch 8/10, Train Loss: 0.0000, Val Loss: 0.0000  
Epoch 9/10, Train Loss: 0.0000, Val Loss: 0.0000  
Epoch 10/10, Train Loss: 0.0000, Val Loss: 0.0000
```



R2 score:

```
R^2 Score on Test Data: 0.8491
```

Reconstruction error:



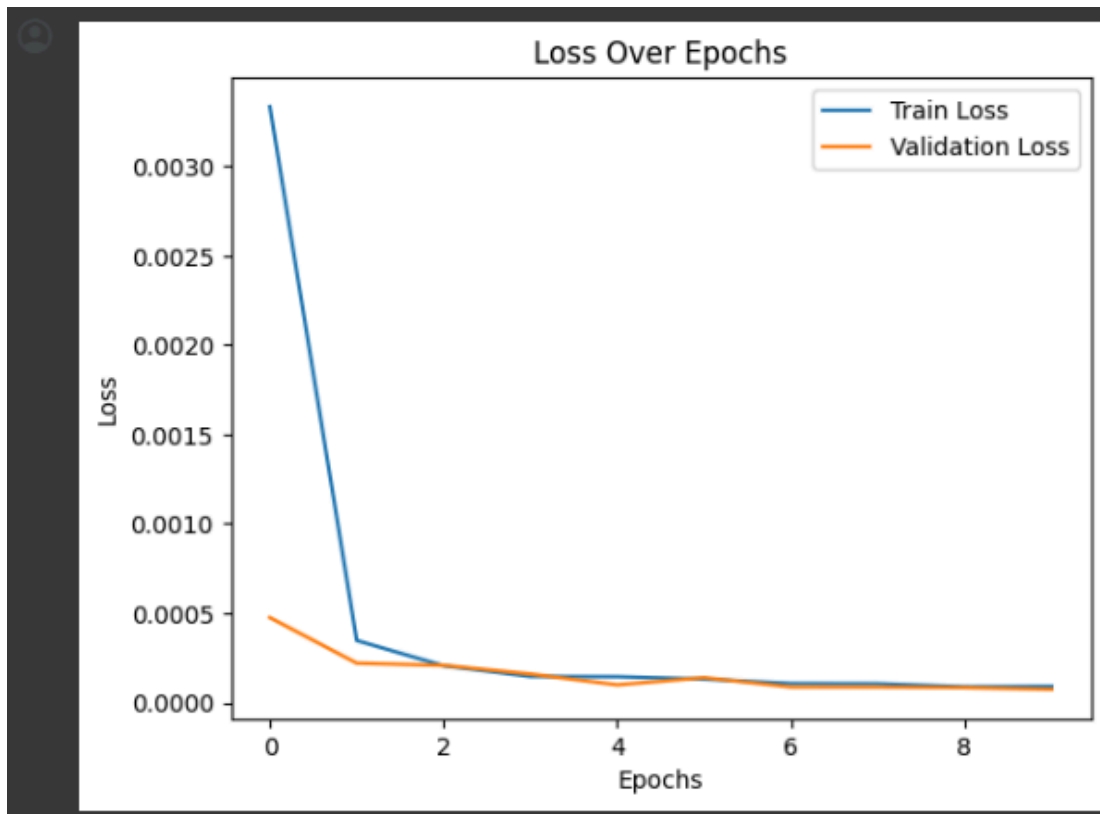
```
[ ] reconstruction_error = evaluate_reconstruction_error(model, test_loader, criterion, device)
print(f'Reconstruction Error (MSE) on Test Data: {reconstruction_error:.4f}')
```

Reconstruction Error (MSE) on Test Data: 0.0000

## LSTM Autoencoder (Temporal) model:

### Losses

