Document Classification Using BERT

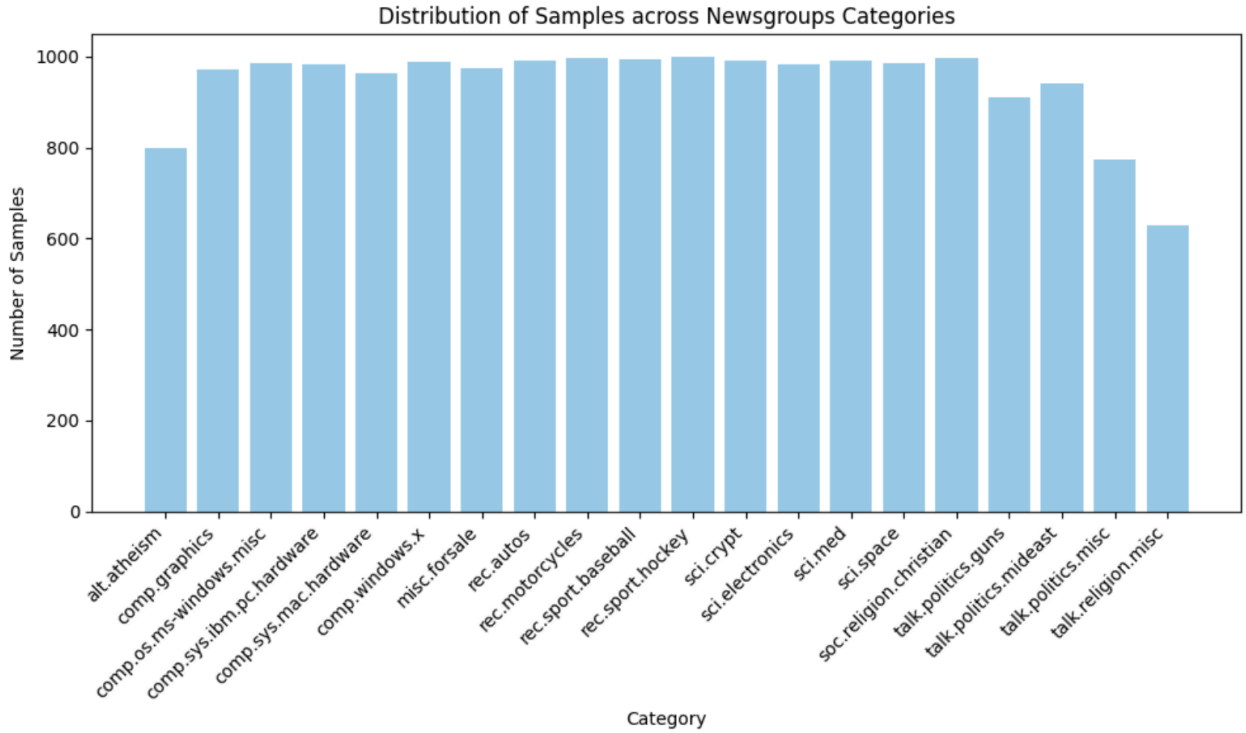
Natural Language Processing: Assignment 3

By:

1) Saket Nerurkar(2021A7PS0001G)
2) Samarth Nagpal (2022A7PS0475G)
3) Vagarth Dvivedi (2021A7PS2426G)

