INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Introduction to NLP (CS7.401)

Assignment 4

Submitted by:

Udrasht Pal

2022201020

(MTech,CSE)

April 24, 2024

ELMo: Deep Contextualized Word Representations

ELMo (Embeddings from Language Models) stands out as a powerful model for capturing nuanced contextual meanings of words within sentences. Unlike traditional word embedding approaches, ELMo integrates bidirectional LSTM (Bi-LSTM) networks to comprehensively interpret language context. Let's delve into what ELMo is and how it operates.

How ELMo Works

ELMo's architecture revolves around two key components: forward and backward language models:

1) Bi-LSTM Structure: ELMo employs a stack of Bi-LSTM layers, allowing it to process text in both forward and backward directions simultaneously. This bidirectional approach enables ELMo to understand the relationships between words and their surrounding context comprehensively.
2) Contextualized Word Representations: ELMo's unique strength lies in its ability to generate contextualized word representations. Instead of assigning fixed embeddings to words irrespective of their context, ELMo dynamically adjusts word representations based on the entire sentence's context. This means that the same word can have different representations depending on its usage within different sentences.
3) Pretraining on Language Modeling: ELMo is pretrained on large-scale language modeling tasks, where it learns to predict the next word in a sequence (forward language model) or the previous word (backward language model). By training on these tasks, ELMo acquires a deep understanding of syntactic and semantic patterns in language.