```
Epoch 1/10, Train Loss: 0.0033, Val Loss: 0.0005
Epoch 2/10, Train Loss: 0.0003, Val Loss: 0.0002
Epoch 3/10, Train Loss: 0.0002, Val Loss: 0.0002
Epoch 4/10, Train Loss: 0.0001, Val Loss: 0.0002
Epoch 5/10, Train Loss: 0.0001, Val Loss: 0.0001
Epoch 6/10, Train Loss: 0.0001, Val Loss: 0.0001
Epoch 7/10, Train Loss: 0.0001, Val Loss: 0.0001
Epoch 8/10, Train Loss: 0.0001, Val Loss: 0.0001
Epoch 9/10, Train Loss: 0.0001, Val Loss: 0.0001
Epoch 10/10, Train Loss: 0.0001, Val Loss: 0.0001
```



R2 score:



R<sup>2</sup> Score on Test Data: 0.8362

### Reconstruction Error:

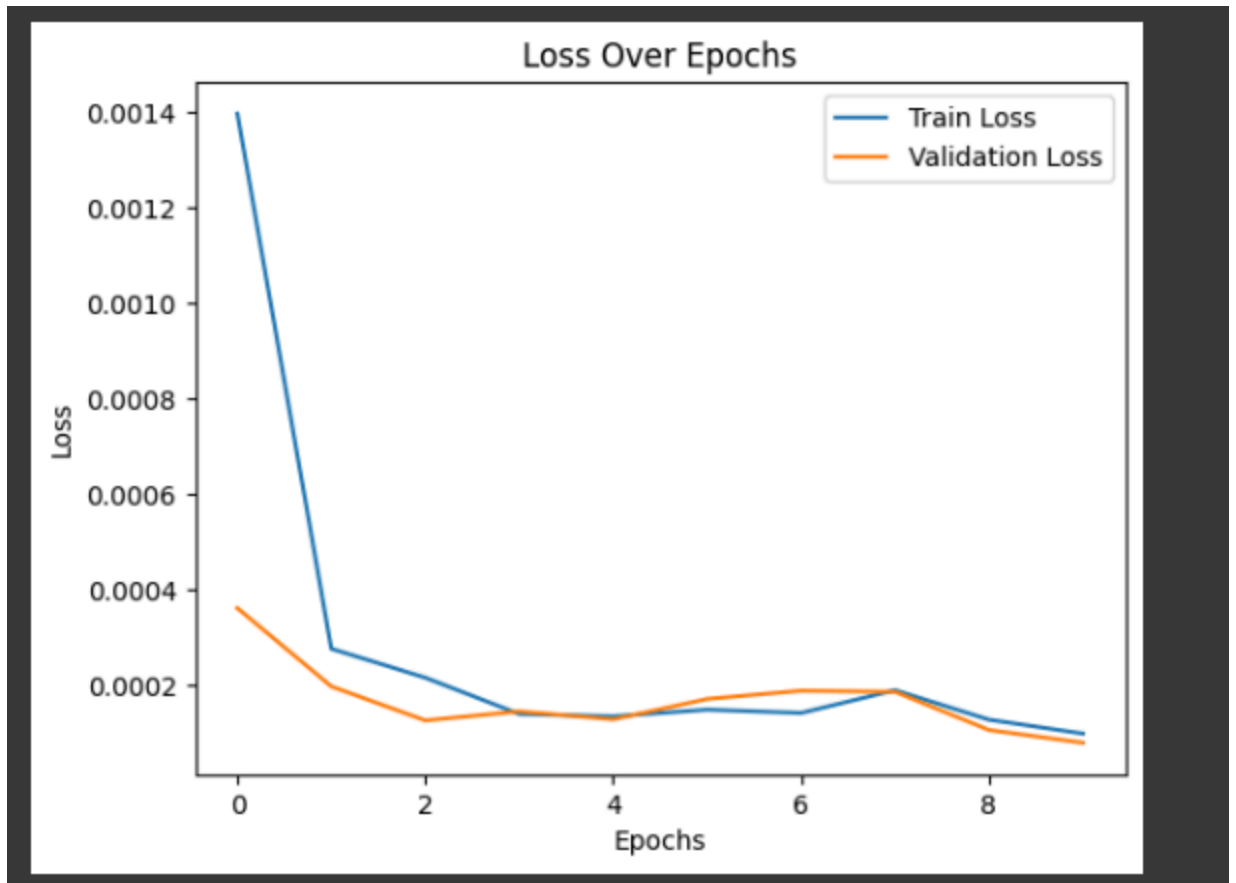


Reconstruction Error (MSE) on Test Data: 0.0001

### Compact AutoEncoder Model:



```
Epoch 1/10, Train Loss: 0.0014, Val Loss: 0.0004  
Epoch 2/10, Train Loss: 0.0003, Val Loss: 0.0002  
Epoch 3/10, Train Loss: 0.0002, Val Loss: 0.0001  
Epoch 4/10, Train Loss: 0.0001, Val Loss: 0.0001  
Epoch 5/10, Train Loss: 0.0001, Val Loss: 0.0001  
Epoch 6/10, Train Loss: 0.0001, Val Loss: 0.0002  
Epoch 7/10, Train Loss: 0.0001, Val Loss: 0.0002  
Epoch 8/10, Train Loss: 0.0002, Val Loss: 0.0002  
Epoch 9/10, Train Loss: 0.0001, Val Loss: 0.0001  
Epoch 10/10, Train Loss: 0.0001, Val Loss: 0.0001
```



**R2-score:**

R<sup>2</sup> Score on Test Data: 0.8214

**Reconstruction Error:**

Reconstruction Error (MSE) on Test Data: 0.0001

- 4) Autoencoders are good when it comes to spotting the odd ones out in the data. They learn by trying to copy their input to their output, getting really good at recognizing what's normal. So, when something unusual pops up, they stumble, and that's how we catch anomalies.

**Strengths:**

Self-learning: They can learn what's typical without labeled data, which makes them ideal in scenarios when you have a ton of data and no one to interpret it.

Flexibility: They are capable of handling several kinds of data, including numbers, images, and sounds.

Feature Learning: Autoencoders possess an intelligent ability to discern the subtle, sometimes imperceptible, underlying patterns or features in the input.

### **Limitations:**



Noise Sensitivity: When they pick up on data noise too quickly, they may fail to notice some irregularities because they mistake it for normalcy.

Complex Anomalies: Autoencoders may become confused and allow some anomalies to pass through if the unusual data is somewhat too similar to the typical data.

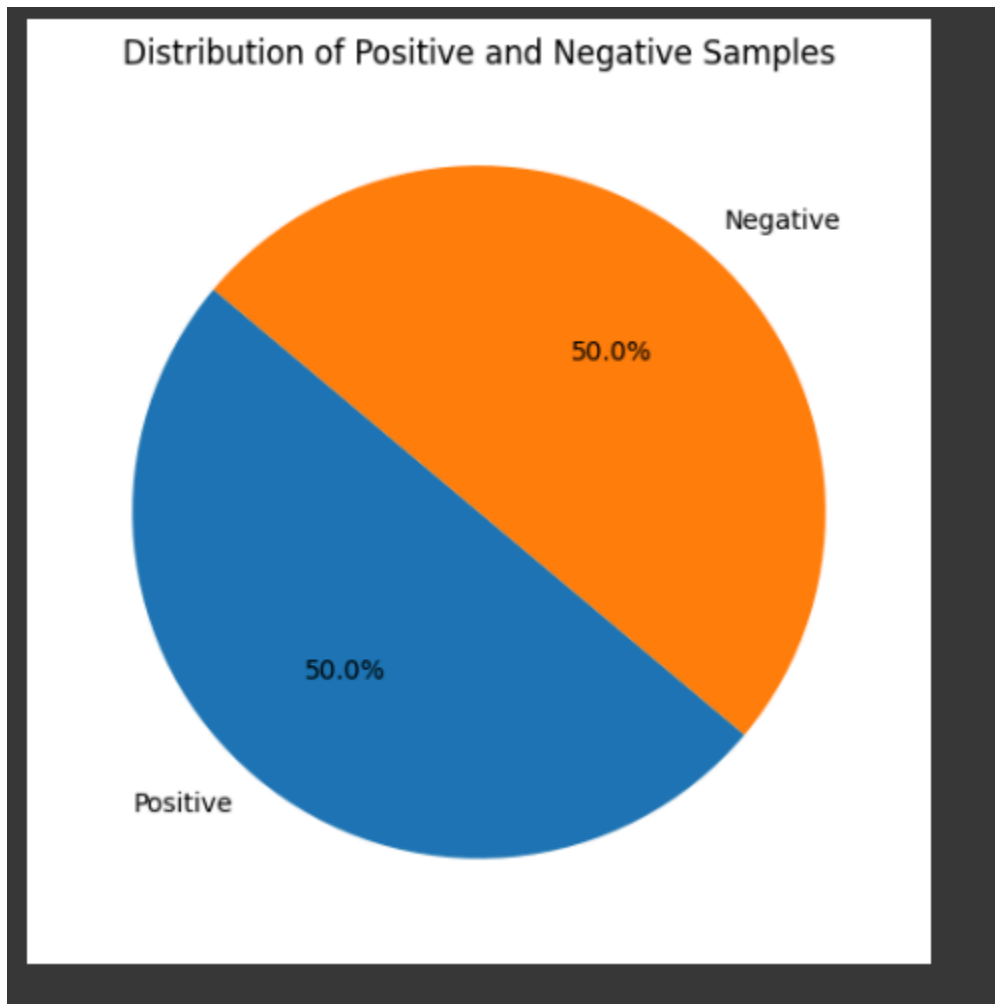
Rich in Resources: It can require a significant amount of processing power and time to train them, particularly the complex ones with several layers.

## Part -4

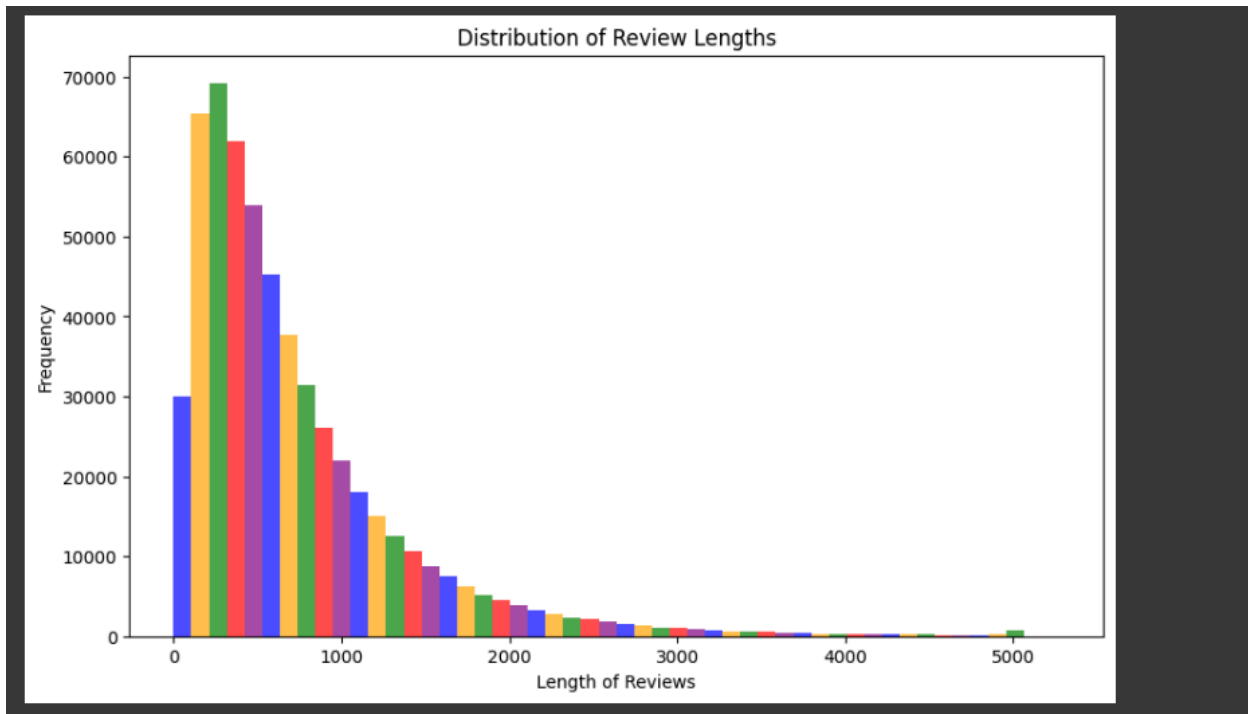
Some preprocessing and visualizations:

	<code>train_df.describe()</code>																		
	<table><thead><tr><th></th><th>polarity</th></tr></thead><tbody><tr><td>count</td><td>560000.0</td></tr><tr><td>mean</td><td>1.5</td></tr><tr><td>std</td><td>0.5</td></tr><tr><td>min</td><td>1.0</td></tr><tr><td>25%</td><td>1.0</td></tr><tr><td>50%</td><td>1.5</td></tr><tr><td>75%</td><td>2.0</td></tr><tr><td>max</td><td>2.0</td></tr></tbody></table>		polarity	count	560000.0	mean	1.5	std	0.5	min	1.0	25%	1.0	50%	1.5	75%	2.0	max	2.0
	polarity																		
count	560000.0																		
mean	1.5																		
std	0.5																		
min	1.0																		
25%	1.0																		
50%	1.5																		
75%	2.0																		
max	2.0																		
[ ]	<code>test_df.describe()</code>																		
	<table><thead><tr><th></th><th>polarity</th></tr></thead><tbody><tr><td>count</td><td>38000.000000</td></tr><tr><td>mean</td><td>1.500000</td></tr><tr><td>std</td><td>0.500007</td></tr><tr><td>min</td><td>1.000000</td></tr><tr><td>25%</td><td>1.000000</td></tr><tr><td>50%</td><td>1.500000</td></tr><tr><td>75%</td><td>2.000000</td></tr><tr><td>max</td><td>2.000000</td></tr></tbody></table>		polarity	count	38000.000000	mean	1.500000	std	0.500007	min	1.000000	25%	1.000000	50%	1.500000	75%	2.000000	max	2.000000
	polarity																		
count	38000.000000																		
mean	1.500000																		
std	0.500007																		
min	1.000000																		
25%	1.000000																		
50%	1.500000																		
75%	2.000000																		
max	2.000000																		

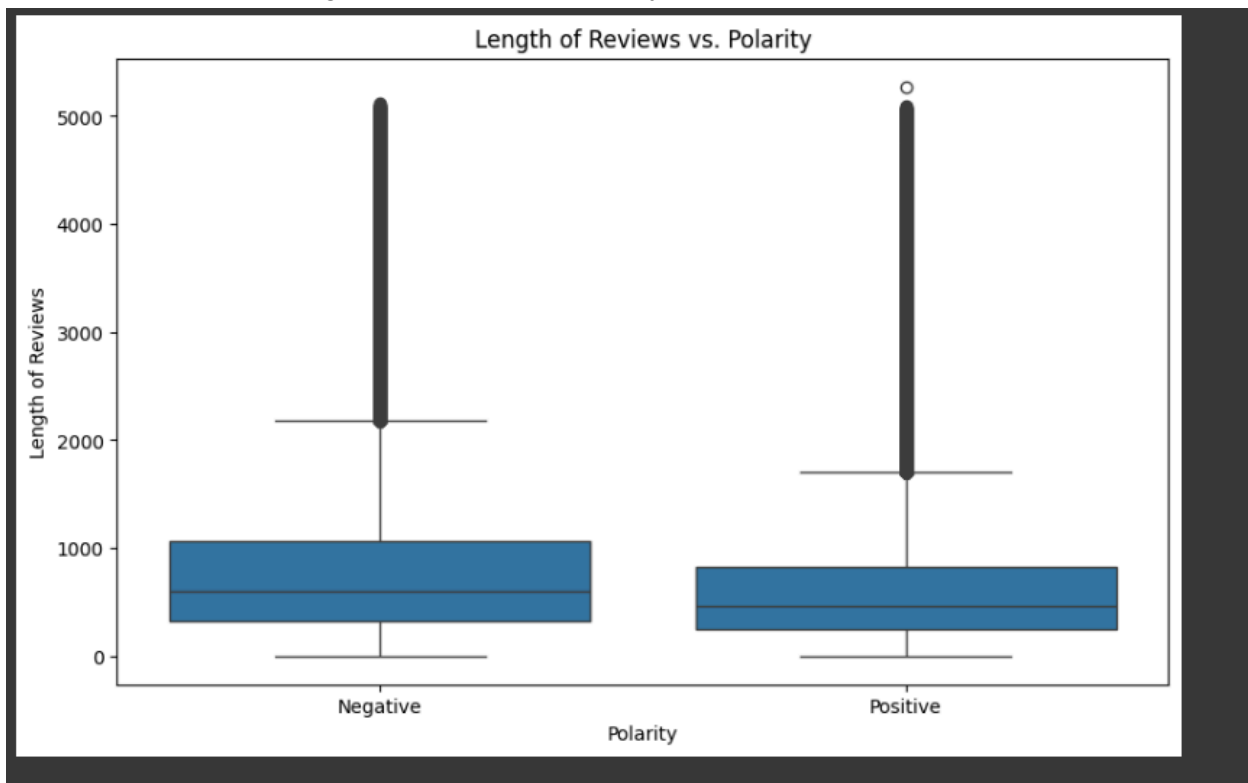
Pie Chart for Positive and Negative Samples



Distribution of review lengths:



Box plot between the length of reviews and polarity:



Converting into lowercase, making a translation table that'll be applied to remove, punctuation characters from the text, tokenization into one or two syllable containing words, remove stopwords by checking if each word is in nltk.stopwords