Exploratory Data Analysis

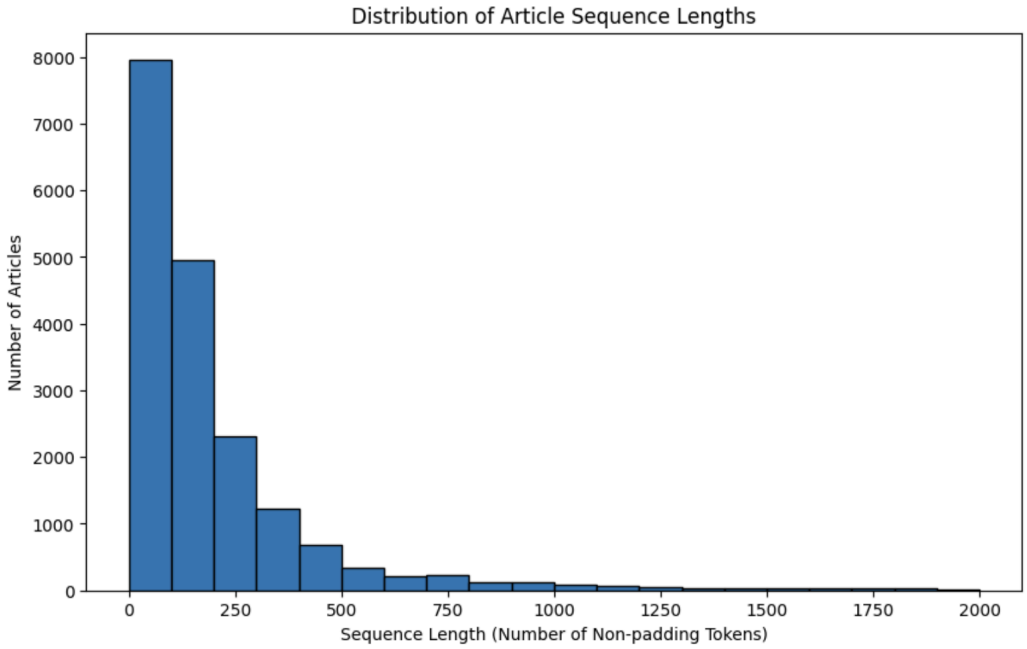
Documents per class



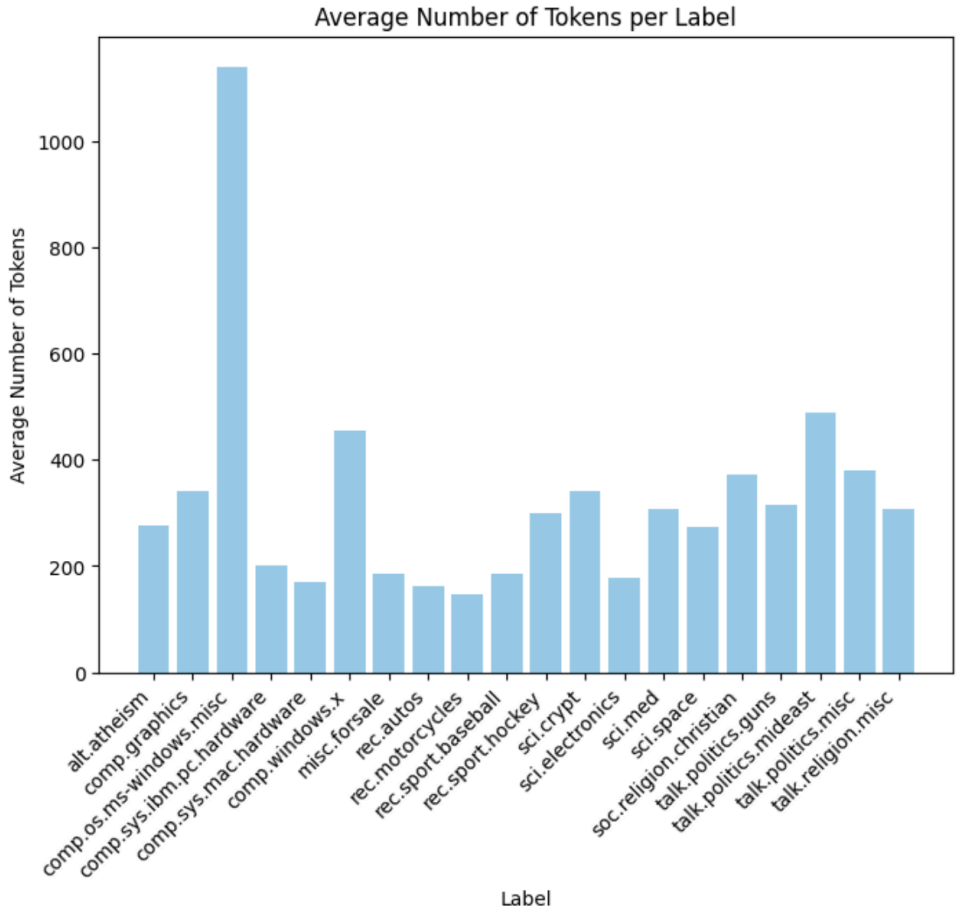
Tokens per document

Dataset Tokenization Statistics

- **Total Articles:** 18,846
- **Articles with Token Count > 512:** 1,654
- **Articles with Token Count > 1,024:** 657
- **Articles with Token Count > 2,048:** 282
- **Articles with Token Count > 4,096:** 135



Average tokens per class



Most frequently occurring words per class

These are the most frequent words in each category of the dataset. During training, the model learns to assign varying levels of importance to different words. For instance, when encountering words like 'atheists' or 'God,' the model may classify the document as related to atheism, while it may give less weight to commonly used terms like 'edu,' which appear across multiple categories.

Category	Top Word 5	Top Word 4	Top Word 3	Top Word 2	Top Word 1
alt.atheism	livesey	atheists	keith	god	edu
comp.graphics	files	3d	image	edu	graphics
comp.os.ms-windows.misc	microsoft	file	dos	edu	windows
comp.sys.ibm.pc.hardware	edu	card	ide	drive	scsi
comp.sys.mac.hardware	drive	monitor	apple	edu	mac
comp.windows.x	Edu	mit	server	motif	window
misc.forsale	shipping	offer	0	edu	sale
rec.autos	Article	engine	cars	edu	car
rec.motorcycles	Ride	ca	edu	dod	bike
rec.sport.baseball	year	team	game	baseball	edu
rec.sport.hockey	edu	ca	team	hockey	game
sci.crypt	government	chip	encryption	key	clipper
sci.electronics	organization	subject	lines	use	edu
sci.med	gordon	geb	msg	pitt	edu
sci.space	access	henry	nasa	edu	space
soc.religion.christian	christian	edu	church	jesus	god
talk.politics.guns	guns	stratus	fbi	edu	gun
talk.politics.mideast	armenian	edu	jews	israeli	israel
talk.politics.misc	Clayton	people	optilink	edu	cramer
talk.religion.misc	christian	edu	jesus	sandvik	god

Cleaning the data

Data Cleaning techniques:

1. Removing excessive whitespaces: While BERT's tokenizer is designed to handle spaces effectively, excessive whitespace—such as extra spaces, multiple newlines, or inconsistent formatting—can lead to unnecessary padding. This padding often inflates the token count, resulting in inefficient utilization of the input sequence length (e.g., sentences or paragraphs separated by multiple spaces or newlines). Addressing these issues can help optimize tokenization and reduce the need for splitting long documents into chunks.
2. Removing header footer quotes : These might be irrelevant to the class but might be common across various articles in the class. This might result in the model overfitting to these values and not generalizing properly.
3. Ignoring articles <30 tokens in length: Small articles might not have relevant information to help classify them. We ignore such documents.

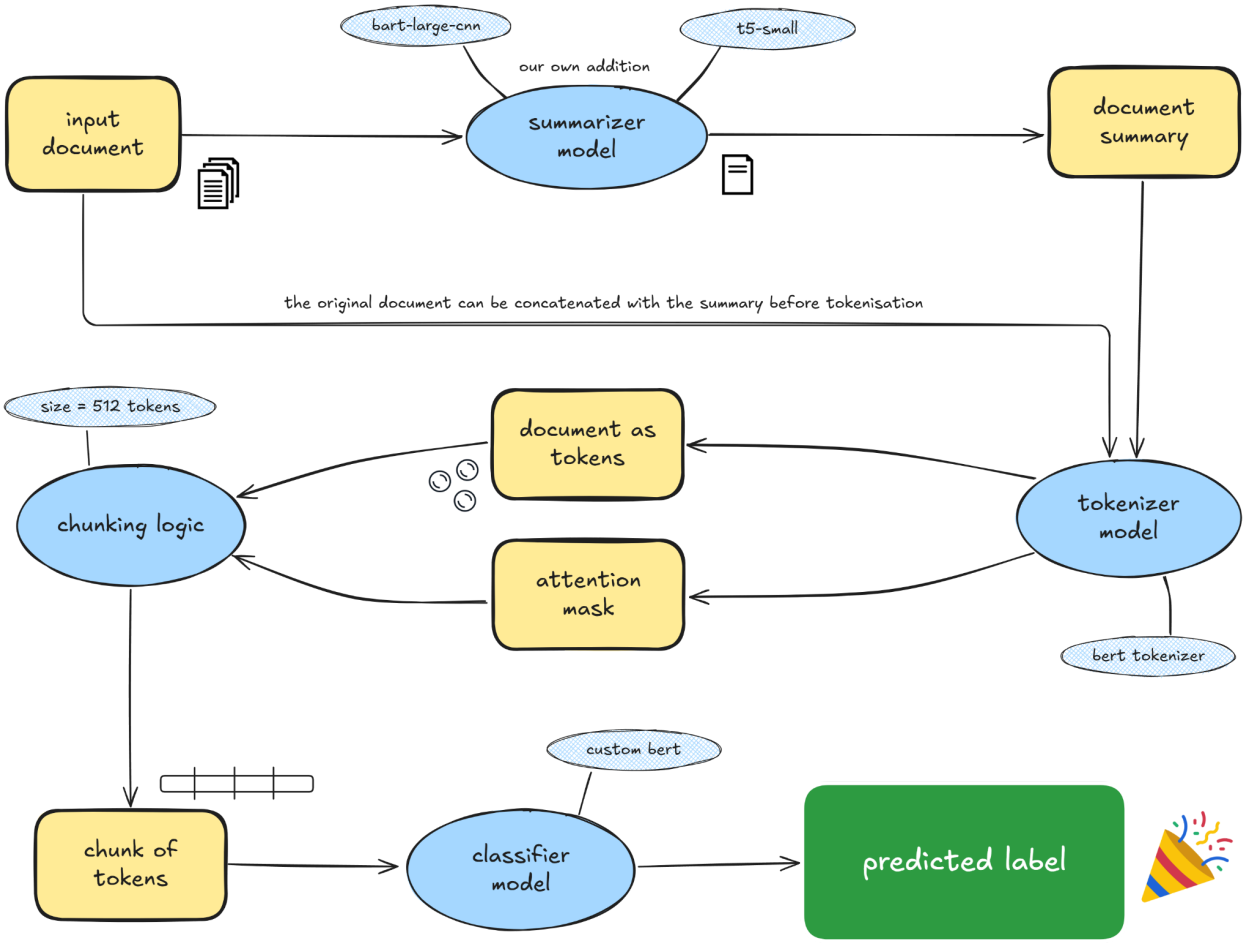
```
>say they have a "history of untrustworthy behavoir[sic]"?

Original class is  sci.crypt
```