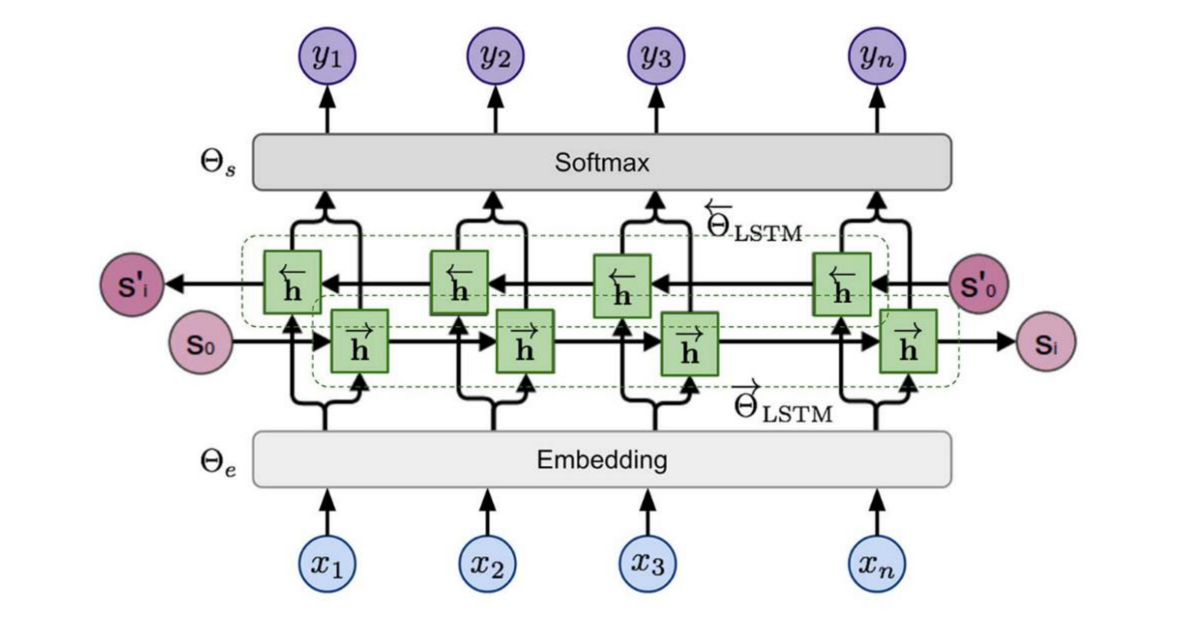


Figure 1: ELMo Architecture

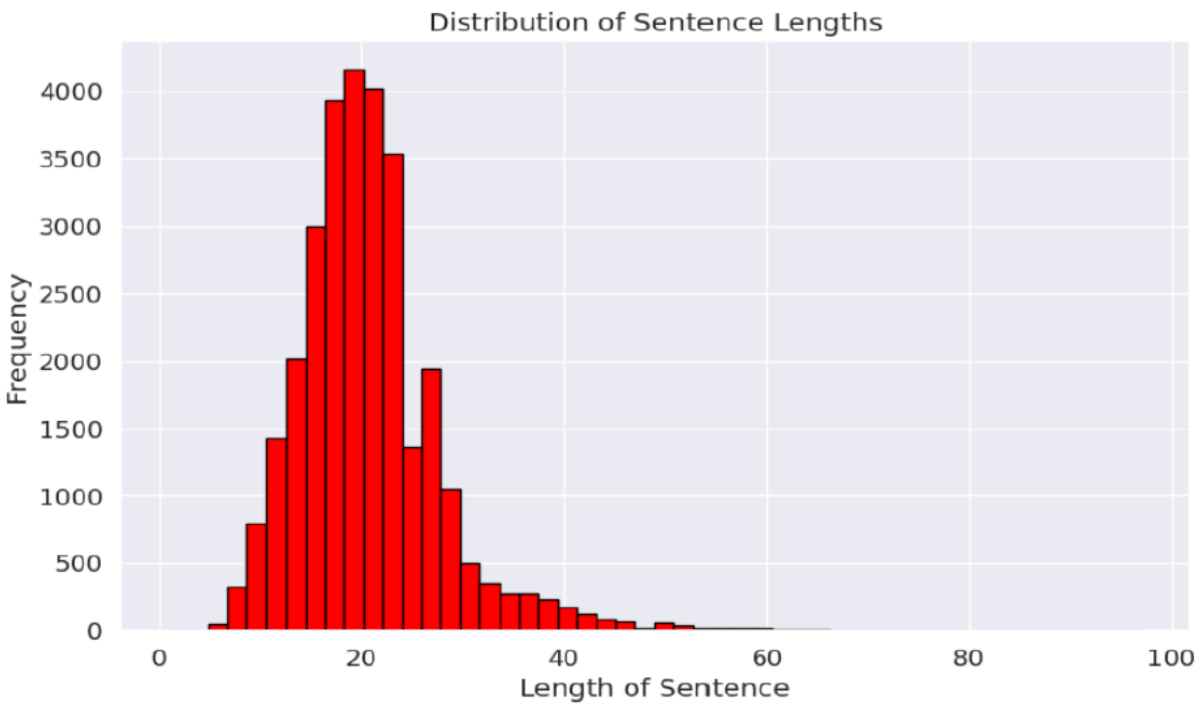
- For the Elmo embeddings, I utilized **pre-trained GLOVE** embeddings as the basis. Subsequently, I fine-tuned the Elmo model using this pre-trained embedding on the specified dataset.

Dataset Overview:

The dataset comprises news data categorized into labels 1 to 4 for training, with each label containing **30,000 sentences**. For testing, each label is represented by **1,900 sentences**. Specifically, the dataset consists of 120,000 training sentences and 7,600 testing sentences.

Elmo Training:
To train the Elmo model, I utilized **120,000 sentences**, each of which had a length of 35 words or less. An analysis of sentence lengths revealed that the majority of sentences fell within the range of 0 to 35 words.

Downstream Task:
For the downstream task, I employed **120,000 sentences**, all of which had a length of 35 words or less. During testing, I used **7,600 sentences** to evaluate the model's performance.



Hyperparameter tuning

1) Trainable λs

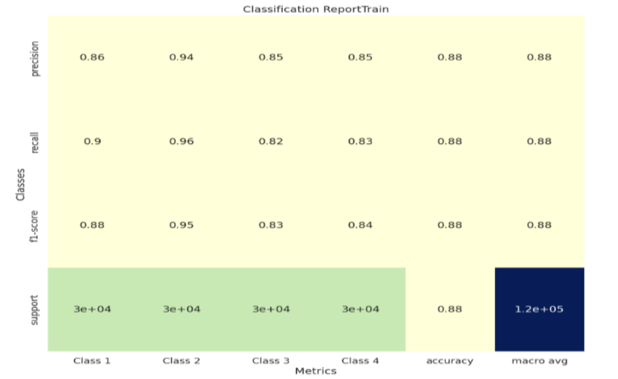
Trained λs (s1): 3.072214126586914
Trained λs (s2): 1.205386757850647
Trained λs (s3): 0.5462216734886169

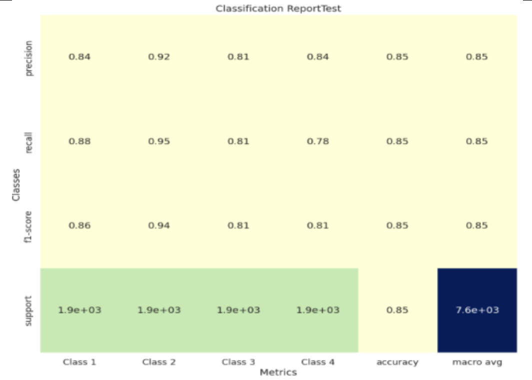
```
BATCH_SIZE=64
embedding_dim=300
learning_rate=0.001
hidden_size=50
dropout=0.1
epochs=10
```

