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

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

Part 4: Building Transformer with PyTorch

Dataset Used:
ag_news_csv.tar

Origin of Dataset:
AG is a news article collection that exceeds one million. ComeToMyHead has been collecting news stories for more than a year from over 2000 news sources. Since July 2004, ComeToMyHead has operated as a search engine for academic news. The academic community is providing the dataset for research purposes in the fields of xml, data compression, data streaming, information retrieval (ranking, search, etc.), data mining (clustering, classification, etc.), and any other non-commercial activity. Please click this link: http://www.di.unipi.it/~gulli/AG_corpus_of_news_articles.html for additional information.

Description:
Four of the largest classes from the original corpus are selected to create the AG's news topic classification dataset. 1,900 testing samples and 30,000 training samples are included in each class. There are 120,000 training samples and 7,600 testing samples in all.

A list of classes for each label may be found in the file classes.txt.

Every training sample is included in the files train.csv and test.csv as comma-separated values. They have three columns that represent the class index (1–4), description, and title. Double quotes ("") are used to escape the title and description, while two double quotes ("") are used to escape any inside double quotations. New lines are terminated with a backslash and the "n" character ("\n").

Data Preprocessing and Visualization

```
# Defining columns and dataframes

columns = ['Index',"Title",'Description']
df_train = pd.read_csv('train.csv',names=columns)
df_test = pd.read_csv('test.csv',names=columns)
```

```
# Generating fourth column text by concatinating "Title" and "Article"
# Performed on both train and test datasets

df_train['Text'] = df_train['Title'] + " " + df_train['Description']
df_test['Text'] = df_test['Title'] + " " + df_test['Description']
```

```
# Preparing final dataset by concatinating "Class" and "Test" columns of Train and Test Datasets

df= pd.concat([df_train[['Index', 'Text']], df_test[['Index', 'Text']]])
```

```
# Displaying our final dataframe
df
```

	Index	Text
0	3	Wall St. Bears Claw Back Into the Black (Reute...
1	3	Carlyle Looks Toward Commercial Aerospace (Reu...
2	3	Oil and Economy Cloud Stocks' Outlook (Reuters...
3	3	Iraq Halts Oil Exports from Main Southern Pipe...
4	3	Oil prices soar to all-time record, posing new...
...	...	...
7595	1	Around the world Ukrainian presidential candid...
7596	2	Void is filled with Clement With the supply of...
7597	2	Martinez leaves bitter Like Roger Clemens did ...
7598	3	5 of arthritis patients in Singapore take Bext...
7599	3	EBay gets into rentals EBay plans to buy the a...

127600 rows × 2 columns

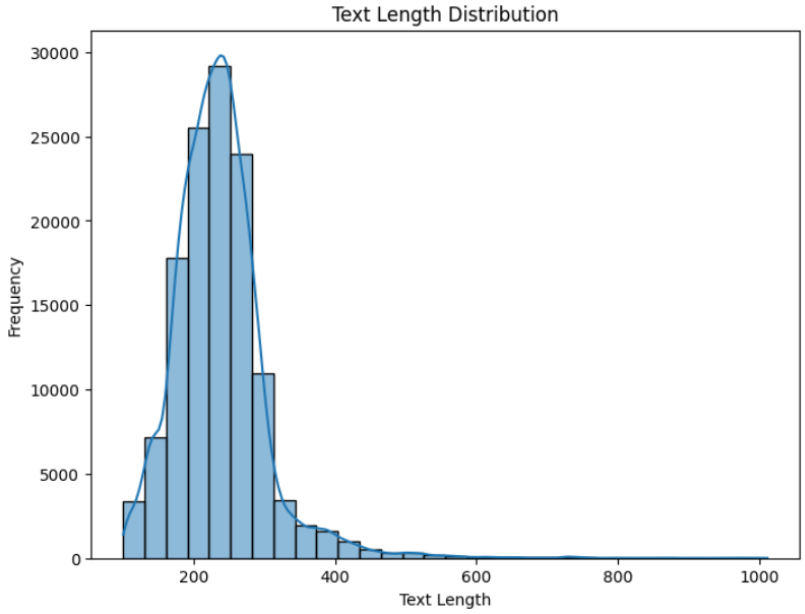
```
# Information of dataframe
# Columns against datatype

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 127600 entries, 0 to 7599
Data columns (total 2 columns):
 #   Column  Non-Null Count   Dtype 
---  ------  --------------   ----- 
 0   Index   127600 non-null  int64 
 1   Text    127600 non-null  object
dtypes: int64(1), object(1)
memory usage: 2.9+ MB
```

```
# Main Statistics of the dataset

df.describe()
```

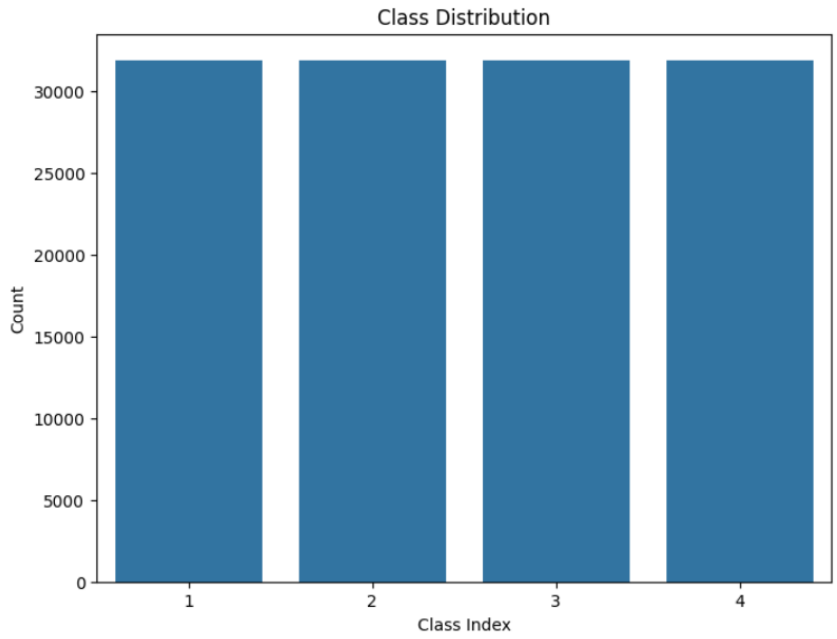
	Index
count	127600.000000
mean	2.500000
std	1.118038
min	1.000000
25%	1.750000
50%	2.500000
75%	3.250000
max	4.000000

Text Length Distribution Plot



Most Frequent Words

Class distribution plot



Architecture of defined transformers:

Transformer Model:

```
# Defining Transformer class

class transformerModel(nn.Module):
    def __init__(self, vocab_size, num_classes):
        super().__init__()
        self.embed_dim = 256
        self.embedding = nn.Embedding(vocab_size, self.embed_dim)
        self.pos_encoder = positional_encoding(self.embed_dim)
        self.encoder_layer = nn.TransformerEncoderLayer(d_model=self.embed_dim, nhead=8, dim_feedforward=512)
        self.transformer_encoder = nn.TransformerEncoder(self.encoder_layer, num_layers=6)
        self.decoder_layer = nn.TransformerDecoderLayer(d_model=self.embed_dim, nhead=8, dim_feedforward=512)
        self.transformer_decoder = nn.TransformerDecoder(self.decoder_layer, num_layers=6)
        self.fc_out = nn.Linear(self.embed_dim, num_classes)

    def forward(self, src):
        src = self.embedding(src) * math.sqrt(self.embed_dim)
        src = self.pos_encoder(src)
        memory = self.transformer_encoder(src)
        output = self.transformer_decoder(src, memory)
        output = output.mean(dim=1)
        output = self.fc_out(output)
        return output
```

The __init__ method

The class constructor is this method. It sets the model's layers and parameters to their initial values.

vocab_size: The total number of distinct tokens in the input text data, or the size of the vocabulary.

num_classes: The total number of classes that the classification task will produce.