4. Normalize Case : As we are using bert-base-uncased the tokenizer automatically converts the words into lowercase.
5. Stop word removal : This is not recommended when using bert as stopwords help the model capture context. The full context helps create accurate embeddings and hence improves results.

Data Pipeline

Here is a visualization of the data pipeline of this project at the point of evaluation. This pipeline gives a top-down view of the steps that are undertaken when a new document is being processed to produce a predicted label.



Handling Long Documents

Since BERT has an maximum input size limit (defaulted to 512 tokens) we must figure out a way to pass documents whose length exceeds 512 tokens to BERT. Our test dataset contained 1649 documents that exceeded the input length. We will now refer to these documents that exceed this token length as *long documents*. We tested three different approaches to handling long documents - tossing, chunking, and summarization.

The Token Tossing Technique

The most basic approach to handling long documents is to take the first 512 tokens and toss out the rest as if they did not exist at all.

Advantages:

- Faster to run
- Easier to verify and debug

Disadvantages:

- Ignores possibly import parts of the article that can help in classification.
- Reduces the size of training data.

Using this as an initial method we were able to achieve an f1 score of 0.72.

Chunking

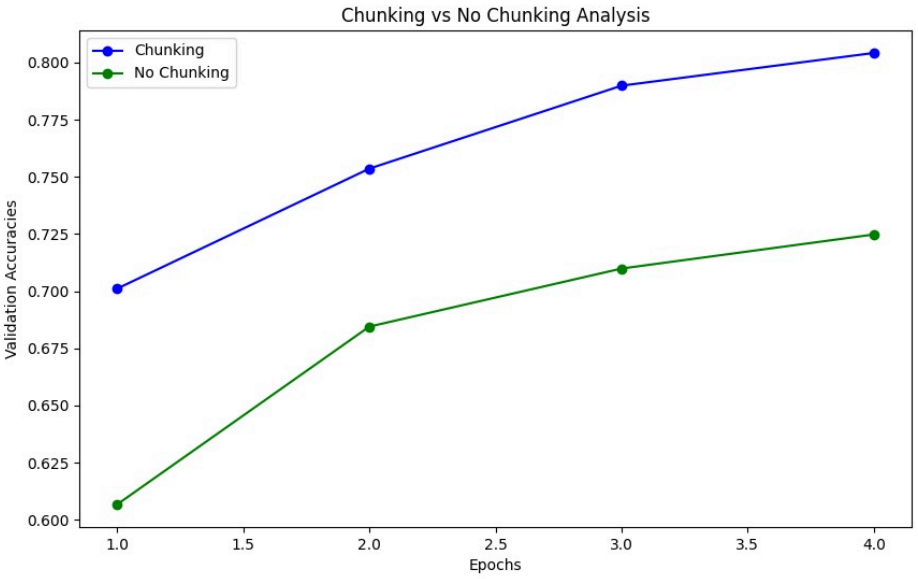
The bert model is limited to a max tokens of 512, to accommodate articles which have total tokens of greater than 512 we used chunking. In essence, all chunking approaches involve a sliding window across the series of tokens which gets passed into the model sequentially as chunks. However, there is some diversity of approaches even among how chunking can be done, we tried one of them. After splitting a document into chunks, we consider each chunk as an individual article of the same class as the original article and train the model on this newly created dataset. This helps create a larger dataset than the token tossing technique. When we run the model against articles in real life the model will run against each chunk and classify them separately, we will add the probabilities of each class for all the chunks and take the class with the highest probability as the final label.

Various chunking policies

We researched on 3 different chunking policy and tested one of them::

1. **Chunking Based on Token Number:**
    - The simplest chunking policy is to simply have a fixed-size sliding window over our document and continually pick the next 512 tokens as an input chunk until the end of the document is reached.

- All chunks except the last must have the same number of tokens in them.
- In our implementation, we skip the last chunk if it doesn't have enough tokens, this improved performance marginally because it removed very small and anomalous chunks from being trained upon.
- Problematically, this could mean that two chunks of the same document lack context between each other and that meaningful sentences might get sliced down the middle.
- This is the most computationally efficient form of chunking and we ended up picking this one for that reason alone because we observed all chunking policies having roughly similar accuracy in the end.
- We implemented this chunking algorithm and were able to get significantly better results than the no chunking logic with a final f1 score of 0.81 compared to the 0.72 score we got from the no chunking logic.



2. **End of Sentence Chunking:**
   - Instead of cutting off the chunk at 512 tokens precisely, this policy involves creating the largest chunks while also ending at the end of a sentence.
   - This chunking policy should be able to fix the problem of having parts of sentences in the start and end of the chunks.

3. **Overlapped Chunking:**
   - This approach involves having the ending tokens of one chunk be present as the first tokens of the second chunk, effectively creating an overlapping set of tokens between any two adjacent chunks.
   - This should help preserve context between chunks
   - This does mean that we will be passing some tokens twice, and thereby increasing the computational overhead of the task. This also might lead to overfitting.