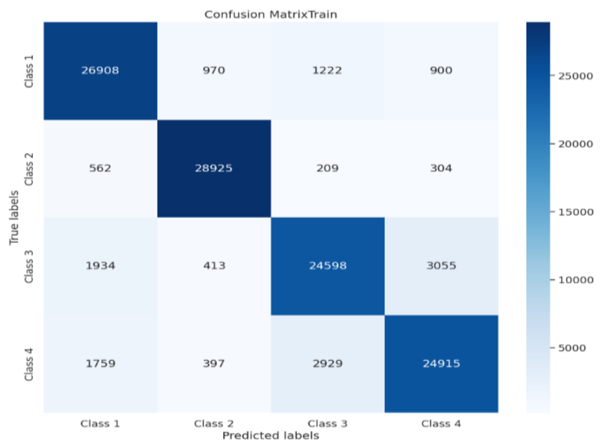
```python
class Downstream(nn.Module):
    def __init__(self, embedding_size):
        super(Downstream, self).__init__()

        self.s1 = nn.Parameter(torch.ones(1))
        self.s2 = nn.Parameter(torch.ones(1))
        self.s3 = nn.Parameter(torch.ones(1))
        self.alpha = nn.Parameter(torch.ones(1))

        # Change the output layer to 4 classes
        self.linear = nn.Linear(embedding_size, 4)  # Output layer with 4 units for 4 classes

    def forward(self, sentence):
        embeddings = elmo_bilstm.embedding(sentence)
        out_1, _ = elmo_bilstm.layer_1(embeddings)
        out_2, _ = elmo_bilstm.layer_2(out_1)


        s_sum = self.s1 + self.s2 + self.s3

        output = self.alpha * (
            self.s1 / s_sum * embeddings
            + self.s2 / s_sum * out_1
            + self.s3 / s_sum * out_2
        ).to(torch.float32)
        aggregated_output = torch.mean(output, dim=1)

        output = self.linear(aggregated_output)  # Apply linear transformation
        return output
```

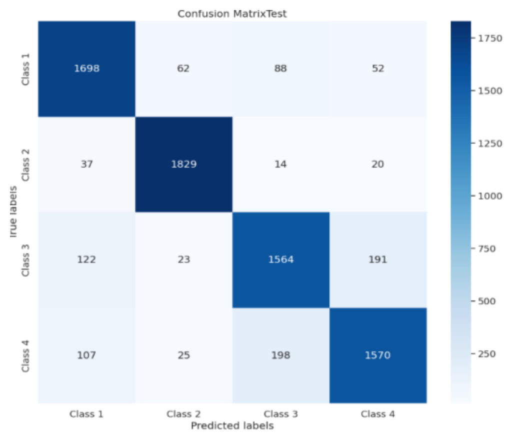
```python
downstream_model= Downstream(embedding_dim).to(DEVICE)
optimizer = optim.Adam(downstream_model.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss().to(DEVICE)
```

Trainable λs Results	
Train Accuracy: 87.3%	Test Accuracy: 88.2%









2) Frozen λs

For this task, we will randomly initialize the λs (s1, s2, s3, alpha) and freeze them, preventing further updates during training.

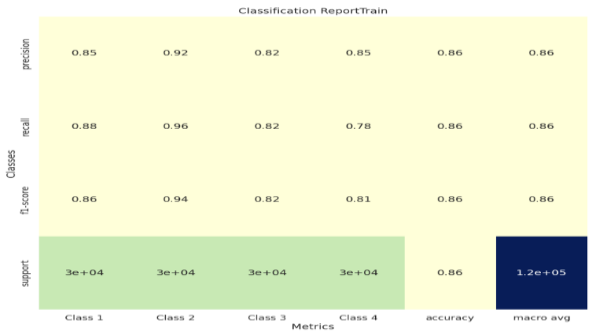
Initialize Random values of λs between 0.5 to 1

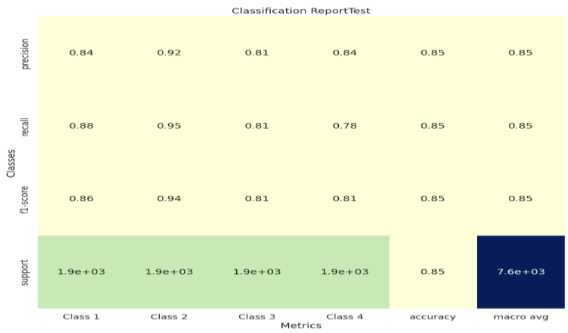
```
BATCH_SIZE=64
embedding_dim=300
learning_rate=0.001
hidden_size=50
dropout=0.1
epochs=10
```

