Comparing Novel Machine Learning Approaches for Recommending Quality Writing

Rohan Bansal
Central High School

Research Question:

How do transformer models with pre-contextualized word embeddings compare to set-based recommendation models, such as a dot-product model, in their ability to classify articles on the basis of the "quality" of their writing, as assessed by expert humans?

Word Count: 4092

Contents

1 Introduction **3**

2 Theory **3**

3 Approaches **4**
- 3.1 NLP Fundamentals 4
  - 3.1.1 One-Hot Encoding 4
  - 3.1.2 Embedding Vectors 5
- 3.2 RankFromSets (RFS) 6
- 3.3 Bidirectional Encoder Representations from Transformers (BERT) 8
  - 3.3.1 Encoders 9
  - 3.3.2 General Working 13

4 Hypothesis/Applied Theory **14**

5 Data Collection and Processing **14**
- 5.1 Collection 14
- 5.2 Cleaning 15
- 5.3 Preparation 15

6 Experiments **16**
- 6.1 Experimental Setup: RankFromSets 16
- 6.2 Experimental Setup: BERT 17
- 6.3 Quantitative Evaluation 18

7 Discussion **19**

8 Conclusion **20**

A Code Base **23**

B Crawling Scripts **23**
- B.1 Longform.org Crawling Script 23
- B.2 Vox Crawling Script 25

C Scraping Scripts **27**
- C.1 Vanilla Scraping Script 27

C.2 Selenium Scraping Script 29

D Data Processing **31**

E Data Distribution **34**

F Training Scripts **34**
F.1 Evenly-Split Batch Generator 34
F.2 Batched Predictions 35
F.2.1 RankFromSets 36
F.2.2 BERT 36
F.3 Recall Calculation 36
F.4 Generate Rankings List 37

G Permission Email **38**
G.1 Email Sent to Mr. Uri Bram of The Browser 38
G.2 Permission Received from Mr. Uri Bram of The Browser 38

1 Introduction

This essay is focused on comparing two modern machine learning approaches in their ability to classify and predict "quality" writing. This type of writing can be thought of as the material published in curated newsletters (such as The Browser) or quality-focused publications such as Longform.org. There seems to be a general parcity of serious work in this space, partially because it may be quite difficult to train a machine to understand what we humans would consider fantastic literature and also because finding adequate data can be difficult and expensive. The two approaches considered are RANKFROMSETS (RFS) [4] and BERT [7], both novel machine-learning approaches. The primary method of comparison between the two models will come from their respective recall on the held out evaluation set. The top 1000 predictions for each model will be considered, and the percentage of true positives will determine the model's score. Other time and computing constraints will also be discussed throughout to provide enhanced context for all results. Hence, the question: *How do transformer models with pre-contextualized word embeddings compare to set-based recommendation models, such as a dot-product model, in their ability to classify articles on the basis of the "quality" of their writing, as assessed by expert humans?*

2 Theory

The concept of machine learning has continued to grow more popular for a multitude of tasks due to its ability to reduce human error and provide more efficient methods for solving real world problems. Prior to the advent of machine learning, programming was generally a practice in which a developer gave instructions to a computer explicitly and the machine executed the commands it received. In contrast, machine learning teaches a computer to "think" or develop a model on its own that can then be utilized to tackle new situations. For example, a classification-based system, such as the ones being used in this model, typically rely on large amounts of labelled data in the form of what is called a **training set**. **Labels** refer to the category of each instance in the dataset, so they would be "quality" (0) or "regular" (1) in this instance. The training set is typically fed to the model in **batches** (for computational speed and even label splits), where the model utilizes its **parameters** to generate predictions for the examples within the batch. These predictions are then compared to the actual labels of the set to calculate the **loss**, which measures the discrepancy between the predictions and

intended results. Finally, an **optimizer** uses **back-propagation** (calculating the derivative of the final output with respect to each parameter) to update the parameters based on the loss. Machine learning typically also utilizes an evaluation and test dataset, which are used to monitor the progress of the model's parameters. The model generates predictions on the evaluation set with a fixed frequency during training with back-propagation turned off. This allows the researcher to ensure that the model is not over-fitting on the training set, or becoming overly specific to the articles being trained on. Once evaluation performance begins to decrease, the model is then used to generate predictions on a test set to get a final list of results on completely new data. Assuming the test performance is adequate, the model can then be used to generate new predictions on unlabelled data, such as for the purpose of a recommendation application system that fetches new articles daily. The specific subset of machine learning that is utilized in this discussion is called natural language processing (NLP) which pertains to any type of model based on text analysis.