The word embeddings' dimensionality is indicated by self.embed_dim. It is set to 256 in this instance.

self.embedding: An embedding layer that creates a dense vector representation for every token in the input sequence.
A positional encoding layer to add positional information to the input embeddings is called self.pos_encoder.
self.encoder_layer: The Transformer encoder's single layer. It is composed of feedforward neural network layers and multi-head self-attention.
The Transformer encoder module (self.transformer_encoder) processes the input sequence by stacking multiple encoder layers.
self.decoder_layer: A single Transformer decoder layer. Feedforward neural network layers and multi-head self-attention are also included.
self.transformer_decoder: The Transformer decoder module, which creates the output sequence by stacking several decoder layers.
self.fc_out: a completely linked layer that associates the output classes with the final hidden state.

forward technique

This technique outlines the model's forward pass, or how input data moves through its layers and generates output.
src: The token index input sequence. In the Transformer architecture, it stands for the source sequence.
The input sequence is scaled by the square root of the embedding dimension after first passing through the embedding layer.
Positional information is incorporated into the embedded sequence by adding the positional encoding.
A memory tensor is produced by the Transformer encoder after processing the embedded sequence.
The input sequence and the memory tensor are used by the Transformer decoder to produce the output sequence.
Next, the output sequence is averaged over the length of the sequence.
Ultimately, the final output logits for classification are obtained by passing the averaged output via a linear layer (self.fc_out).

```
# Defining Optimizer and Loss Function

optimizer = torch.optim.SGD(BaseModel1.parameters(), lr=0.01, momentum=0.9)
criterion = nn.CrossEntropyLoss()
```

Function for training the model:

```python
# Function for training the model

def training(model, train_loader, optimizer, criterion, device):
    model.train()
    train_losses, train_accuracies = [], []
    for texts, labels in train_loader:
        texts, labels = texts.to(device), labels.to(device)
        optimizer.zero_grad()
        output = model(texts)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        train_losses.append(loss.item())
        _, predicted = torch.max(output, 1)
        accuracy = (predicted == labels).sum().item() / labels.size(0)
        train_accuracies.append(accuracy)

    avg_loss = sum(train_losses) / len(train_losses)
    avg_acc = sum(train_accuracies) / len(train_accuracies)
    return avg_loss, avg_acc
```

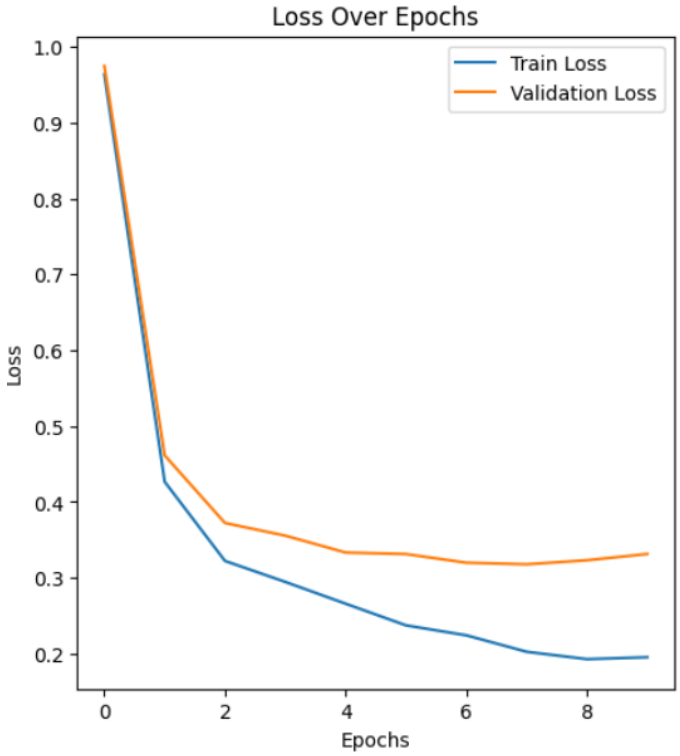
Function for evaluating the model:

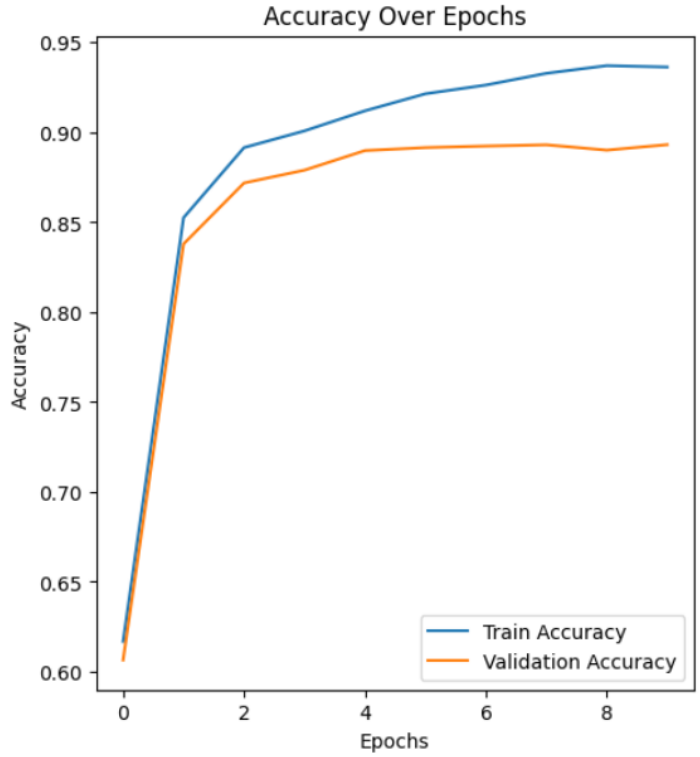
```python
# Function for evaluating the model

def evaluate(model, loader, criterion, device):
    model.eval()
    total_loss, total_correct, total_count = 0, 0, 0
    all_preds, all_probs, all_labels = [], [], []
    with torch.no_grad():
        for texts, labels in loader:
            texts, labels = texts.to(device), labels.to(device)
            output = model(texts)
            loss = criterion(output, labels)
            total_loss += loss.item()

            _, predicted = torch.max(output, 1)
            probabilities = nn.functional.softmax(output, dim=1)
            total_correct += (predicted == labels).sum().item()
            total_count += labels.size(0)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_probs.extend(probabilities.cpu().numpy())

    avg_loss = total_loss / len(loader)
    avg_acc = total_correct / total_count
    return avg_loss, avg_acc, all_labels, all_preds, all_probs
```