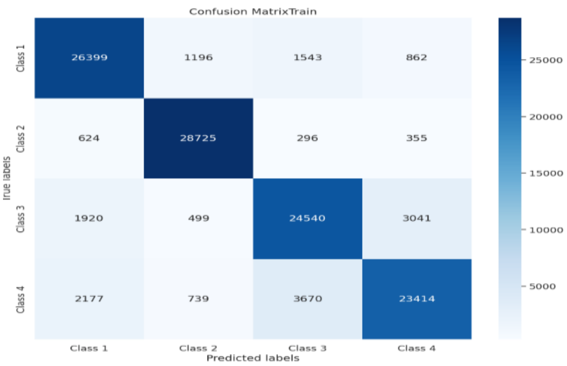
```
class DownstreamFrozenLambdas(nn.Module):
    def __init__(self, embedding_size):
        super(DownstreamFrozenLambdas, self).__init__()

        # Lambda parameters (frozen)
        self.s1 = nn.Parameter(torch.rand(1) * 0.5 + 0.5, requires_grad=False)  # Random value between 0.5 and 1
        self.s2 = nn.Parameter(torch.rand(1) * 0.5 + 0.5, requires_grad=False)  # Random value between 0.5 and 1
        self.s3 = nn.Parameter(torch.rand(1) * 0.5 + 0.5, requires_grad=False)  # Random value between 0.5 and 1
        self.alpha = nn.Parameter(torch.rand(1) * 0.5 + 0.5, requires_grad=False)  # Random value between 0.5 and 1

        # Output layer
        self.linear = nn.Linear(embedding_size, 4)  # Output layer with 4 units for 4 classes

    def forward(self, sentence):
        # Similar forward pass as above
        embeddings = elmo_bilstm.embedding(sentence)
        out_1, _ = elmo_bilstm.layer_1(embeddings)
        out_2, _ = elmo_bilstm.layer_2(out_1)

        # Combine word representations using frozen lambda weights
        s_sum = self.s1 + self.s2 + self.s3
        output = self.alpha * (
            self.s1 / s_sum * embeddings +
            self.s2 / s_sum * out_1 +
            self.s3 / s_sum * out_2
        ).to(torch.float32)

        # Aggregate output
        aggregated_output = torch.mean(output, dim=1)

        # Apply linear transformation
        output = self.linear(aggregated_output)

        return output
```

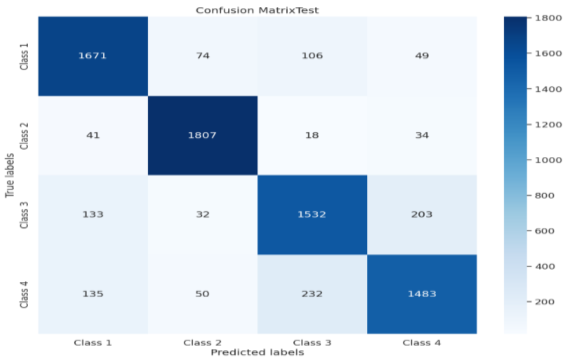
```
downstream_frozen_model  = DownstreamFrozenLambdas(embedding_dim).to(DEVICE)
criterion = nn.CrossEntropyLoss().to(DEVICE)  # Use CrossEntropyLoss for multi-class classification
# Define optimizer and criterion
optimizer_frozen  = optim.Adam(filter(lambda p: p.requires_grad, downstream_frozen_model .parameters()), lr=learning_rate)
```

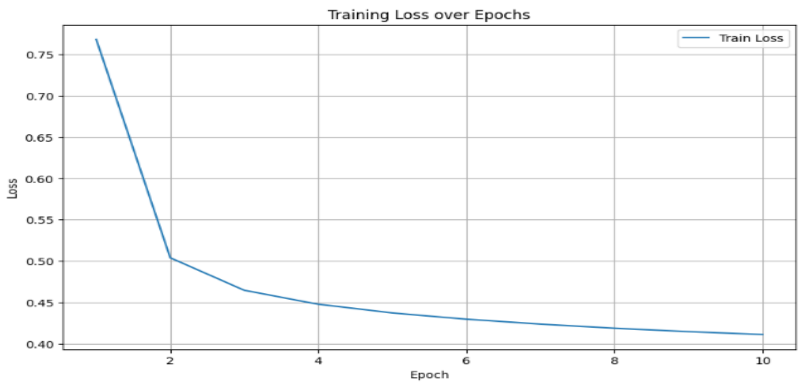
Frozen λs Results	
Train	Test
Train Accuracy: 86.3%	Test Accuracy: 85.2%











3 Learnable Function

A neural network-based function f is employed to combine word representations $e0$,$e1$,$e2$ from different layers. This function f incorporates trainable parameters to adaptively aggregate the input embeddings into a final contextual word embedding E. Through training f with labeled data using techniques such as backpropagation, the goal is to optimize E for downstream tasks like classification, leveraging its ability to capture nuanced linguistic contexts.