3 Approaches

3.1 NLP Fundamentals

The primary challenge in any text-based modeling approach is the representation of data. A computer cannot comprehend words like humans, so it needs a numeric approach for text analysis. This leads to the idea of using ids to map words [14]. For example, consider the sentence: "John went to the park.". We can easily construct a basic dictionary for the words in this sentence and map them to a unique id as such:

{"*john*" : 0, "*went*" : 1, "*to*" : 2, "*the*" : 3, "*park*" : 4}

3.1.1 One-Hot Encoding

This process could also occur for labels/categories that need to be numerically represented, such as colors or publications. However, there is a very obvious issue with this approach when using it on unordered data without obvious spatial relationships. This type of map tells the model that "park" is more important, or weighted more heavily, than "john". If the model calculates averages, it will find the average of the words "went" and "the" to equal "to". These types of relationships are obviously fraught and would cause major errors in the model's

effectiveness. To mitigate this problem, researchers developed the concept of one-hot encoding, which essentially binarizes the process of mapping items to their numeric representations [10]. A matrix is generated with dimensions length of items in entry X length of total dictionary for each example. Each row of the matrix has a single one and all other entries are zero to represent the single word. Using the above example, our sentence representation would become:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Although this mitigates the earlier concerns, there are a variety of disadvantages to this approach. Firstly, the creation of such sparse matrices for each entry in the dataset, which can span millions of entries, is inefficient for storage purposes. It is also plagued by the "curse of dimensionality" as each category/word appends an entire dimension to the matrix [5]. Finally, the adding of items to the dictionary would lead to changing the representations for all previous words. Aside from the structural issues, this approach lacks interpretability. As the items are fed into the model, the one-hot encoded matrix tells the user nothing about the relationship between words. It also breaks down any order of words, which is required for models such as BERT.

3.1.2 Embedding Vectors

These major problems with one-hot encoding led to the development of embeddings and vectorized representations of words in natural language processing. The fundamental premise of word embeddings is the concept that each word maps to a unique n-sized vector of a single dimension. Thus, words can be mapped to ids as they were above, and a separate mapping of ids to vectors can be utilized to convert each word into a corresponding mathematical representation. The embeddings are obviously not directly interpretable by humans, as they capture latent qualities of the word as the model is trained, however embeddings offer a significant advantage over one-hot encoding in their ability to offer comprehensible representations. Similar words will be found to be spatially contiguous. The relationship between words can also be calculated directly through cosine similarity, which is essentially the arccosine of the dot product of two vectors normalized

by their magnitudes. The most famous example of this type of interpretability is the mathematical relationship **king - man + woman = queen** which can be graphically observed in Figure 1, produced by Allen and Hospedales [3]:

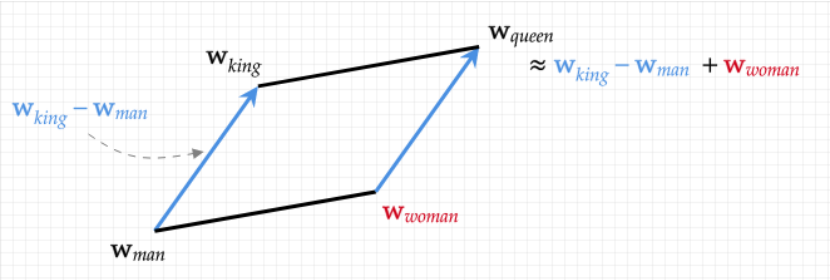

Figure 1: Spatial representation of king - man + woman = queen

This then allows for each example in the dataset to be represented as a unidimensional list of word ids, which can then be mapped to their corresponding vectors when being processed by the model. The data storage advantages are obvious, as each entry becomes easily stored as a single dense list instead of the highly-sparse matrices generated by one-hot encoding. Thus, because of both the data-related and interpretability advantages offered by embeddings, they were utilized for both of the tested models.