Results for training loop with 10 epochs:

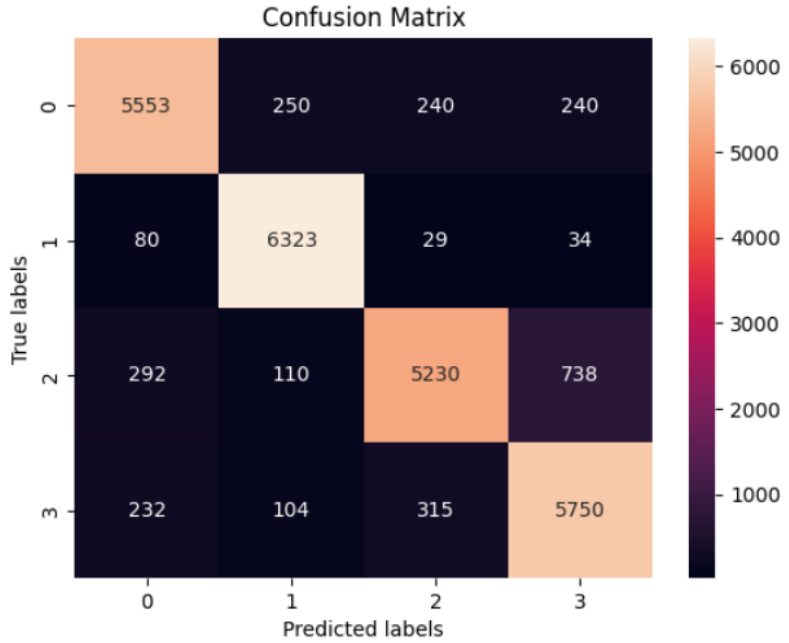
```
Epoch 1: Train Loss: 0.9640, Train Acc: 0.6169, Val Loss: 0.9750, Val Acc: 0.6065
Epoch 2: Train Loss: 0.4266, Train Acc: 0.8526, Val Loss: 0.4614, Val Acc: 0.8379
Epoch 3: Train Loss: 0.3222, Train Acc: 0.8915, Val Loss: 0.3722, Val Acc: 0.8719
Epoch 4: Train Loss: 0.2945, Train Acc: 0.9009, Val Loss: 0.3555, Val Acc: 0.8790
Epoch 5: Train Loss: 0.2657, Train Acc: 0.9120, Val Loss: 0.3332, Val Acc: 0.8899
Epoch 6: Train Loss: 0.2372, Train Acc: 0.9215, Val Loss: 0.3312, Val Acc: 0.8915
Epoch 7: Train Loss: 0.2241, Train Acc: 0.9263, Val Loss: 0.3199, Val Acc: 0.8924
Epoch 8: Train Loss: 0.2023, Train Acc: 0.9329, Val Loss: 0.3177, Val Acc: 0.8931
Epoch 9: Train Loss: 0.1926, Train Acc: 0.9371, Val Loss: 0.3231, Val Acc: 0.8901
Epoch 10: Train Loss: 0.1951, Train Acc: 0.9364, Val Loss: 0.3313, Val Acc: 0.8932
```

Plotting testing loss and testing accuracy
Loss over epochs
Accuracy over epochs

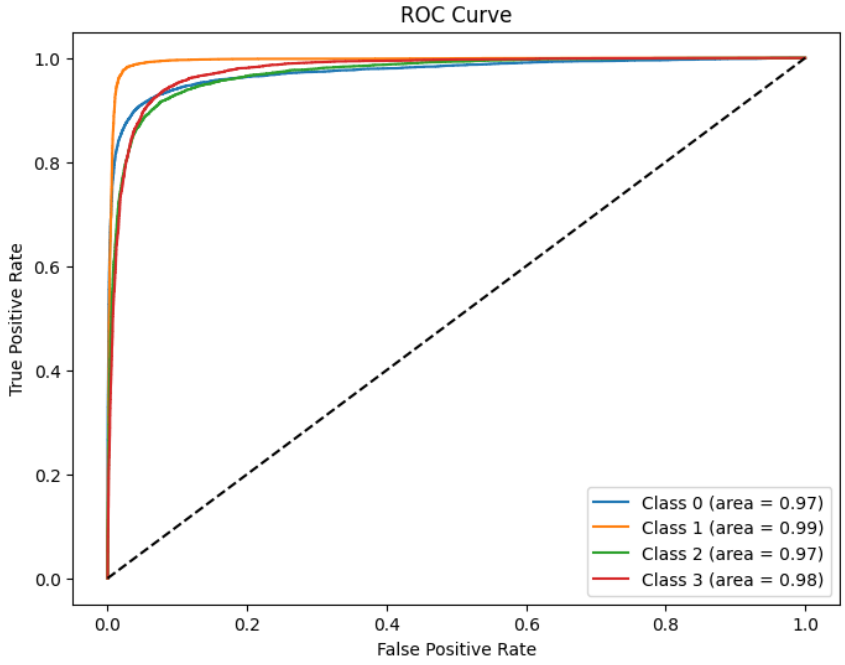




Confusion Matrix:



Receiver Operating Characteristic Curve:




```
Precision: 0.8959, Recall: 0.8953, F1-Score: 0.8948
```

Dropout Transformer Model:

```
# Dropout Transformer Model

class positional_encoding(nn.Module):
    def __init__(self, d_model, max_len=7000):
        super().__init__()
        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, d_model)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
```

The __init__ method

The class constructor is this method. The positional encodings are initialized by it.

d_model: The embeddings' dimensionality in the model.

max_len: The longest sequence that can be entered. 7000 is the default value, although it can be changed based on the dataset's maximum sequence length.

position: A tensor that holds the locations ranging from 0 to max_len - 1. div_term: A tensor that holds the exponential term that was utilized in the computation of positional encoding.

pe: The shape of the tensor to hold the positional encodings is (max_len, d_model).

Depending on the position and dimension of the embeddings, different frequency sine and cosine functions are used to compute the positional encodings.

The positional encodings that are obtained are kept in the pe tensor.

forward technique

This method describes the module's forward pass, or the path that input data takes via the layer of positional encoding.

x: The input embeddings, which are usually of the form (internal_dim, batch_size, sequence_length).

Along the sequence length dimension, the positional encodings kept in the pe tensor are added to the input embeddings.

The input embeddings plus the positional encodings add up to the output.