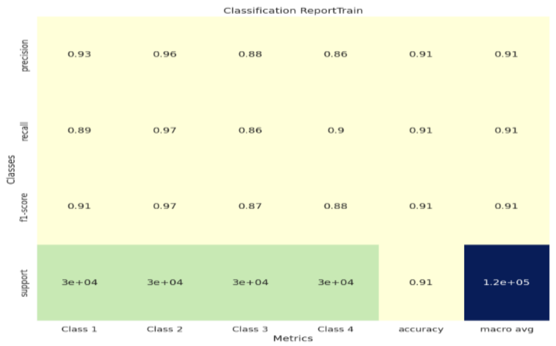
```
BATCH_SIZE=64
embedding_dim=300
learning_rate=0.001
hidden_size=50
dropout=0.1
epochs=10
```

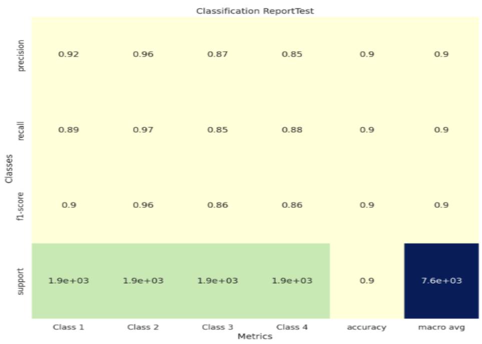
```python
class LearnableFunction(nn.Module):
    def __init__(self, input_size, output_size):
        super(LearnableFunction, self).__init__()
        self.fc1 = nn.Linear(input_size, output_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(output_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

class DownstreamLearnable(nn.Module):
    def __init__(self, embedding_size):
        super(Downstream, self).__init__()

        # Define the learnable function to combine word representations
        self.learnable_function = LearnableFunction(3 * embedding_size, embedding_size)

        # Other components of your model
        self.embedding_size = embedding_size
        self.linear = nn.Linear(embedding_size, 4)  # Output layer with 4 units for classification

    def forward(self, sentence):
        # Implement your forward pass using the learnable function and other components
        embeddings = elmo_bilstm.embedding(sentence)
        out_1, _ = elmo_bilstm.layer_1(embeddings)
        out_2, _ = elmo_bilstm.layer_2(out_1)

        # Combine word representations using the learnable function
        combined_representation = torch.cat([embeddings, out_1, out_2], dim=2).to(torch.float32)
        learned_embedding = self.learnable_function(combined_representation)
        learned_embedding=torch.mean(learned_embedding, dim=1)

        # Apply linear transformation
        output = self.linear(learned_embedding)
        return output
```

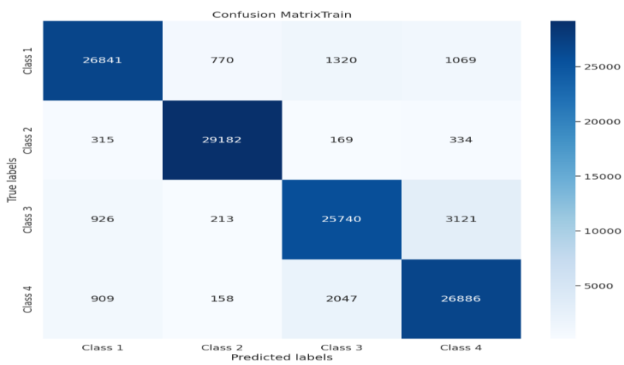
```python
downstream_model_Learnable = DownstreamLearnable(embedding_dim).to(DEVICE)
optimizer_Learnable = optim.Adam(downstream_model_Learnable.parameters(), lr=learning_rate)
criterion = nn.CrossEntropyLoss().to(DEVICE)
```