3.2 RankFromSets (RFS)

RFS was proposed by Altosaar, Tansey, and Ranganath [4] and was initially used for the task of recommending meals to users based on their prior data of favorite food combinations. I used a slightly modified version in my research. The recommender system utilizes an unordered set of attributes to represent each individual item. An inner-product architeture parametrizes the classifier:

$$f(q, x_m) = \theta_p^\top \left(\frac{1}{|x_m|} \sum_{j \in x_m} \beta_j \right) + \frac{1}{|x_m|} \sum_{j \in x_m} \gamma_j \,. \tag{1}$$

Each aspect of Equation (1) has an intuitive definition:

$\theta_p^\top$: the publication embedding for publication p. In this essay, there is only one publication (that of quality writing), however the simplicity of the model allows for easy scalability to generate predictions for multiple publications.

x_m: the set of unique word tokens in the article. Because the model does not take positional relationships of words into account, the ordered set of tokens can be used to mitigate the effects of the repetition of frequently used words such as *the, and, a* etc.

β_j: the latent quality of each word in the set represented by x_m. This is a vector representing the word's "meaning" that is updated during training. These vectors are summed across the set and normalized by word-list length.

γ_j: the scalar bias of each word in the set represented by x_m. These are used to prevent overfitting and account for qualities of a word that were not represented within the embedding vector (such as frequency). These were also summed and normalized.

Normalization of the sums of biases and vector representations is done to prevent article length from playing an outsized role in predictions. The vector sum is then multiplied with the "quality" writing publication embedding, which has identical dimensions. The normalized word biases are added to generate the final output of the function.

The final probability is defined as:

$$p(y_{qm} = 1 \mid q, m) = \sigma\left(f\left(q, x_m\right)\right) \tag{2}$$

In Equation (2), σ is the sigmoid function, defined by:

$$S(x) = \frac{1}{1 + e^{-x}}, \tag{3}$$

σ takes an input and and generates a final probablity between 0 and 1, which is then rounded to get a positive or negative prediction [18]. The implementation of the model is presented through a custom subclass of the PyTorch Module [16] and [4]:

```
class RankFromSets(nn.Module):
    def __init__(self, n_pubs, n_attrs, emb_size):
        super().__init__()
        self.emb_size = emb_size
        self.publication_embeddings = nn.Embedding(n_pubs, emb_size)
        self.publication_bias = nn.Embedding(n_pubs, 1)
        self.attribute_emb_sum = nn.EmbeddingBag(n_attrs,
```

```
                                                emb_size, mode="mean")
        self.attribute_bias_sum = nn.EmbeddingBag(n_attrs, 1, mode="mean")

    # this method resets default parameters to normal distribution
    def reset_parameters(self):
        for module in [self.publication_embeddings, self.attribute_emb_sum]:
            scale = 0.07
            nn.init.uniform_(module.weight, -scale, scale)
        for module in [self.publication_bias, self.attribute_bias_sum]:
            nn.init.zeros_(module.weight)

    def forward(self, publications, word_attributes, attribute_offsets):
        # get publication embedding and bias
        publication_emb = self.publication_embeddings(publications)
        publication_bias = self.publication_bias(publications)
        # get sum of attribute embedding vectors
        article_and_attr_emb = self.attribute_emb_sum(
            word_attributes, attribute_offsets
        )
        # get sum of attribute biases
        attr_bias = self.attribute_bias_sum(word_attributes, attribute_offsets)
        # dot-product of pub_emb and attr_emb sum
        inner_prod = (publication_emb * article_and_attr_emb).sum(-1)
        # add biases to result and return final logits
        logits = inner_prod + attr_bias.squeeze() + publication_bias.squeeze()
        return logits
```

3.3 Bidirectional Encoder Representations from Transformers (BERT)

BERT was proposed by Devlin et al. [7] and obtained state-of-the-art results on a variety of NLP-related tasks, including sentence classification and next word predictions. The model is predicated on the concept of transformers, which were released in 2017 and provided a method for ensuring that the sequential nature of text data was accounted for in machine learning models [20]. Prior to their advent, the primary method for encoding spatial word relationships was through the use of Recurrent Neural Networks (RNN). They input tokens sequentially and create

a new latent state based on the previous $n-1$ tokens combined with the recently inputted nth token. The general input process is depicted by Olah [15]:

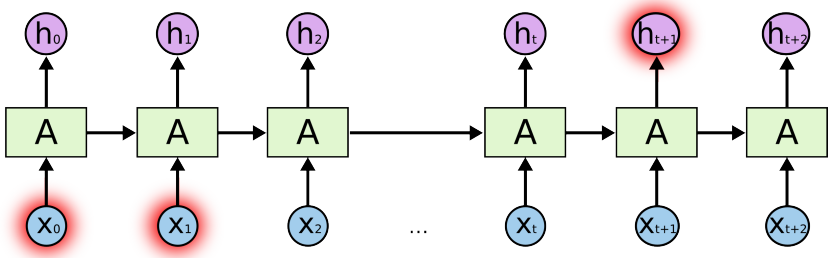

Figure 2: RNN sequencing example (displays sequential input handling and state updating)

There were a variety of problems that hindered this approach, including a lack of concrete connection between early and later words of a sequence. Additionally, as only a single hidden state was stored after each token was added, the weight (or visibility) of the beginning of the input could often become hidden and difficult for the model to recover [11]. For example, if a model is being trained to translate from English to Spanish, it can be reasonably assumed that the first few words of the English input have a strong correlation to the correct beginning tokens of the Spanish Output. However, the model only contains the final latent state of the input, thus making it difficult for it to correctly decipher the beginning of the Spanish translation. Transformers were created to offer an alternative to this process, and they rely on a concept called attention and involve two components, an encoder and a decoder [20]. As BERT relies only on encoders, this paper will focus exclusively on them.

3.3.1 Encoders

An encoder consists of two sub-layers, a self-attention layer that calculates the relation of each token to other tokens in the same input and a feed-forward network which works like a normal multi-layer perceptron. Self-attention is essentially the concept of helping the model comprehend the relationships of words to other tokens in the input. For example, consider the sequence:

The money has gone missing. Where is it?

For a human, it would be fairly easy to understand that **it** is referring to the **money**. But for a machine, it can be much more difficult to gather these types of contextual clues, especially if the corresponding words are separated [11]. Thus, self-attention generates "scores" for each input token with every other token (including itself) which helps it understand the relationships that are not explicitly present in the direct translation.

The encoder takes the raw embedding sequence as input which is passed to the self-attention layer. The embeddings are slightly modified before being fed to the model however. It is important for the computer to have some sense of the spatial relationships between words (the relative distances) to help with the positional aspect of the sequence. Thus, the model utilizes learned positional embeddings, which are the same size as the word embeddings themselves [7]. These positional vectors are added to the token vectors to create the real input:

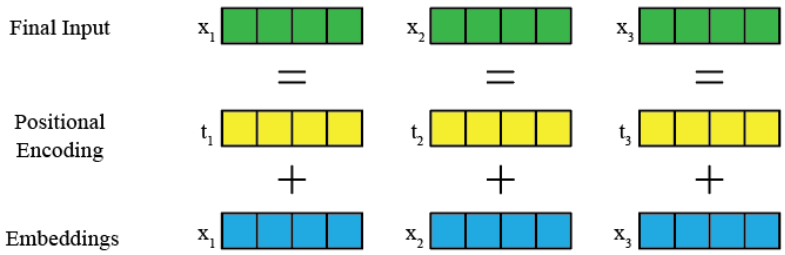

Figure 3: The basic addition of positional encodings to the input vectors before being fed to the model [2].

This layer then utilizes three matrices to generate a key, value, and query for each input token. For BERT, the embedding vectors are of length 768, and each of these matrices has dimensions 768×64. For each token, the embedding vector is multiplied by the three matrices to get the key, query, and value vectors which each have a length of 64. Then, the dot-product between the query vector and each token's key is calculated to generate a score. This can be observed here:

The scores are then normalized with a $softmax$ function, defined by:

$$S(y_i) = \frac{e^{y_i}}{\sum_{j \in y} e^{y_j}}, \tag{4}$$

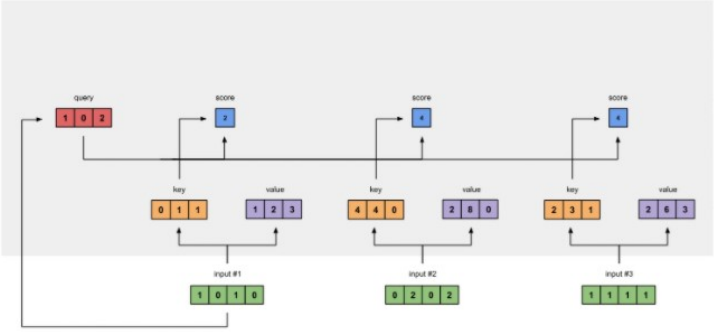

Figure 4: Initial score generation for first token by taking a dot-product between the query vector and the key vectors for each token [12].)