The pe tensor is registered as a module buffer using the register_buffer function. This guarantees that the parameters of the module are saved on the same device as the positional encodings.

Second Transformer Model:

```
# Second Transformer Model

class transformerModel2(nn.Module):
    def __init__(self, vocab_size, num_classes, dropout_rate=0.1):
        super().__init__()
        self.embed_dim = 256
        self.embedding = nn.Embedding(vocab_size, self.embed_dim)
        self.pos_encoder = positional_encoding(self.embed_dim)
        self.dropout = nn.Dropout(dropout_rate)
        self.encoder_layer = nn.TransformerEncoderLayer(d_model=self.embed_dim, nhead=8, dim_feedforward=512, dropout=dropout_rate)
        self.transformer_encoder = nn.TransformerEncoder(self.encoder_layer, num_layers=6)
        self.decoder_layer = nn.TransformerDecoderLayer(d_model=self.embed_dim, nhead=8, dim_feedforward=512, dropout=dropout_rate)
        self.transformer_decoder = nn.TransformerDecoder(self.decoder_layer, num_layers=6)
        self.fc_out = nn.Linear(self.embed_dim, num_classes)

    def forward(self, src):
        src = self.embedding(src) * math.sqrt(self.embed_dim)
        src = self.pos_encoder(src)
        src = self.dropout(src)
        memory = self.transformer_encoder(src)
        output = self.transformer_decoder(src, memory)
        output = output.mean(dim=1)
        output = self.fc_out(output)
        return output
```

The __init__ method

The class constructor is this method. It sets the model's layers and parameters to their initial values.

vocab_size: The total number of distinct tokens in the input text data, or the size of the vocabulary.

num_classes: The total number of classes that the classification task will produce.

dropout_rate: The dropout rate that will be used for the Transformer layers and input embeddings. By default, it is set at 0.1.

The word embeddings' dimensionality is indicated by self.embed_dim. It is set to 256 in this instance.

self.embedding: An embedding layer that creates a dense vector representation for every token in the input sequence.

A positional encoding layer to add positional information to the input embeddings is called self.pos_encoder.

self.dropout: A dropout layer that regularizes dropouts for the Transformer layers and input embeddings.

self.encoder_layer: The Transformer encoder's single layer with dropout regularization.

self.transformer_encoder: The dropout regularization-equipped Transformer encoder module, which processes the input sequence by stacking numerous encoder layers.

self.decoder_layer: A Transformer decoder layer with dropout regularization on a single layer.

self.transformer_decoder: This module, which stacks several decoder layers to produce the output sequence, is the Transformer decoder with dropout regularization.

self.fc_out: A fully connected layer that associates the output classes with the final hidden state.

forward technique

This technique outlines the model's forward pass, or how input data moves through its layers and generates output.

src: The token index input sequence. In the Transformer architecture, it stands for the source sequence.

The input sequence is scaled by the square root of the embedding dimension after first passing through the embedding layer.

Positional information is incorporated into the embedded sequence by adding the positional encoding.

We then apply dropout regularization to the embedded sequence.

A memory tensor is produced by the Transformer encoder after processing the embedded sequence.

The input sequence and the memory tensor are used by the Transformer decoder to produce the output sequence.

Next, the output sequence is averaged over the length of the sequence.

Ultimately, the final output logits for classification are obtained by passing the averaged output via a linear layer (self.fc_out).

Training loop results for 10 epochs:

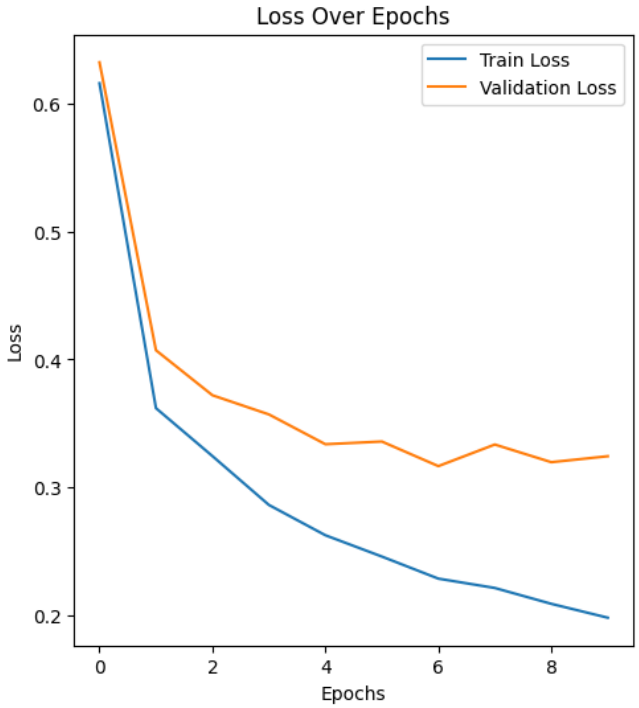
```
# Training Loop

train_losses, train_accuracies, val_losses, val_accuracies = [], [], [], []
epochs = 10
for epoch in range(epochs):
    modelArchitecture2.train()
    for texts, labels in train_loader:
        texts, labels = texts.to(device), labels.to(device)
        optimizer.zero_grad()
        output = modelArchitecture2(texts)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

    train_loss, train_acc, _, _, _ = evaluate(modelArchitecture2, train_loader, criterion, device)
    val_loss, val_acc, _, _, _ = evaluate(modelArchitecture2, val_loader, criterion, device)
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    val_losses.append(val_loss)
    val_accuracies.append(val_acc)
    print(f"Epoch {epoch + 1}: Training Loss: {train_loss:.4f}, Training Accuracy: {train_acc:.4f}, Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")
```

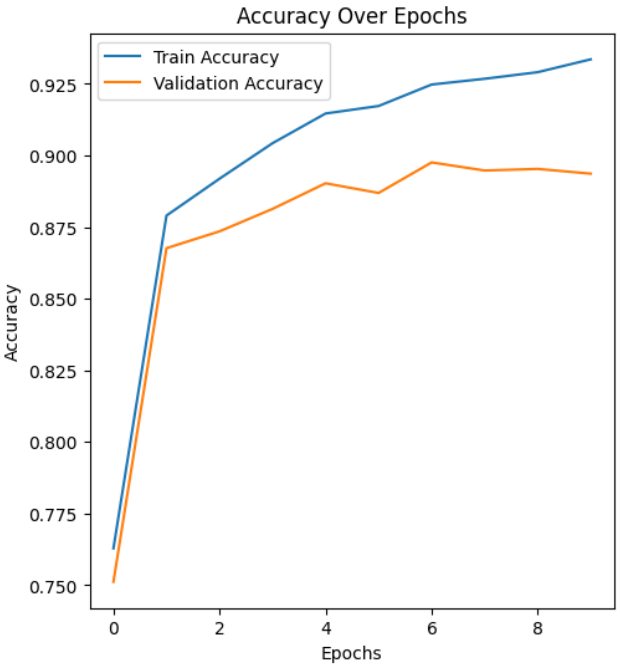
```
Epoch 1: Train Loss: 0.6159, Train Acc: 0.7629, Val Loss: 0.6322, Val Acc: 0.7512
Epoch 2: Train Loss: 0.3620, Train Acc: 0.8790, Val Loss: 0.4073, Val Acc: 0.8676
Epoch 3: Train Loss: 0.3246, Train Acc: 0.8919, Val Loss: 0.3721, Val Acc: 0.8735
Epoch 5: Train Loss: 0.2627, Train Acc: 0.9146, Val Loss: 0.3338, Val Acc: 0.8903
Epoch 6: Train Loss: 0.2461, Train Acc: 0.9173, Val Loss: 0.3360, Val Acc: 0.8869
Epoch 7: Train Loss: 0.2289, Train Acc: 0.9248, Val Loss: 0.3167, Val Acc: 0.8976
Epoch 8: Train Loss: 0.2216, Train Acc: 0.9268, Val Loss: 0.3336, Val Acc: 0.8947
Epoch 9: Train Loss: 0.2092, Train Acc: 0.9291, Val Loss: 0.3198, Val Acc: 0.8953
Epoch 10: Train Loss: 0.1983, Train Acc: 0.9336, Val Loss: 0.3245, Val Acc: 0.8937
```