Changing chunk sizes

We also experimented with changing chunk sizes. We tested on 512, 300 and 100. We found that the model with `chunk_size=512` had an F1 score of 81.567%, the model with `chunk_size=300` had an F1 score of 81.570% and the model trained on `chunk_size=100` had an F1 score of 77.843%.

Analysis:

- Most documents in the dataset are not excessively long, so reducing chunk size to 300 doesn't lose much context.
- Further reducing chunk size to 100 results in a lot more articles getting chunked and the smaller context size makes it harder to get useful context resulting in a more significant drop in score.
- As Bert is not able to make logical connections between chunks the model is impacted by breaking an article into more chunks.

Comparing de-chunking policies

After chunking a document into smaller parts due to token limitations, de-chunking is required to combine the predictions from each chunk into a final output. Since chunks are processed independently, we need a method to merge these predictions in a way that accurately reflects the entire document's content, ensuring meaningful results.

There are various ways to combine predictions from different chunks:

- **Majority Selection**: This method assigns the final label based on the majority vote across all chunks. Each chunk's predicted label is considered, and the label with the most occurrences across chunks is chosen as the final document label.
- **Aggregate (Our Approach)**: In our approach, we sum the class predictions across all chunks, and then select the class with the highest sum. This approach allows us to consider the overall context of the document by accumulating the predictions of all chunks, rather than making a decision based solely on individual chunk labels.
- **Weighted Sum**: In this method, each chunk's prediction is weighted based on factors like the number of tokens in that chunk. The final prediction is then based on the sum of these weighted predictions, allowing for more influence from longer chunks.

The Summarization Approach

This approach involves explicitly using another external model to generate a text summary of the document and then feeding that summary into our data pipeline. We can either feed in the summary by itself or feed it by concatenating it to the rest of the document. This was our own implementation outside of the scope of the assignment and is discussed further in the section titled "Extra: Document Summarization" towards the end of this report.

Comparing Pooling Techniques

What is Pooling?

Pooling is a technique for condensing or summarizing data from a series of embeddings (word or token representations) into a fixed-size vector (this is usually a 768 length vector in bert). This vector represents input and is used for the task of categorization which is being performed here. We need pooling for BERT specifically because it generates an embedding for each token in the input but we want a single vector to represent the meaning of the entire sequence of input tokens. Although this means that we lose some information, it also means that we are able to reduce computational complexity and significantly speed up the entire labeling process.

Implementing Different Pooling Techniques

CLS Pooling

For every input to BERT, we have a special `[CLS]` token at the starting. The primary purpose of the `[CLS]` token is to capture a representation of the entire input sequence, which is then used for tasks like classification. In CLS-pooling we take the embedding which was assigned to the `[CLS]` token as it is representative of the entire input sequence. This works for BERT because it is already pre-trained to give a *summarized* embedding of the input to the `[CLS]` token.

Mean Pooling

This involves taking the average of all the embeddings for every token present in the input text instead of selecting any specific token. This means that the embeddings of each token are equally valued in this mean representation. Theoretically, this should help us capture information from longer documents that have more varied tokens because each token has some contribution to the overall mean.

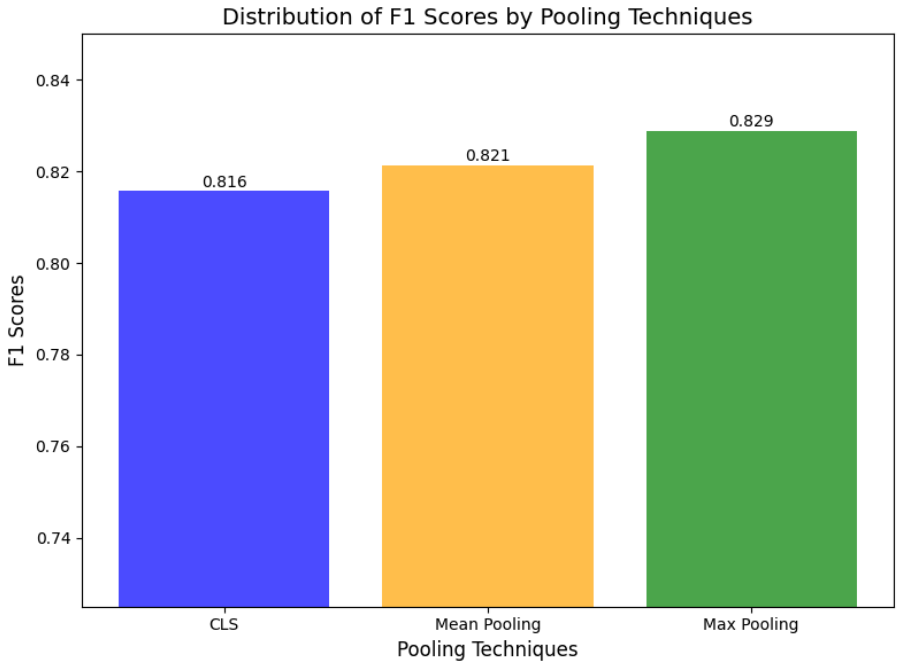
Max Pooling

This method of pooling involves creating a single embedding which has the maximum value for each dimension across all the tokens in the input. This can help highlight the presence of specific features very clearly on smaller sets of data because their significant features are very likely to be captured here. If there are specific tokens that carry a lot of meaning, then max-pooling is very likely to be able to identify them.

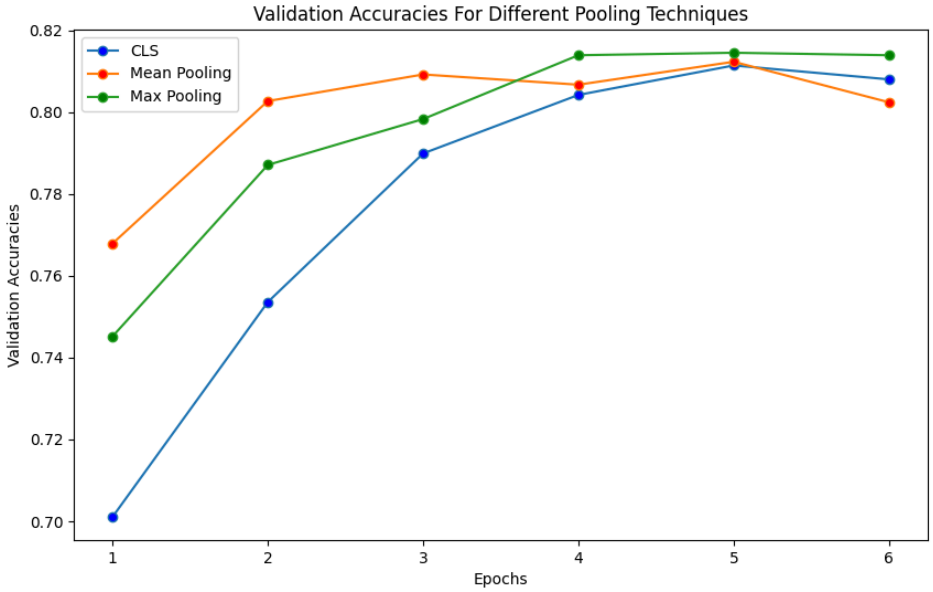
Comparison Between Different Pooling Techniques

Improvements in Accuracy

The F1 scores of all three pooling techniques after 6 epochs of training:



We also monitored the increase in validation accuracy as the models were trained for 6 epochs. Validation accuracy depicts how well the model is able to perform on the validation dataset as it is being trained. Here is how the validation loss of the three models changed over the epochs.