Equation (4) takes a list of numbers and normalizes them into a probability distribution based on the exponential values of the inputs [6]. These normalized scores are multiplied with the corresponding value vectors, which then are summed element-wise to get a final score for the token:

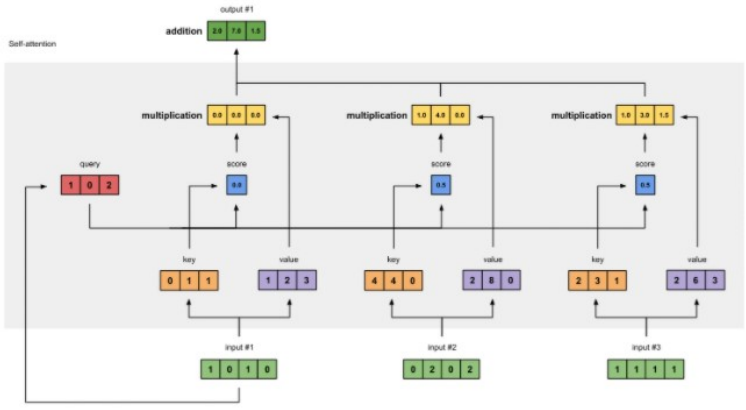

Figure 5: The SoftMax function is applied to the initial score, then multiplied with the value vectors. The results are then summed to generate a final output for the first token. [12].

This process is performed for each input token and is vectorized for faster computation. Transformers are able to utilize this process multiple times per encoder

to generate what is known as multi-headed attention. In the case of BERT, this process is done 8 separate times, each with a different key, query, and value matrix to get 8 vectors of length 64 for each input token. This makes the attention layer even more powerful and allows for more concrete connections to be encoded between words [20]. However, the feed-forward layer is not expecting each token to have multiple associated vectors, thus these outputs must be combined first before sending. To alleviate this problem, the vectors are all concatenated together to create a vector of length 512 (8×64), then multiplied with another parameter matrix of dimensions 512×768, to return to the original embedding size of 768 [7], which is outlined here:

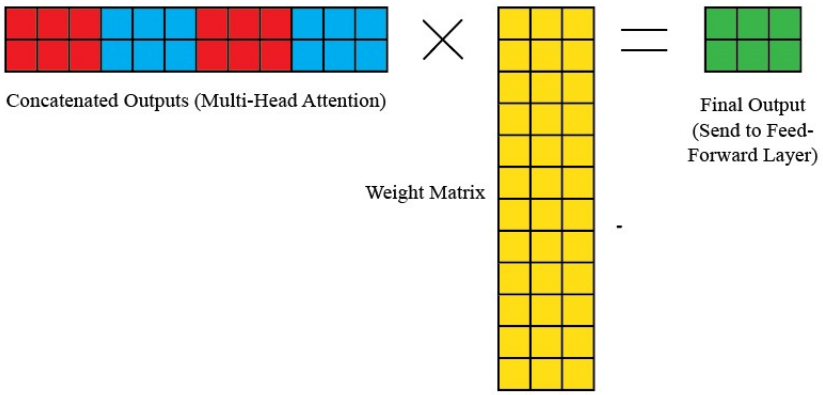

Figure 6: The outputs are concatenated as above then multiplied with the weighted matrix that is updated throughout training to get a final output with a 768 long vector for each token.

The overall attention process can be summarized as the following:

$$Attention\,(Q, K, V) = softmax\left(\frac{QK^\top}{\sqrt{d_k}}\right)V, \tag{5}$$

Equation (5) outlines the attention function for a (Q, K, V) value tuple. The results of the query and key product are divided by the square root of the size of the vectors (8 for BERT) to normalize the outputs before applying softmax [20].