```
[ ] def preprocess_text(text):
    #lower case conversion
    text = text.lower()

    #making a translation table that'll be applied to remove
    #punctuation characters from the text
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Tokenization into one or two syllable containing words
    tokens = word_tokenize(text)

    # Remove stopwords by checking if each word is in nltk.stopwords
    stop_words = set(stopwords.words('english'))
    tokens = [word for word in tokens if word not in stop_words]

    return tokens
```

```
[ ] train_df['pr_review'] = train_df['review'].apply(preprocess_text)
    test_df['pr_review'] = test_df['review'].apply(preprocess_text)
```

```
[ ] train_df.head()
```

	polarity	review	review_length	pr_review
0	0	Unfortunately, the frustration of being Dr. Go...	643	[unfortunately, frustration, dr, goldbergs, pa...
1	1	Been going to Dr. Goldberg for over 10 years. ...	495	[going, dr, goldberg, 10, years, think, one, 1...
2	0	I don't know what Dr. Goldberg was like before...	1143	[dont, know, dr, goldberg, like, moving, arizo...
3	0	I'm writing this review to give you a heads up...	1050	[im, writing, review, give, heads, see, doctor...
4	1	All the food is great here. But the best thing...	425	[food, great, best, thing, wings, wings, simpl...

## Transformer architecture:

**Embedding Layer:** Converts tokens to dense vectors. Params: Vocabulary size (vsize), Embedding dimension (edim).

**Positional Embedding:** Adds sequence order info. Params: Max sequence length (max\_length), Embedding dimension (edim).

**Transformer Encoder Layers:** Stacked self-attention and feed-forward layers. Params: Embedding dimension (edim), Attention heads (nhead), Feed-forward dimension (fdim), Layers (nlayers).

**Classification Layer:** Linear classifier. Params: Input dimension (same as edim), Number of classes (nclasses).

**Training:** Cross-entropy loss, Adam optimizer. Accuracy computed via maximum predicted probabilities. Epochs (num\_epochs): 3.