We see that in terms of validation accuracy, all models show significant improvements until the 4th epoch before stagnating.

Speed of Reaching Convergence

- As is visible in the above graph, the CLS pooling started at the lowest validation accuracy out of the three pooling methods but matched the validation accuracy of mean pooling by 5th epoch and further surpassed it.
- Mean pooling on the other hand started at the highest validation accuracy at the start but had the lowest validation accuracy by 6th epoch.
- Max pooling gave the best validation accuracy by the 6th epoch.

Possible Explanation

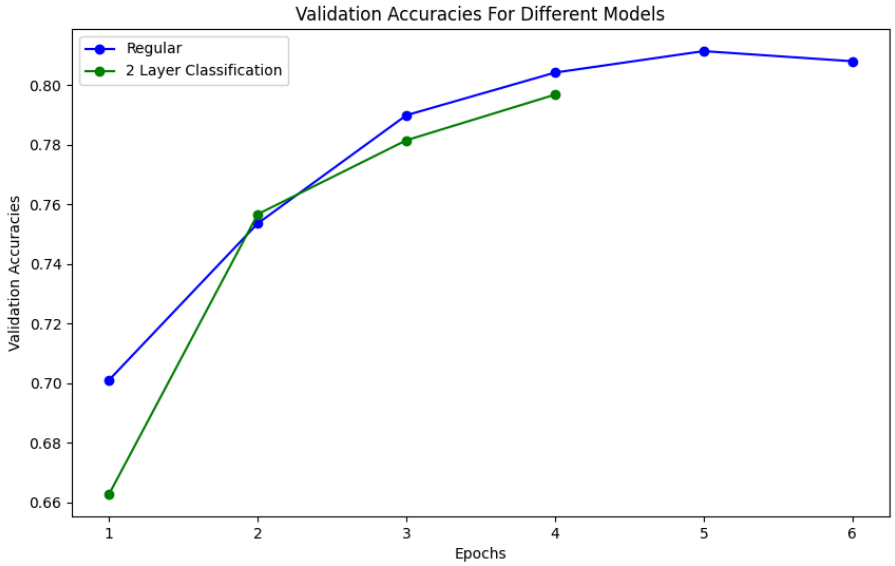
We suspect that max pooling is able to outperform the other two pooling techniques as it is able to focus on the most important elements of the context. As we had seen before, few words occur more frequently in certain classes and max pooling might be able to identify the occurrence of these words and hence classify the articles more accurately. If this is the case the max pooling model might not be able to generalize well in other test cases.

Monitoring Effects of Other Variables

Changing Number of Layers

Our default model worked with a hidden layer of 512 nodes and an output layer of 20 nodes. We tried changing this to a model with 2 hidden layers: one of 512 nodes and one of 256 nodes.The default model had an F1 score of 0.81, and the model with two layers ended up with a roughly similar F1 score of 0.80.

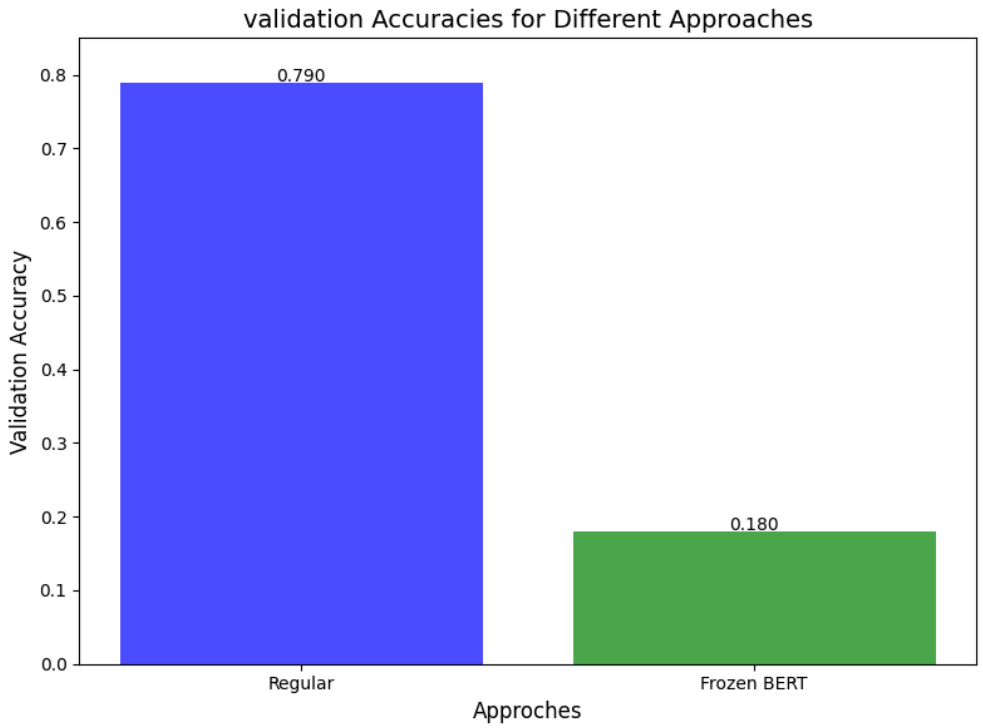
We also analyzed the change in validation accuracy of the these models as they were trained over multiple epochs and here are the results:



The addition of another layer did not drastically change the F1 score after 4 epochs but as seen in the graph, the model with lesser layers was able to converge slightly faster compared to the 2 layer model. This is most likely due to the fact that we have more layers to train before being able to get accurate results.

Freezing BERT's weights

Freezing BERT's internal layers led to a disastrous fall in performance for the model. Whereas the regular model was able to achieve an validation accuracy of 0.7899 after 3 epochs, the model which had BERT's internal layers frozen crashed the accuracy down to 0.1799.