This attention layer output is then passed to the fully-connected feed forward neural network, which takes inputs of length 768 and returns an output of identical size based on a non-linear activation function as such:

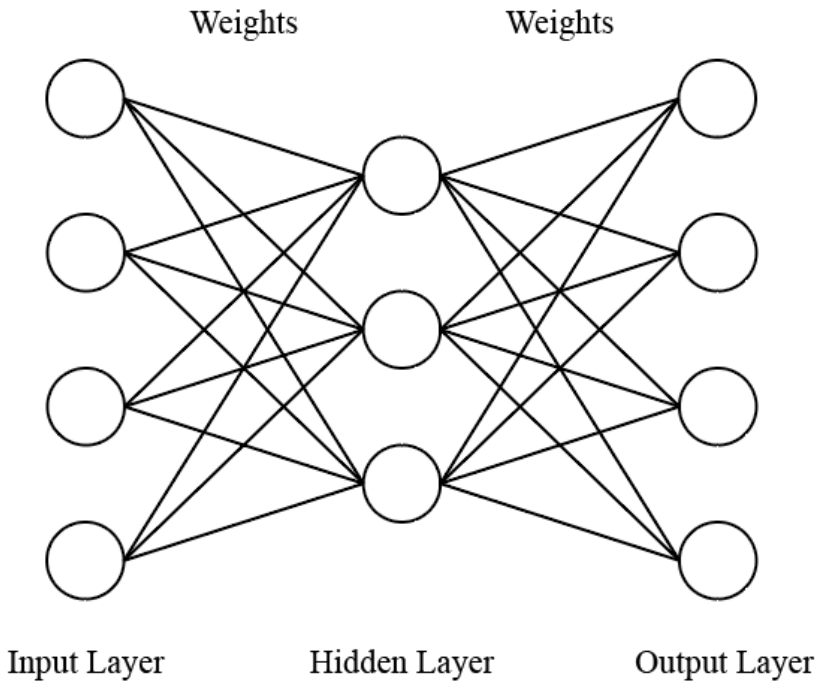

Figure 7: A sample fully-connected feed forward network with a hidden layer

The workings of a neural network have been extensively documented throughout machine learning literature, thus they will not be discussed in detail. However, the basic premise is very similar to the workings of RFs, where a dot-product is calculated with learned weights, and then passed through an activation function, such as σ [13].

3.3.2 General Working

BERT employs 12 of these encoders, with both self-attention and feed forward components, stacked on top of each other. Because the model is being used for classification, another final neural network is placed on top that takes the outputs of the encoders and generates a single probability for the entry [17].

4 Hypothesis/Applied Theory

Based on the information presented about both approaches, it is very clear that RANKFROMSETS is more flexible than BERT. BERT utilizes over 100 million parameters and thus requires significantly more computing power and GPU memory to be run even on smaller datasets. However, this large gap in parameters and empirical evidence suggests that BERT will outperform RFS on practically any NLP task as it is able to better fit unique datasets [7].

This experiment will analyze both the time taken to train the models on the same data, but also the recall/performance of the models. Recall will be tested by looking at the percentage of true positives in the top 1000 predictions of each model. I hypothesize that BERT will have a moderate advantage in recall performance, but RFS will overfit on the data many magnitudes faster.

5 Data Collection and Processing

The majority of the work necessary for training machine learning models pertains to the actual data being trained on. Many steps are involved in acquiring proper data and cleaning it in a manner that makes training both conducive and effective.

5.1 Collection

I began by collecting data from longform.org and longreads.com, which are both publications that scour the web for quality content. The crawling/scraping scripts I wrote and used can be seen in Appendix B and C. I then asked the CEO of The Browser for his content and he graciously provided access to the company's archives (Refer to Appendix G). These articles were combined to be the positive examples. To collect negative data, I began with scraping a few major news sites (Vox, Guardian). I then decided to scrape specific publications that frequently appear in the quality writing locations, as these would provide a better test of the model's ability to draw a distinction from content that is actually hand-picked for its writing style. This included sources such as Rolling Stone and Smithsonian Magazine who both pride themselves on longform content. I also decided to use the Fake News Corpus which provides nearly 9 million articles from various fringe parts of the Internet, including hate speech, genuine fake news, and conspiracy

theories. These were generally used for non-training purposes to ensure that the model was not susceptible to "bad" content.