Plotting testing loss and testing accuracy

Loss over epochs

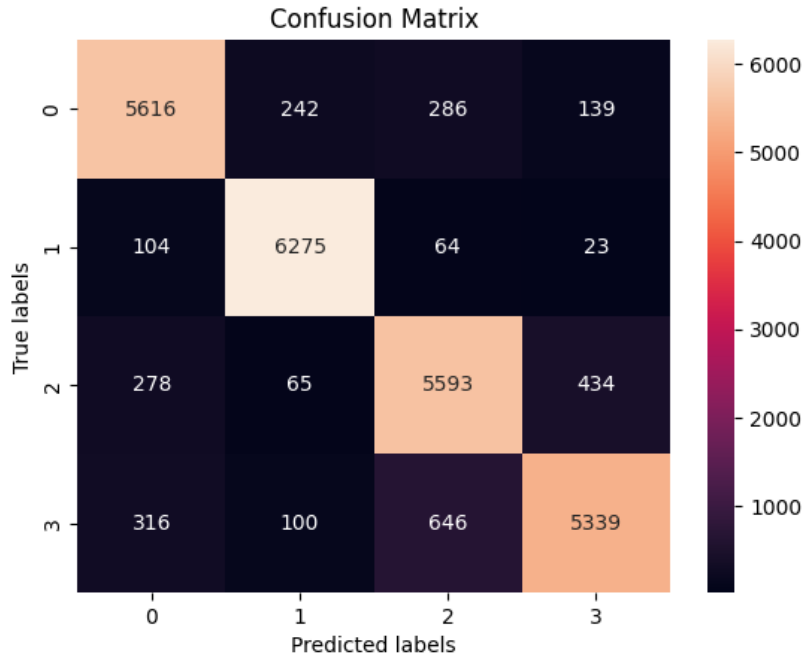
Accuracy over epochs





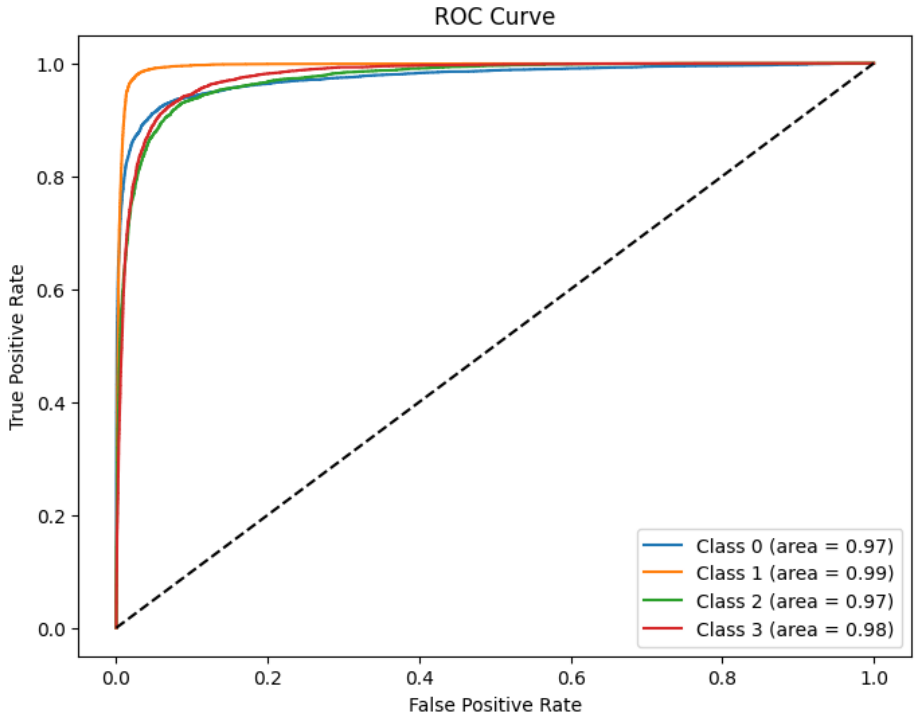

```
Testing Loss: 0.330, Testing Accuracy: 0.894
```

Confusion Matrix:




```
Precision: 0.8942, Recall: 0.8941, F1-Score: 0.8937
```

Receiver Operating Characteristic Curve:



Early Stopping:

```
#Early Stopping

modelEarlyStopping = transformerModel(vocab_size=len(vocab)+1, num_classes=4).to(device)
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/transformer.py:286: UserWarning: enable_nested_tensor is True, but self.use_nested_t
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False because {why_not_sparsity_fast_path}")
```

```
# Defining Optimizer

optimizer = torch.optim.SGD(modelEarlyStopping.parameters(), lr=0.01, momentum=0.9)
```

Training Loop:

```
# Training Loop

train_losses, train_accuracies, val_losses, val_accuracies = [], [], [], []
no_imp = 0
pat = 3
best_loss = float('inf')
epochs = 10
for epoch in range(epochs):
    modelEarlyStopping.train()
    for texts, labels in train_loader:
        texts, labels = texts.to(device), labels.to(device)
        optimizer.zero_grad()
        output = modelEarlyStopping(texts)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

    train_loss, train_acc, _, _, _ = evaluate(modelEarlyStopping, train_loader, criterion, device)
    val_loss, val_acc, _, _, _ = evaluate(modelEarlyStopping, val_loader, criterion, device)
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    val_losses.append(val_loss)
    val_accuracies.append(val_acc)
    print(f"Epoch {epoch + 1}: Training Loss: {train_loss:.4f}, Training Accuracy: {train_acc:.4f}, Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")
    if val_loss < best_loss:
        best_loss = val_loss
        no_imp = 0
    else:
        no_imp += 1
        if no_imp >= pat:
            print("Early stopping")
            break
```