- BERT's pretrained layers encode generic language understanding but lack task-specific adjustments. Without fine-tuning, the model cannot adapt these representations to classify 20 Newsgroups data effectively.
- The classification head relies heavily on meaningful embeddings from BERT. Freezing its layers prevents the embeddings from being optimized for the dataset's nuances.
- The large performance gap highlights the importance of task-specific fine-tuning in extracting full utility from BERT's pretrained knowledge.

Freezing BERT essentially limited the model to work with suboptimal, generic features, which led to the drastic drop in accuracy.

Changing The Activation Function

Rectified Linear Unit (ReLU)

ReLU is a widely used activation function that retains positive inputs as they are and sets negative inputs to zero. It is computationally efficient and helps mitigate the vanishing gradient problem, making it particularly effective for deep networks. However, ReLU can suffer from the "dying neuron" problem, where certain neurons output zero for all inputs and stop learning.

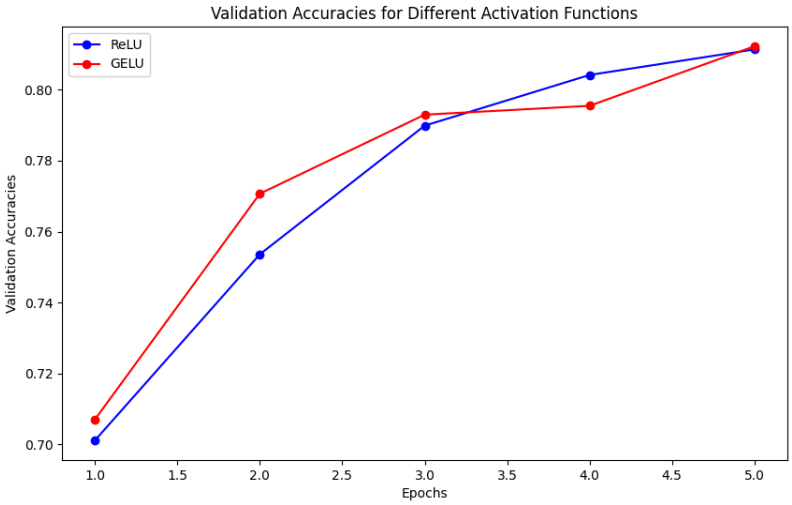
Gaussian Error Linear Unit (GELU)

GELU smoothens the output transition between active and inactive states using a probabilistic approach. Unlike ReLU, which has a hard threshold, GELU uses a sigmoid-based

approximation to apply weights to inputs based on their magnitude. This allows GELU to better model complex patterns and achieve smoother gradient flow, making it particularly effective for transformer-based architectures like BERT.

Comparing Activation Functions

We monitored the validation loss and validation accuracy as both the models were trained and this is how they changed over time



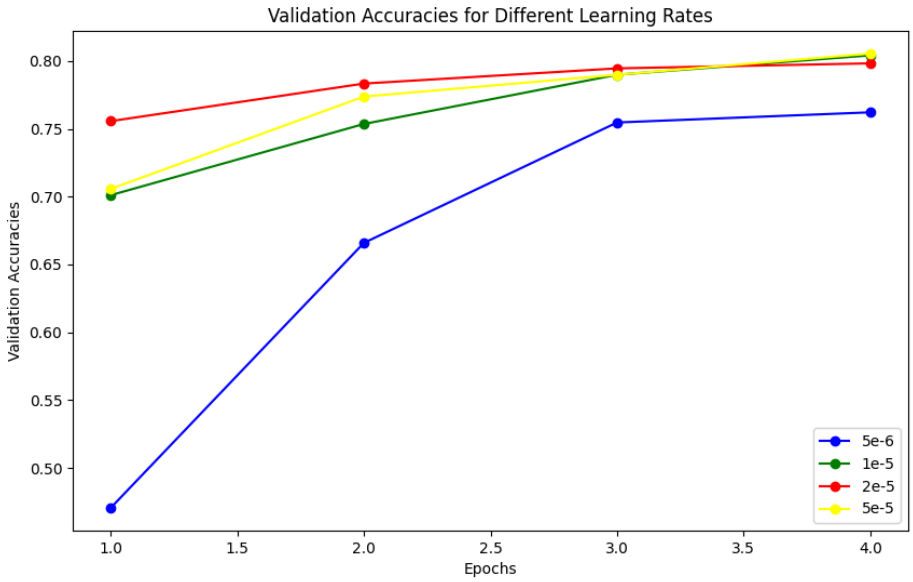
From the validation accuracies across epochs, GELU consistently outperforms ReLU during the initial epochs, with slightly higher accuracy (0.707 vs. 0.7011 in the first epoch and 0.7707 vs. 0.7536 in the second). By the third epoch, both activations show comparable performance, with GELU at 0.793 and ReLU at 0.7899. In later epochs, GELU achieves a marginally higher peak accuracy of 0.8123 compared to ReLU's 0.8114, though both converge similarly.

- GELU facilitates faster convergence in the initial training stages
- Both activations achieve comparable final accuracies, with GELU having a slight edge.
- GELU may be preferred for its early performance gains but comes at a higher computational cost than ReLU.

Changing The Learning Rate

Learning rate is the hyper parameter that controls the size of step taken during optimization in gradient decelent.A too high learning rate can cause the model to overshoot the optimal solution, while a too low learning rate can lead to slow convergence.

We analyzed the impact of different learning rates on the validation accuracies of our model. Below is the graph summarizing our findings:



- **Higher Learning Rates (e.g., 5e-5):** Initially, the validation accuracy with a high learning rate rises quickly, allowing the model to make large weight adjustments early on. Interestingly, this approach leads to the best overall performance by the end of 4 epochs. The model benefits from faster convergence, achieving competitive accuracy in fewer epochs compared to lower learning rates. However, the trade-off is that such learning rates may cause instability or minor oscillations in earlier epochs, but they don't seem to hinder final performance significantly in this case. Higher learning rates might not be able to converge to the most optimal solution as they might overshoot.
- **Lower Learning Rates (e.g., 5e-6):** Although the model converges steadily with lower learning rates, it requires many more epochs to achieve high accuracy. This slower convergence can be seen in the gradual improvement in validation accuracy, which might be beneficial for achieving fine-tuned models, but it is less efficient when rapid convergence is desired. Lower learning rates might also end up at local minima and not converge to optimal solutions. It is preferable to have a lower learning rate after training for a few epochs.
- **Balanced Learning Rates (e.g., 1e-5, 2e-5):** These learning rates strike a balance between the speed of convergence and stability. The model with 1e-5 generally performed well, converging reasonably quickly while avoiding significant instability. However, the highest learning rate (5e-5) still outperformed others in terms of final accuracy, indicating that a larger learning rate can sometimes work better for faster training without sacrificing final performance.