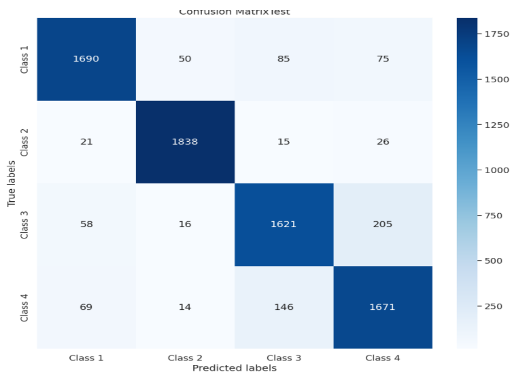
Learnable Function Results

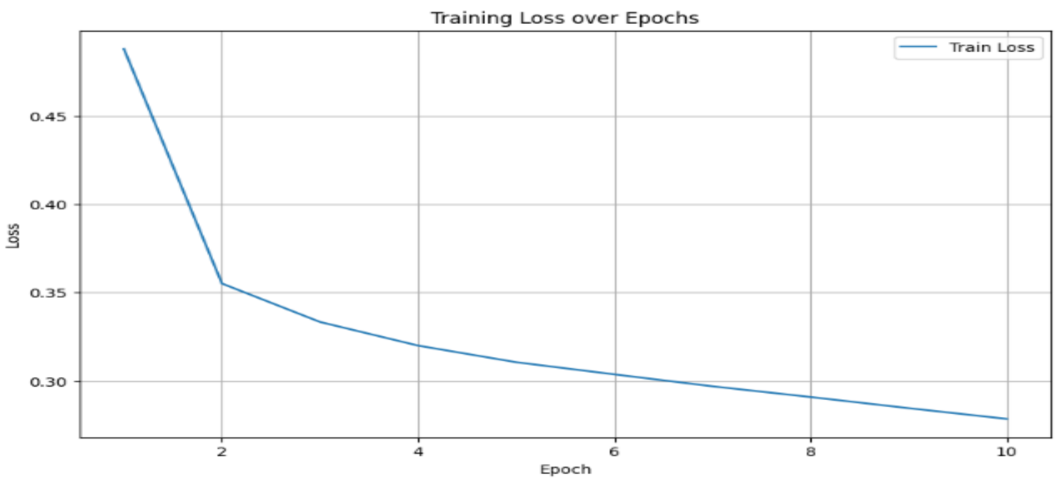
Train	Test
Train Accuracy: 91.1%	Test Accuracy: 90.01%











Analysis

After applying Singular Value Decomposition (SVD), Skip-gram, and various versions of ELMo to generate word embeddings on the provided dataset, I obtained the following results when utilizing these embeddings for downstream classification tasks:

Embedding Model	Train Accuracy	Test Accuracy	Embedding size
SVD	85.9%	83.19%	300
skip-gram	91%	85%	300
Elmo	85%	87.3%	300
Elmo with Frozen λs	85.2%	86.3%	300
Elmo with Learnable Function	**91%**	**90%**	**300**

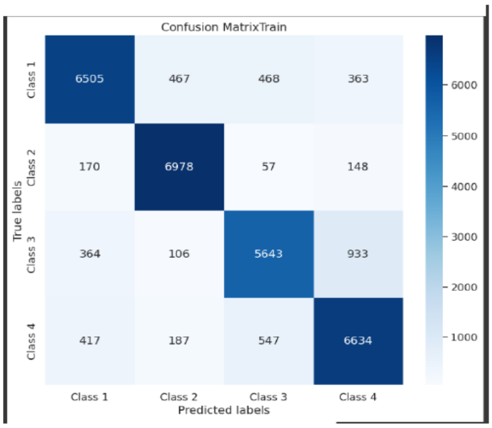
Downstream Task test and train Accuracy

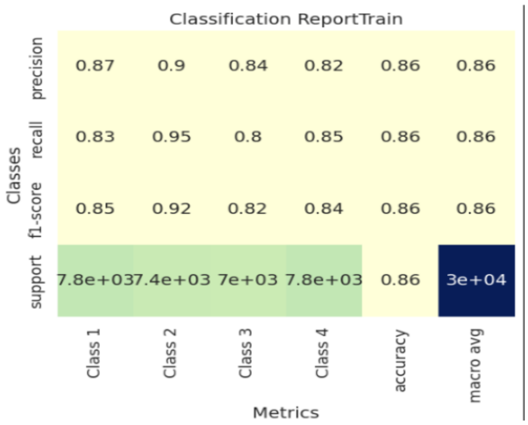
The accuracy values presented in the table represent the highest achieved after extensive hyperparameter tuning for each respective model.

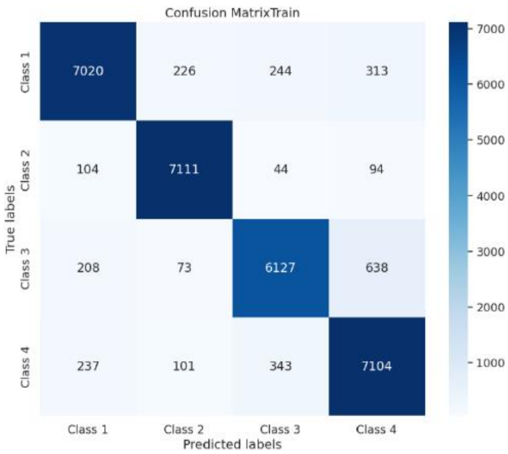
When I created an ELMo embedding with an **embedding size of 100**, the test accuracy was around 85%. However, when I increased the **embedding size to 300**, it showed improved performance.