```

class TransformerModel(nn.Module):
    def __init__(self, vsize, edim, nhead, nlayers, fdim, nclasses, max_length):
        super(TransformerModel, self).__init__()
        self.embedding = nn.Embedding(vsize, edim)
        self.pos_embedding = nn.Parameter(torch.zeros(max_length, edim))
        encoder_layer = TransformerEncoderLayer(edim, nhead, fdim)
        self.encoder = TransformerEncoder(encoder_layer, nlayers)
        self.classifier = nn.Linear(edim, nclasses)

    def forward(self, src):
        x = self.embedding(src)
        pos_encoding = self.pos_embedding.unsqueeze(0).expand(src.size(0), -1, -1)
        x += pos_encoding[:, :src.size(1)]
        x = self.encoder(x)
        x = x.mean(dim=1)
        output = self.classifier(x)
        return output

```

These are the accuracies and losses for base model:

```

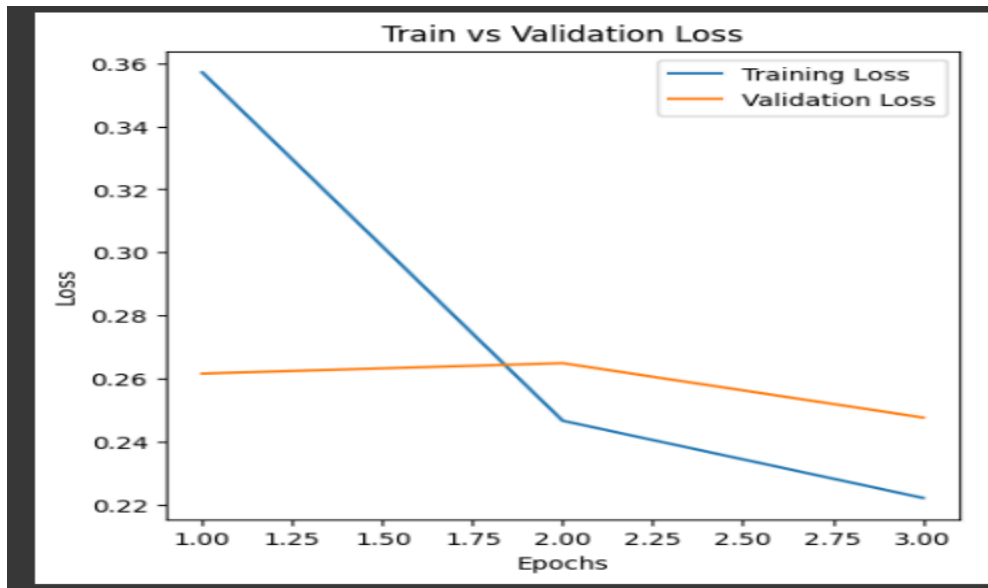
Epoch 1, Train Loss: 0.35710496050385493, Train Accuracy: 0.8449754464285715%, Val Loss: 0.26159680581199274, Val Accuracy: 0.8975714285714286%
Epoch 2, Train Loss: 0.24665217609889806, Train Accuracy: 0.9032589285714285%, Val Loss: 0.26490333982237746, Val Accuracy: 0.8973839285714286%
Epoch 3, Train Loss: 0.22217175107435988, Train Accuracy: 0.9141897321428571%, Val Loss: 0.2476577178968915, Val Accuracy: 0.9033571428571429%