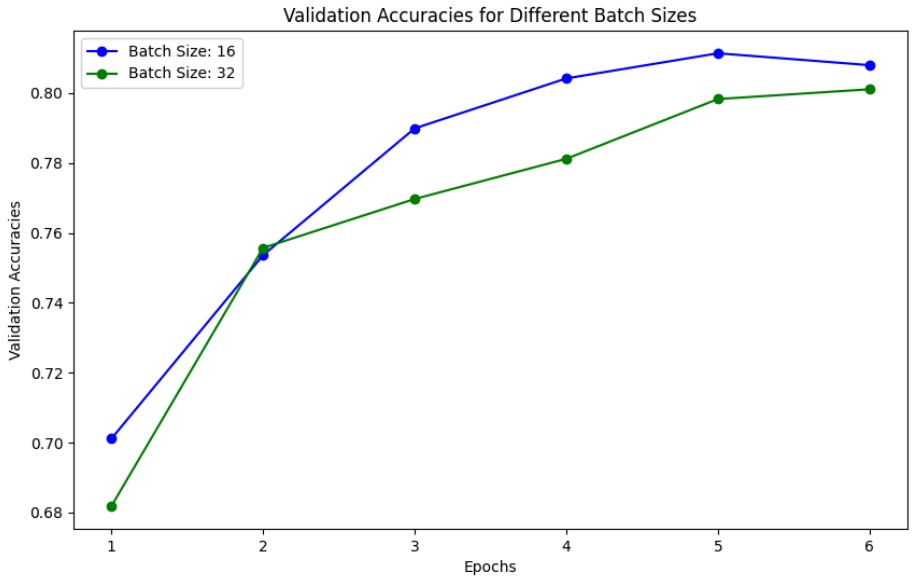
Higher learning rates (e.g., 5e-5) provided the best results within a shorter number of epochs, outperforming the lower learning rates despite potential early-stage instability. While lower

learning rates take smaller steps and adjust weights more accurately which can be helpful in later epochs.

Decaying Learning Rate:

To take advantage of all worlds we implemented a decaying learning rate. The decaying learning rate allows us to start from a higher learning rate in the initial epochs but automatically decrease the learning rate as we progress. This helps the model converge more optimally than fixed learning rate.

Changing The Batch Sizes

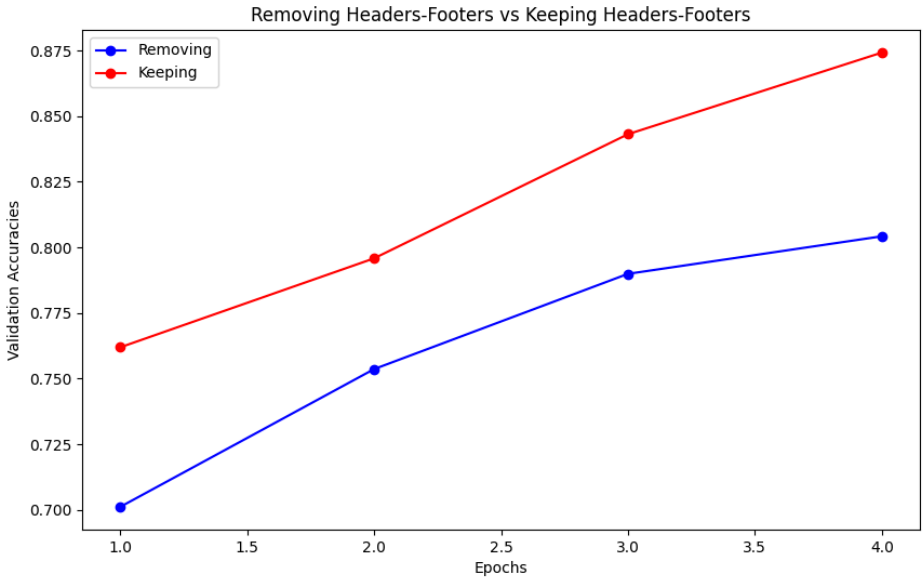


- **Batch Size 16:** Achieved better validation accuracy, likely due to the noisier gradient updates, which can aid in generalization and prevent overfitting. Smaller batch sizes expose the model to more diverse mini-batches, making it harder for the model to memorize patterns and resulting in better performance on unseen data. However, this comes at the cost of longer training times as the weights are updated more frequently.
- **Batch Size 32:** While larger batch sizes provide more stable gradient estimates, they may converge faster initially but can lead to suboptimal generalization, as observed in the lower final accuracy. The model sees the data in larger chunks, which could reduce the exposure to variability in smaller mini-batches. This stability might also make the model more prone to overfitting, as it may memorize patterns in the training data more easily.

Smaller batch sizes may be preferable when aiming for higher accuracy, particularly when overfitting is a concern, though this requires a tradeoff with longer training times. Larger batch

sizes may be more efficient for quick training but might require additional regularization to achieve similar generalization.

Removing Headers, Footers, And Quotes



During our research, we came across an article by scikit-learn that explained why it's important to remove headers, footers, and quotes from the dataset. They pointed out that words in the headers often appear in specific categories, and the model might "overfit" to these words, meaning it would rely too much on them. They also mentioned that whether the sender is connected to a university, shown through their email headers or signature, is a useful feature for identifying the newsgroup. Other useful features include matching names and email addresses of frequent posters. With so many extra clues, the model doesn't need to focus on the actual content of the text, and this leads to high performance without truly understanding the topics.

We compared how the BERT model performed with and without headers, footers, and quotes. We found that BERT performed significantly better when these elements were included, achieving an F1 score of **0.8707** compared to **0.8156** when they were removed. This shows that the model could overfit to the extra data and improve accuracy. However, it's important to note that a higher F1 score in this case isn't necessarily good. It means the model is relying too much on specific data and won't generalize well. As a result, the model would likely perform worse on articles it hasn't seen before.