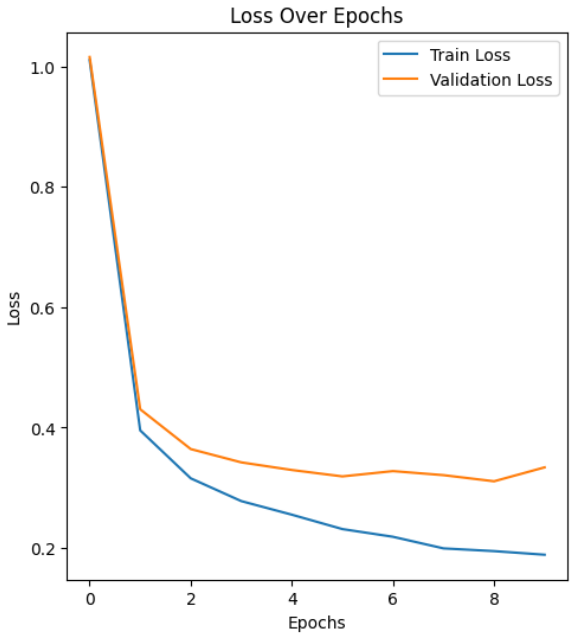
Results:

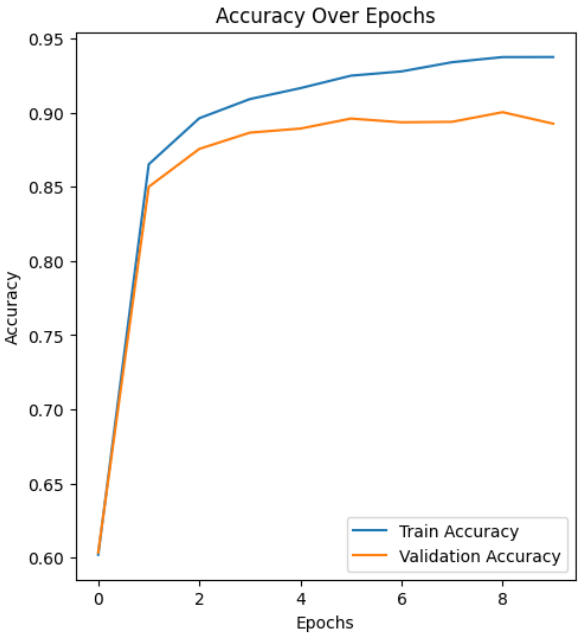
```
Epoch 1: Training Loss: 1.0107, Training Accuracy: 0.6019, Validation Loss: 1.0157, Validation Accuracy: 0.6034
Epoch 2: Training Loss: 0.3954, Training Accuracy: 0.8652, Validation Loss: 0.4305, Validation Accuracy: 0.8500
Epoch 3: Training Loss: 0.3157, Training Accuracy: 0.8962, Validation Loss: 0.3643, Validation Accuracy: 0.8755
Epoch 4: Training Loss: 0.2778, Training Accuracy: 0.9091, Validation Loss: 0.3422, Validation Accuracy: 0.8865
Epoch 5: Training Loss: 0.2553, Training Accuracy: 0.9165, Validation Loss: 0.3295, Validation Accuracy: 0.8893
Epoch 6: Training Loss: 0.2314, Training Accuracy: 0.9249, Validation Loss: 0.3189, Validation Accuracy: 0.8960
Epoch 7: Training Loss: 0.2186, Training Accuracy: 0.9278, Validation Loss: 0.3277, Validation Accuracy: 0.8935
Epoch 8: Training Loss: 0.1993, Training Accuracy: 0.9340, Validation Loss: 0.3210, Validation Accuracy: 0.8938
Epoch 9: Training Loss: 0.1947, Training Accuracy: 0.9375, Validation Loss: 0.3108, Validation Accuracy: 0.9003
Epoch 10: Training Loss: 0.1888, Training Accuracy: 0.9375, Validation Loss: 0.3339, Validation Accuracy: 0.8926
```