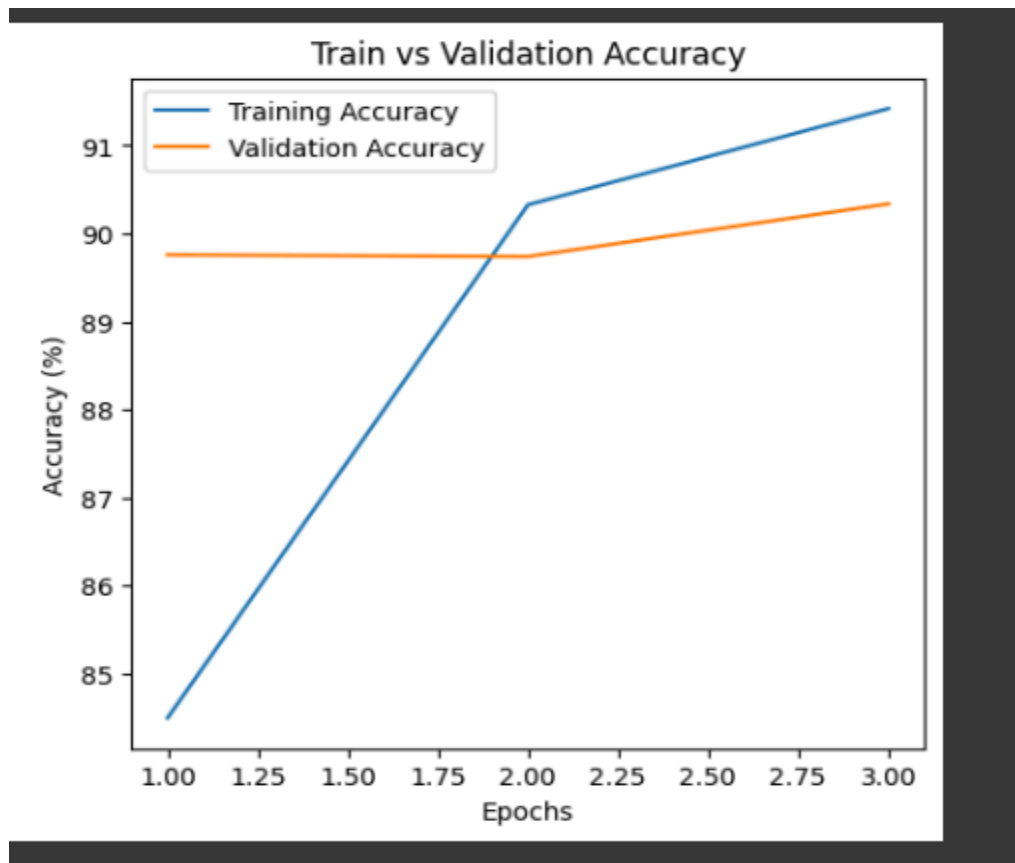
```

Testing Loss: 0.2981

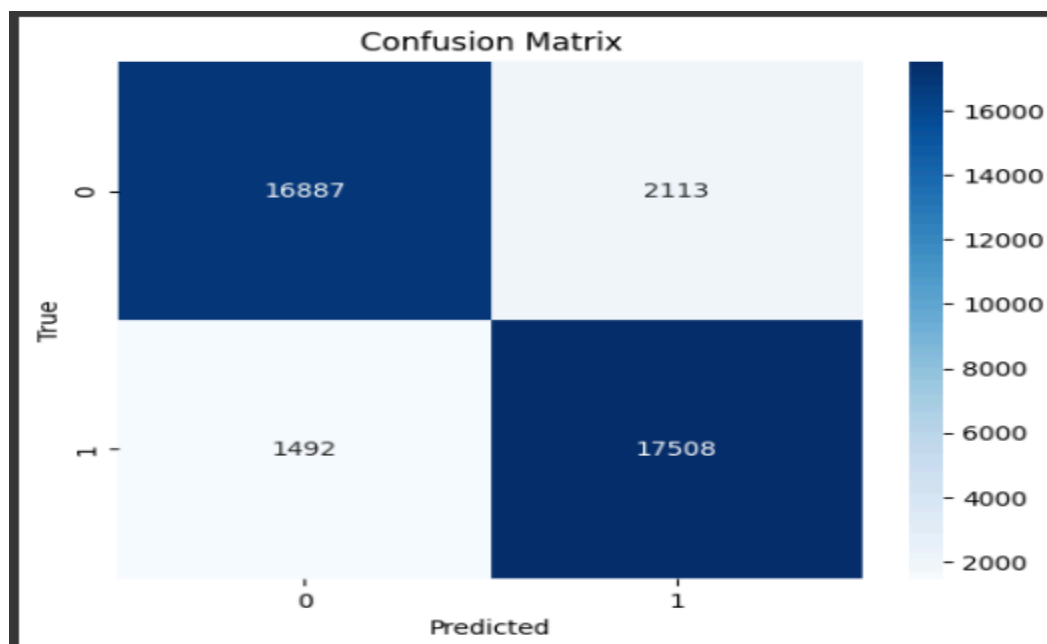
Testing Accuracy: 88.02%

Visualization Graphs for accuracies and losses :





Confusion matrix:



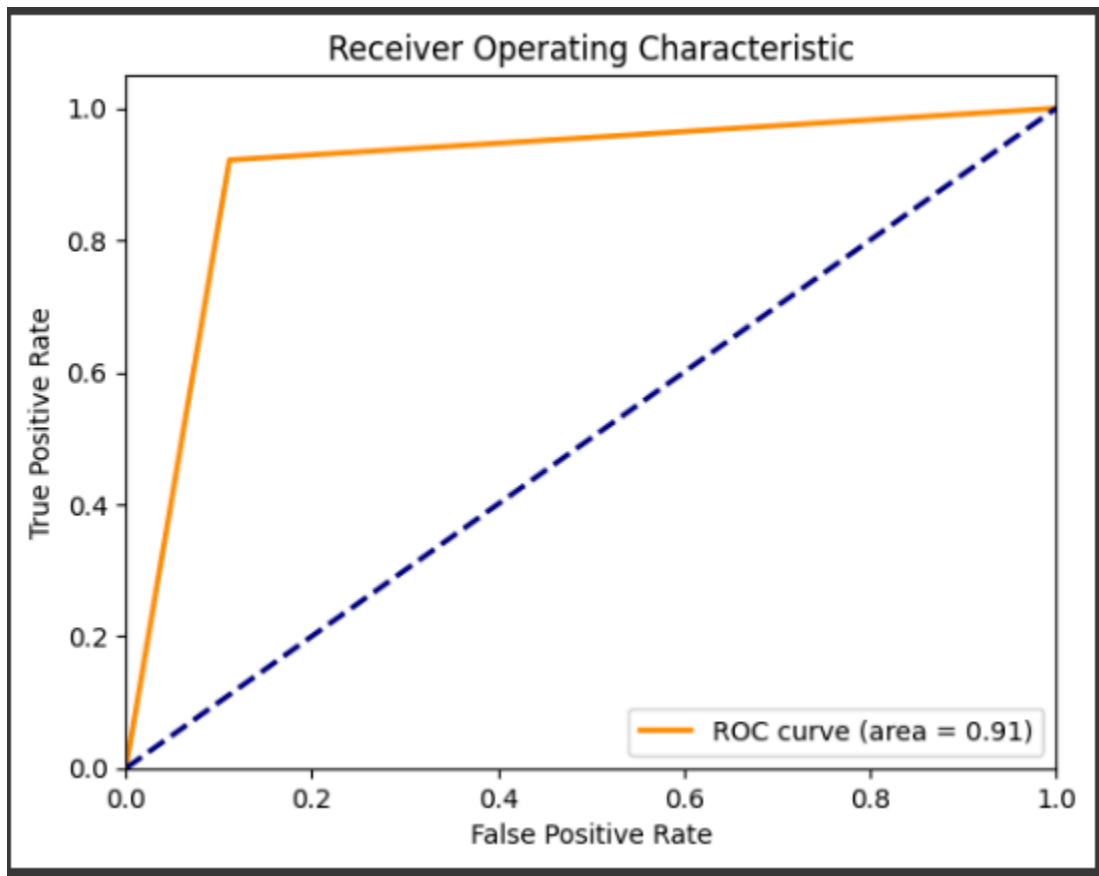
## Scores:

Precision: 0.8923092604862137

Recall: 0.9214736842105263

F1 Score: 0.9066570000776779

## ROC Curve:



## Dropout - accuracies and losses:

Epoch 1, Train Loss: 0.3519810749233833, Train Accuracy: 0.8479732142857143%, Val Loss: 0.26987332255073954, Val Accuracy: 0.8935535714285714%

## L2 regularization:

Epoch 1, Train Loss: 0.37279429386981894, Train Accuracy: 0.8297790178571428%, Val Loss: 0.2970106146080153, Val Accuracy: 0.8831696428571428%

## Early Stopping:

Epoch 1, Train Loss: 0.31256268132451387, Train Accuracy: 0.8741540178571429%, Val Loss: 0.2976584136337042, Val Accuracy: 0.8803125%  
Epoch 2, Train Loss: 0.3090190281367728, Train Accuracy: 0.8748325892857143%, Val Loss: 0.3033824353005205, Val Accuracy: 0.8780982142857143%  
Early stopping triggered after 2 epochs.

**Model Performance:****Base Model:**

Highest Validation Accuracy: 90.34%.

Stable improvement over epochs.

**Dropout:**

Decrease in Training and Validation Accuracies.

Validation Accuracy: 89.36%.

Slight drop in performance.

**L2 Regularization:**

Reduction in Training and Validation Accuracies.

Validation Accuracy: 88.32%.

Possibly overly penalizing the models parameters.

**Early Stopping:**

Stopped after 2 epochs.

Lower Validation Accuracy: 88.03%.

Model may benefit from further training beyond stopped point.