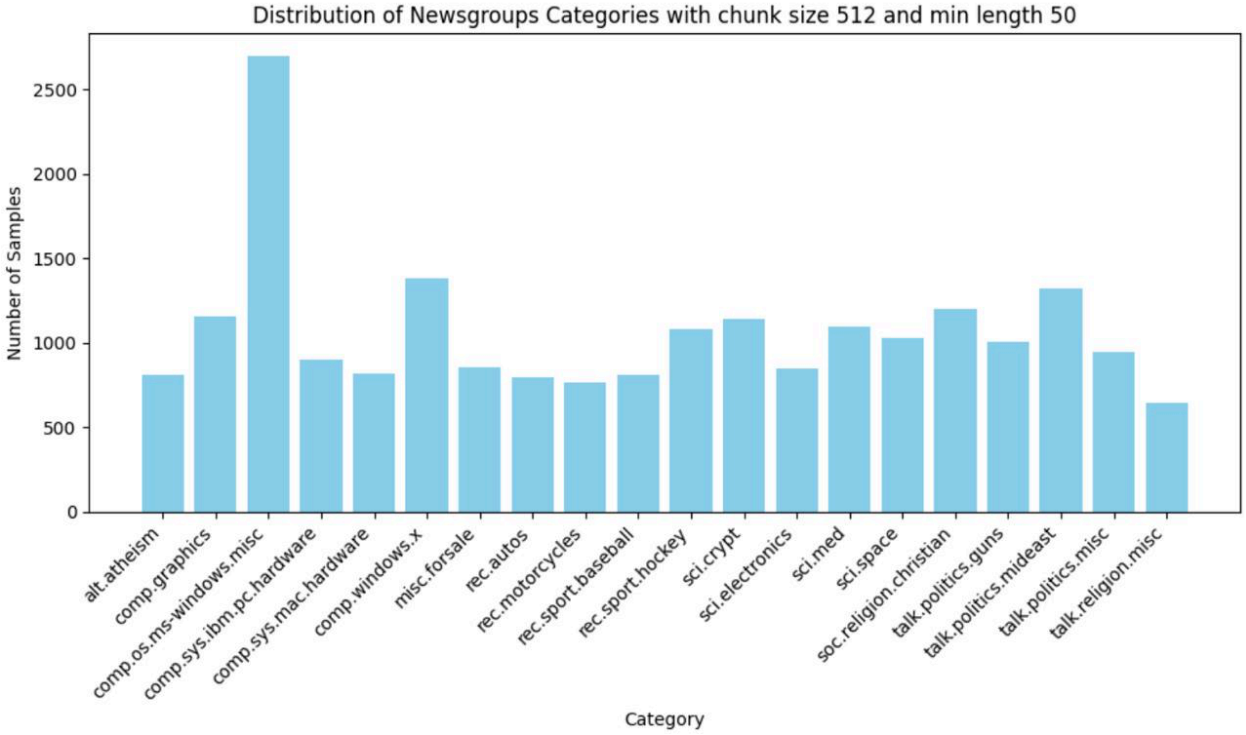
Hence, the F1 score of 0.8156 is more realistic.

The Evaluation Process

Fixing Class Imbalance

When some classes in the dataset have much fewer samples than others then we are said to have a ***class imbalance*** problem. The model tends to ignore or perform poorly on the minority classes while favoring the majority classes which results in biased model predictions. For example, even if the model is unable to accurately recognize instances from the minority classes, it may nonetheless achieve high overall accuracy in classification tasks by mostly predicting the majority class. This leads to poor generalization for underrepresented classes which might actually be fairly common in real-world data.

The class imbalance in this dataset is observed in the following chart:



To address class imbalance, class weights were computed and incorporated into the loss function. Specifically, the `compute_class_weight` function from `sklearn` was used to calculate weights for each class based on their frequency in the training dataset. These weights were then passed to the `CrossEntropyLoss` function, ensuring that the loss penalizes misclassifications of minority classes more heavily than those of majority classes. By doing so, the model learns to pay equal attention to all classes, irrespective of their frequency, resulting in improved performance for underrepresented classes without compromising the overall classification accuracy.

Separation of Data

To ensure that the dataset which the model is evaluated upon is completely unseen to the model during training, we split the dataset into ***train***, ***test***, and ***validation*** datasets right at the start. The model is only exposed to the train dataset when training its weights. The validation dataset allows us to perform checkpointing without leaking the test dataset. The test dataset is much larger than the validation dataset and it is the dataset that is finally used to analyze the actual performance of the model.

Checkpointing

Right after each epoch we evaluate the model on a validation dataset to compute the *validation loss*. This provides an early indication of the model's ability to generalize to unseen data. Additionally, we also compute the *validation accuracy* which directly shows us how well the model has performed on the validation dataset. Whenever the model's validation loss improves, we save the weights of the model, thereby ensuring that the best-performing version of the model has been preserved for the testing procedure later.

Extra: Document Summarization

Inspiration:

Real world articles can vary in size from really small to extremely large. Our model is limited to 512 tokens, and we have to chunk the article to get results. As the model cannot create connections between various chunks, it might miss out on important information that is necessary for accurate classification. This creates a gap that cannot be solved by our chunking and de-chunking logic. Hence we propose a creative approach of summarizing the documents to reduce the size while keeping all the important information of the article.

Implementation Details

We tested this approach using the facebook/bart-large-cnn model that is pre-trained on summarizing various documents. The idea is to give articles larger than 512 tokens to the summarization model so it can reduce their size to less than 512 tokens while maintaining key points from the article. We tested this by telling the summarizer to summarize the articles into a minimum size of 200 to a maximum size of 500 tokens.
Then our BERT classification model was trained on these summarized articles to be able to correctly classify them. We were able to do this for the complete dataset.

Results

The summarization model was able to give a final F1 score of 0.71 compared to the default 0.81. Though we did see a drop in accuracy we do feel that it is because of the fact that the summarization model was not fine tuned for this task and hence was not able to optimally summarize the articles. We feel that given enough training and optimization this will be able to give better results and help more accurately predict the classes for larger documents.

Extra: ML model Dechunking

Inspiration

For larger documents, chunking the article was necessary but to get the final class we needed to combine these results. For this assignment we used the aggregate sum method but we realized that this might not be the most optimal method. For example the starting of the article i.e. the first chunk might play a bigger role than any other chunk and giving it the highest weight might be more optimal.

Solution

Developing a ML model that can take in the final layer of the classification for all the chunks and accurately give a combined final class as output might be more accurate. Through training the model might be able to find patterns through which it can give more weightage to few chunks while reducing weightage for others. This can result in more accurate predictions and hence improving results.

Resources

- https://www.linkedin.com/pulse/fine-tuning-bert-text-classification-20news-group-sharmila-upadhyaya/
- https://www.analyticsvidhya.com/blog/2020/07/transfer-learning-for-nlp-fine-tuning-bert-for-text-classification
- https://medium.com/@datailm/text-classification-mastery-a-step-by-step-guide-using-the-20-newsgroups-dataset-a0a56fc245e0
- https://mccormickml.com/2019/07/22/BERT-fine-tuning/