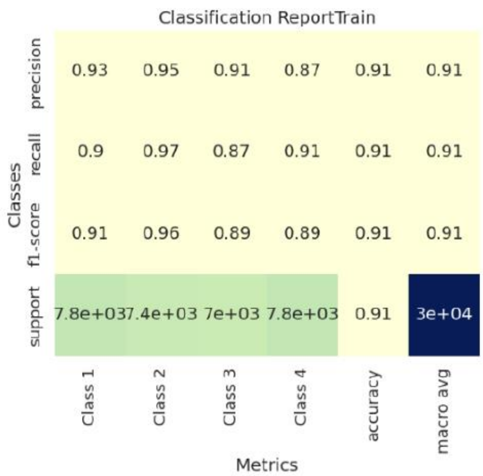
The decision to use a 300-dimensional embedding size resulted in better performance for reasons:

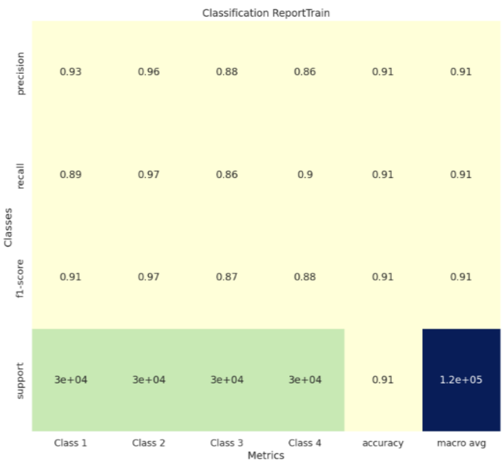
Increasing the embedding size to 300 dimensions enhances semantic richness, contextual understanding, reduces information loss, and improves generalization for ELMo embeddings, resulting in improved test accuracy.

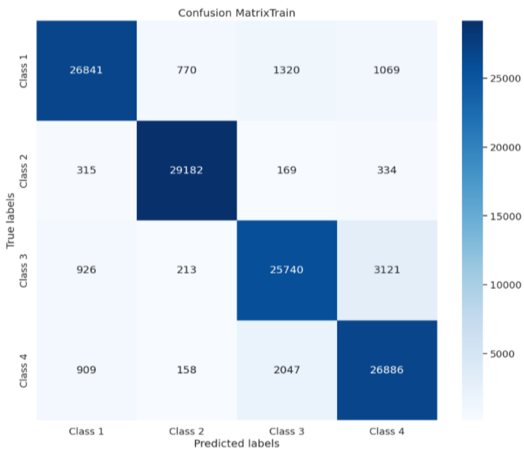
<table>
<tr><th>Embedding Model</th><th>Downstream Task Train Matrix</th></tr>
<tr><td>SVD Embedding</td><td>




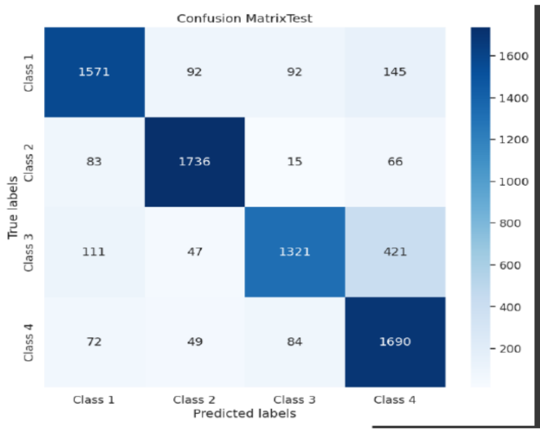
</td></tr>
<tr><td>Skip-gram Embedding</td><td>