Plotting testing loss and testing accuracy

Loss over epochs

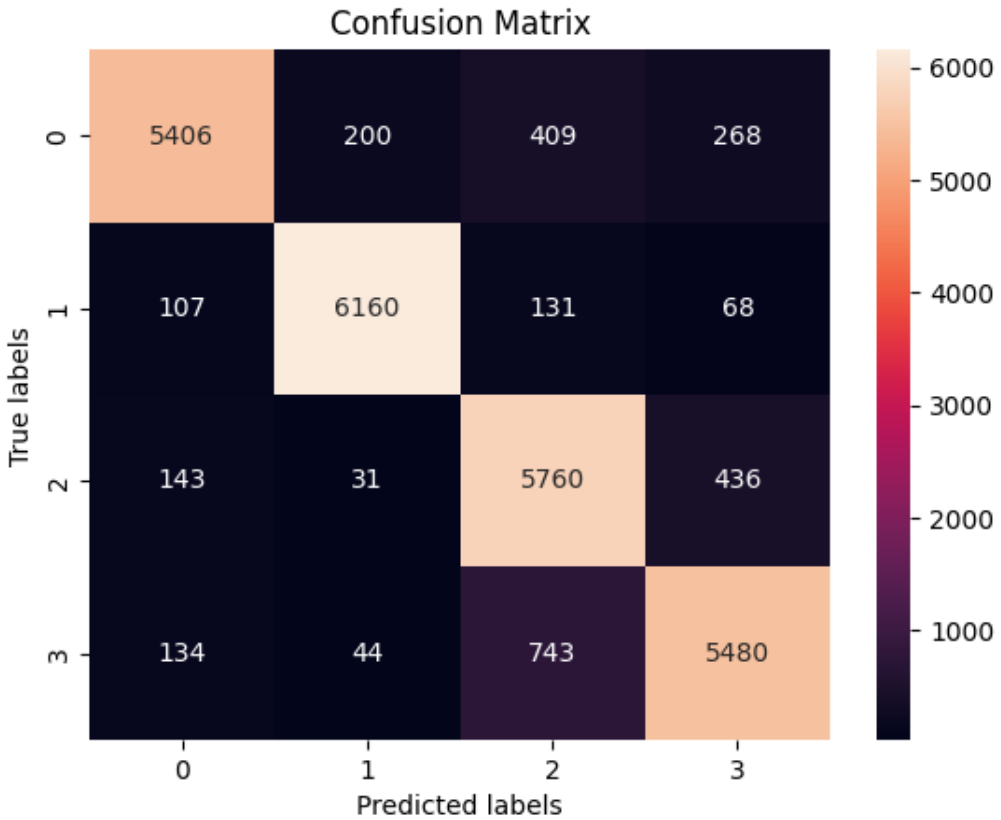
Accuracy over epochs





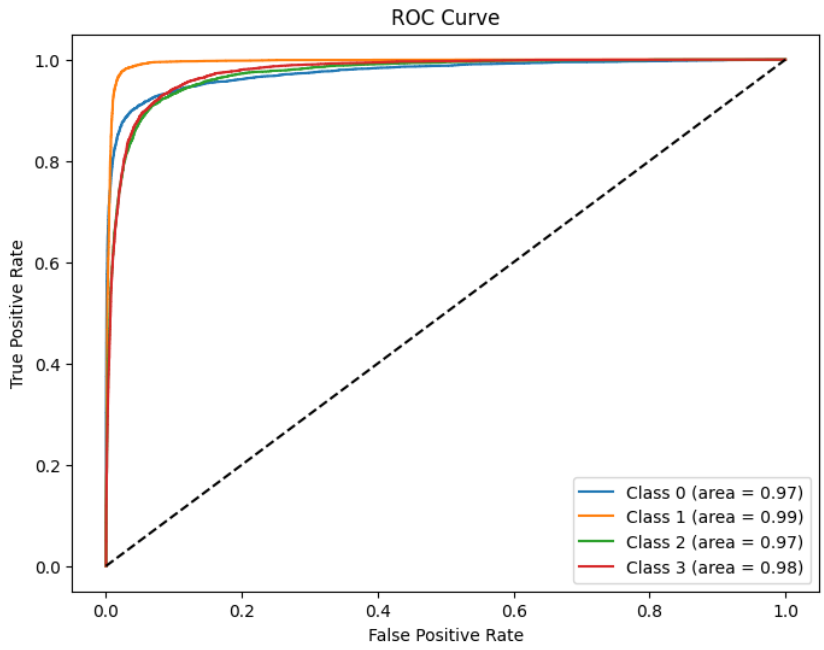

```
Test Loss: 0.3379, Test Accuracy: 0.8937
```

Confusion Matrix:




```
Precision: 0.894, Recall: 0.894, F1-Score: 0.894
```

Receiver Operating Characteristic Curve:



R2 Regularization:

```
#R2 Regularization

modelArchitecture3 = transformerModel(vocab_size=len(vocab)+1, num_classes=4).to(device)
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/transformer.py:286: UserWarning: enable_nested_tensor is True, but self.use_nested_tensor is False because encoder_layer.self_attn.batch_first was not
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False because {why_not_sparsity_fast_path}")
```

Training Loop:

```
# Training Loop
train_losses, train_accuracies, val_losses, val_accuracies = [], [], [], []
epochs = 10
for epoch in range(epochs):
    modelArchitecture3.train()
    for texts, labels in train_loader:
        texts, labels = texts.to(device), labels.to(device)
        optimizer.zero_grad()
        output = modelArchitecture3(texts)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

    train_loss, train_acc, _, _, _ = evaluate(modelArchitecture3, train_loader, criterion, device)
    val_loss, val_acc, _, _, _ = evaluate(modelArchitecture3, val_loader, criterion, device)
    train_losses.append(train_loss)
    train_accuracies.append(train_acc)
    val_losses.append(val_loss)
    val_accuracies.append(val_acc)
    print(f"Epoch {epoch + 1}: Training Loss: {train_loss:.4f}, Training Accuracy: {train_acc:.4f}, Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")
```

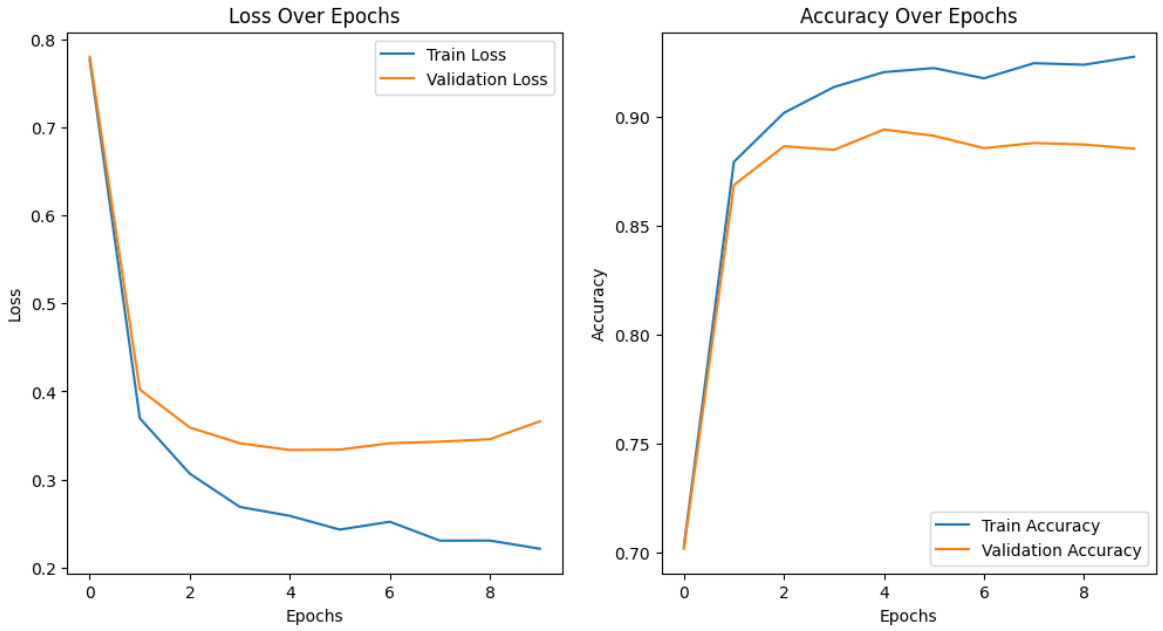
Results:

```
Epoch 1: Training Loss: 0.7768, Training Accuracy: 0.7027, Validation Loss: 0.7803, Validation Accuracy: 0.7018
Epoch 2: Training Loss: 0.3696, Training Accuracy: 0.8791, Validation Loss: 0.4021, Validation Accuracy: 0.8685
Epoch 3: Training Loss: 0.3065, Training Accuracy: 0.9016, Validation Loss: 0.3590, Validation Accuracy: 0.8863
Epoch 4: Training Loss: 0.2688, Training Accuracy: 0.9134, Validation Loss: 0.3411, Validation Accuracy: 0.8846
Epoch 5: Training Loss: 0.2586, Training Accuracy: 0.9203, Validation Loss: 0.3334, Validation Accuracy: 0.8940
Epoch 6: Training Loss: 0.2430, Training Accuracy: 0.9221, Validation Loss: 0.3339, Validation Accuracy: 0.8911
Epoch 7: Training Loss: 0.2519, Training Accuracy: 0.9174, Validation Loss: 0.3412, Validation Accuracy: 0.8854
Epoch 8: Training Loss: 0.2304, Training Accuracy: 0.9244, Validation Loss: 0.3429, Validation Accuracy: 0.8878
Epoch 9: Training Loss: 0.2305, Training Accuracy: 0.9237, Validation Loss: 0.3457, Validation Accuracy: 0.8871
Epoch 10: Training Loss: 0.2212, Training Accuracy: 0.9273, Validation Loss: 0.3660, Validation Accuracy: 0.8852
```

Plotting testing loss and testing accuracy

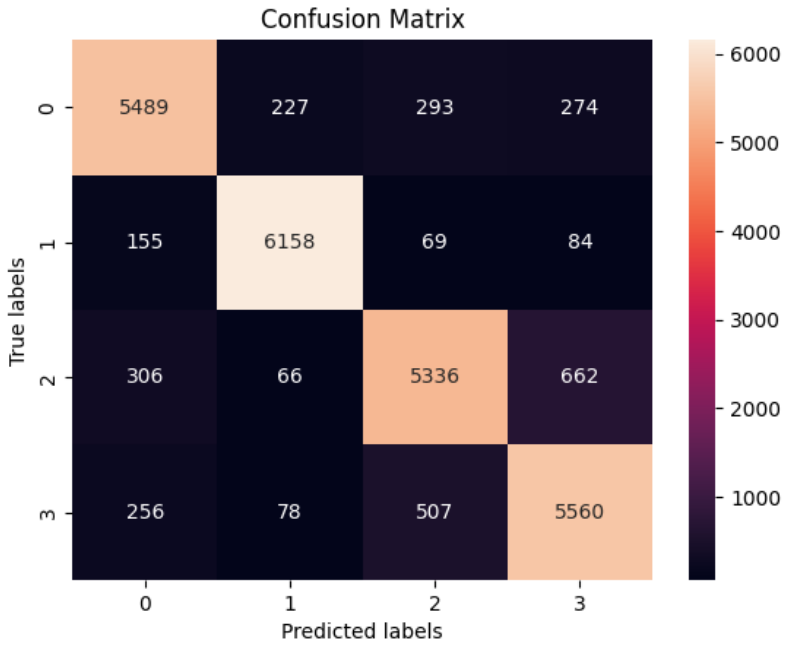
Loss over epochs

Accuracy over epochs



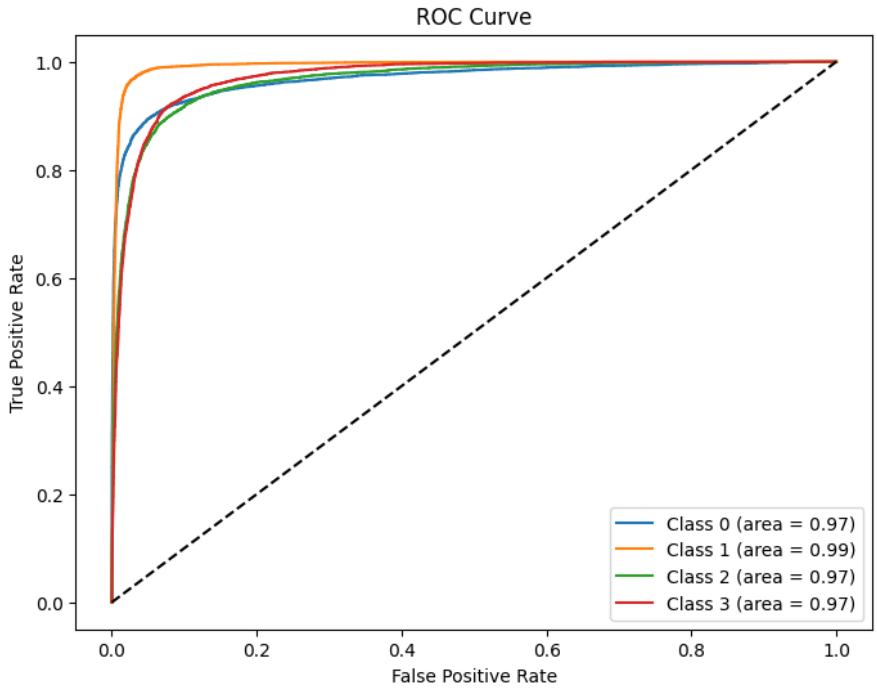

```
Final Test Loss: 0.369, Test Accuracy: 0.883
```

Confusion Matrix:




```
Precision: 0.8963, Recall: 0.8934, F1-Score: 0.8939
```

Receiver Operating Characteristic Curve:



Saving best model weights:

```
torch.save(BaseModel1.state_dict(), 'bhavesht_assignment2_part_4.h5')
```

How used techniques have impacted the model:

Dropout:

- By randomly removing (zeroing out) a certain percentage of units (or neurons) from each layer during training, the regularization approach known as "dropout" is frequently employed in neural network architectures to avoid overfitting.
- By adding noise to the representations the model learns, dropout helps prevent overfitting in Transformer models by pressuring the model to acquire more robust features.
- Dropout lessens the model's dependence on particular features or correlations in the data by arbitrarily removing units, strengthening the model's resistance to noise and enhancing its generalization capabilities.
- The learning process can be hampered by excessive dropout, so it's critical to adjust the dropout rate in accordance with the particular dataset and model architecture.

Early Stopping:

- By keeping an eye on the model's performance on a validation set during training and halting the training process when the performance begins to deteriorate, early stopping is a technique used to prevent overfitting.
- The method involves keeping an eye on a predetermined measure (such accuracy or validation loss) on the validation set and stopping training if the metric does not improve after a predetermined number of epochs (patience).
- By terminating training before the model begins to internalize noise in the training set, early stopping helps avoid the model overfitting to the data.

- To strike a balance between stopping too soon and training for an excessive amount of time, early stopping necessitates close observation of the validation metric and patience parameter tweaking.

Regularization:

Reduced Overfitting

By incorporating noise into the model's representations during training, regularization strategies like dropout aid in the prevention of overfitting.

Dropout compels the model to acquire more resilient characteristics that more closely match previously unseen data by arbitrarily removing units from the model.

Regularization makes the model less sensitive to noise and unimportant patterns in the training data, which might therefore result in better generalization performance on unknown data.

Improved Generalization:

By inhibiting the model from memorizing noise or unimportant information in the training data, regularization helps it acquire more broadly applicable representations.

As a result, the model learns to concentrate on the most important aspects of the input rather than learning every detail from the training set, which might result in better performance on unknown data.

The model's decision boundaries can be smoothed out with the use of regularization techniques like dropout, which will increase the model's robustness and reduce its sensitivity to slight changes in the input.

Balancing Bias and Variance:

Regularization approaches manage the complexity of the learnt function, which helps maintain a balance between variance and bias in the model.

Regularization encourages the model to learn simpler representations, which may slightly increase bias even as it reduces variance by preventing overfitting.

The total effect on performance is contingent upon the right trade-off between variance and bias for the given task and dataset.

References:

https://www.tensorflow.org/tutorials/generative/autoencoder

https://web.stanford.edu/class/cs25/

https://h2o.ai/wiki/transformer-architecture/

https://d2l.ai/

https://research.google/blog/transformer-a-novel-neural-network-architecture-for-language-understanding/

https://www.deeplearningbook.org/

https://www.tensorflow.org/text/tutorials/transformer