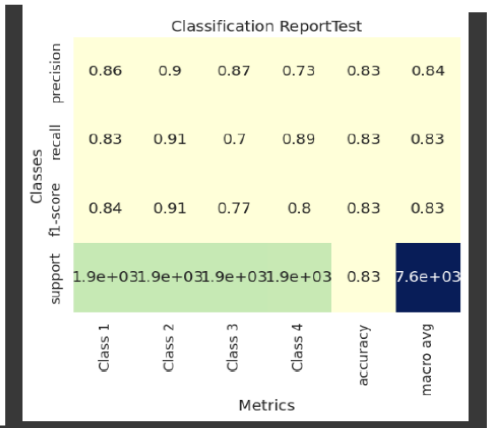


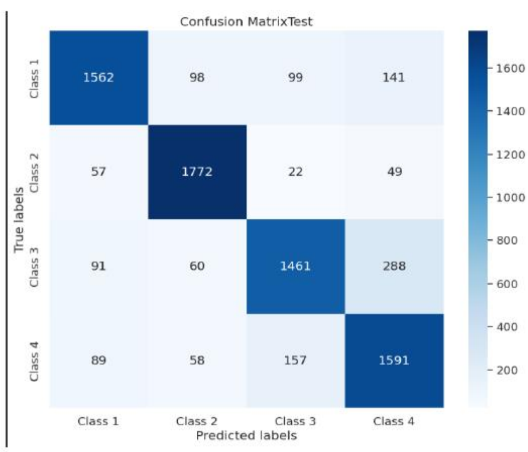
</td></tr>
<tr><td>Elmo</td><td>


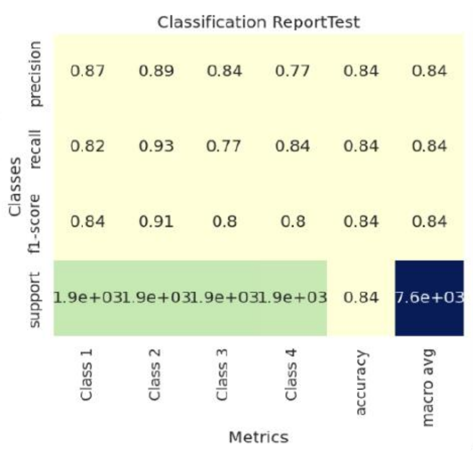


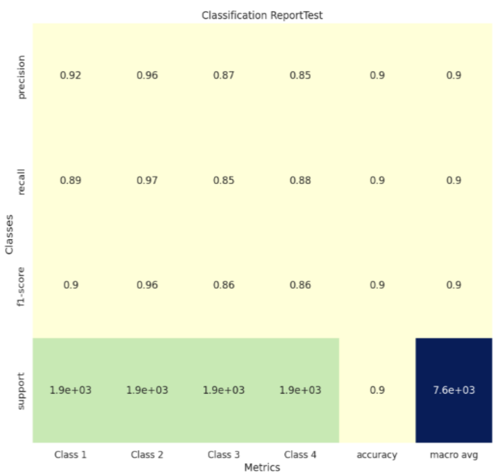
</td></tr>
</table>

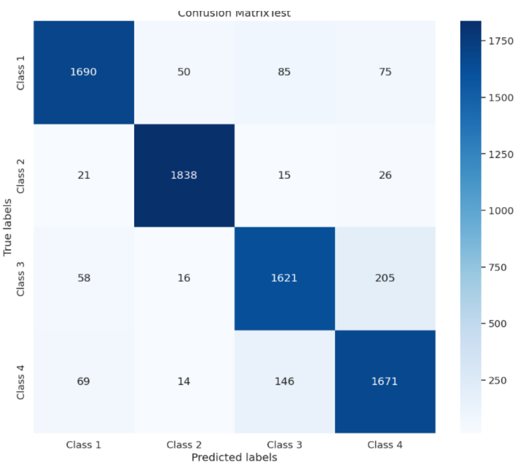
<table>
<tr><th>Embedding Model</th><th>Downstream Task Test Matrix</th></tr>
<tr><td>SVD Embedding</td><td>




</td></tr>
<tr><td>Skip-gram Embedding</td><td>




</td></tr>
<tr><td>Elmo</td><td>




</td></tr>
</table>

In my observations, I found that using the **ELMo embedding consistently yielded higher accuracy compared to both SVD and Skip-gram.**

The ELMo embedding model with a learnable function achieved the highest accuracy in the downstream task, with a test accuracy of 90% and a train accuracy of 91%. This indicates that the ELMo model with a learnable function outperformed other embedding methods in terms of both generalization (test accuracy) and training performance

Elmo is better then SVD, Skip-gram:

- **Contextualized Representations:** ELMo captures nuanced word meanings based on their context within sentences, offering more detailed representations compared to static embeddings.
- **Deep Bidirectional Modeling:** ELMo's deep Bidirectional LSTM architecture comprehensively captures word relationships in both directions, enhancing its ability to understand complex linguistic patterns.
- **Subword Information Handling:** ELMo's consideration of subword information allows it to handle rare or unseen words effectively, unlike SVD and Skip-gram which operate at the word level.
- **Transfer Learning Advantage:** ELMo embeddings can be fine-tuned for specific tasks using pre-trained language representations, leading to improved performance on new datasets.
- **Higher Dimensionality for Richer Semantics:** ELMo embeddings have higher dimensionality than SVD and Skip-gram, enabling them to capture richer semantic information and language nuances.
- **Proven State-of-the-Art Performance:** ELMo has consistently demonstrated state-of-the-art results across various language tasks, highlighting its effectiveness in natural language understanding and processing.

ELMo with a learnable function outperforms other methods in downstream tasks due to several key factors:

- Flexible Learning Patterns: The learnable function dynamically adapts to specific task patterns and relationships, improving performance by learning complex interactions among word representations.
- Task-Specific Optimization: This function optimizes how word representations are combined, customizing integration strategies to the unique requirements of each downstream task. This customization leads to enhanced accuracy.
- Hypermeter Tuning: The learnable function acts as a hyperparameter that can be fine-tuned to optimize the model's performance for specific tasks. This tuning allows for flexibility and adaptability, contributing to improved task performance.
- Capturing Non-Linear Relationships: ELMo's learnable function captures non-linear relationships between word representations, which is crucial for understanding intricate linguistic structures and achieving higher accuracy in downstream tasks.