5.2 Cleaning

The majority of the cleaning and processing was done via the direct crawling/scraping scripts. JSON files were generated for each publication, which included fields for publication, text, and links. The only major caveat with this type of scraping is the possibility for certain sites to detect the script and send back an error message. For this step, I manually checked a few articles from each publication scrape to make sure that proper content was being received by the request. I also filtered out articles that were under 75 words in length as these were primarily error messages or not technically full length articles.

5.3 Preparation

After collection, the text and publication fields had to be mapped to ids for the model. For publications, I mapped all of the quality content I had collected to publication id 0. The remaining publication ids were essentially irrelevant as the model was only relying on the embedding vector for publication 0 to generate predictions. The text then needed to be mapped to the corresponding tokens. The tokenizer library from huggingface made this process easy, as it allows for a simple input of a normal sequence of text (in the way that the article content was saved) and generates a dataset of both tokenized words and the corresponding ids which can be observed here:

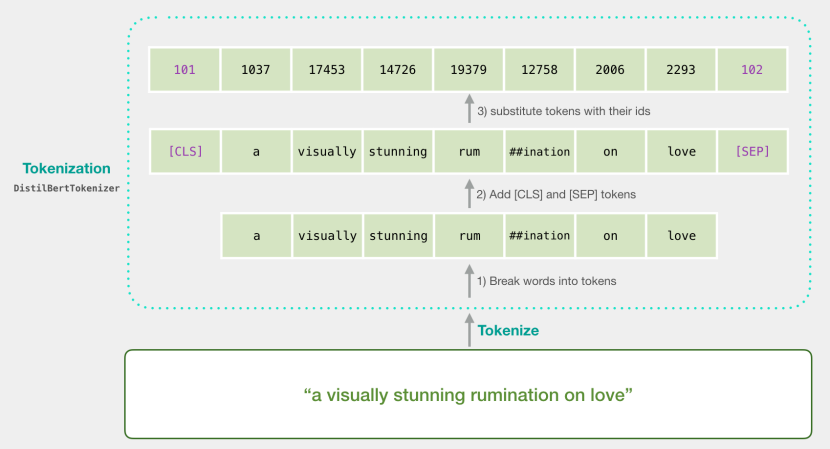

Figure 8: Full Scale Tokenization Example, where an English sentence is converted to tokens. [1]

In order to limit the need for separate data files for both approaches, I chose not to add any other pre-processing to the text fields, such as removing duplicates or adding custom tokens. BERT relies on the spatial relationship between words in the sequence, and thus I wanted to maintain that order and only generate changes, or perform additional processing, when needed (see Appendix D). This was done in the actual training scripts themselves to reduce overhead.

6 Experiments

For both experiments, 15% of the data was held out as a test and validation set each. The exact breakdowns of the sets can be seen in Appendix .Recall was utilized to measure performance (% of true positives in top 10 and 1000 of predictions). Gridsearchs were used to run models with all posible combinations of chosen configurations. The models were also evaluated with early-stopping (training was stopped when model exhibited maximum performance on the validation set) to prevent over-fitting.

6.1 Experimental Setup: RankFromSets

RFS was run using RMSProp optimizer [19] with a momentum of 0.9 and a grid-search was done over learning rates of $\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$, embedding sizes of

$\{10, 25, 50, 100, 500, 1000\}$, and a decision of whether or not to pre-initialize the model with BERT embeddings. The basic training loop is shown here:

```
for step, batch in enumerate(cycle(train_loader)):
    # turn to training mode and calculate loss for backpropagation
    torch.enable_grad()
    model.train()
    optimizer.zero_grad()
    publications, articles, word_attributes, attribute_offsets, real_labels = ba
    publication_set = [args.target_publication] * len(real_labels)
    publication_set = torch.tensor(publication_set, dtype=torch.long)
    publication_set = publication_set.to(device)
    articles = articles.to(device)
    word_attributes = word_attributes.to(device)
    attribute_offsets = attribute_offsets.to(device)
    logits = model(publication_set, articles, word_attributes, attribute_offsets
    L = loss(logits, labels)
    L.backward()
    optimizer.step()
    running_loss += L.item()
```

6.2 Experimental Setup: BERT

As BERT already comes "pre-trained", I fine-tuned it to the collected data with the AdamW optimizer and a linear learning rate scheduler with warm up steps according to best practices, as outlined by Devlin et al. [7] and Wolf et al. [21]. The model used a batch size of 32, and articles had maximum length of 512 tokens. A grid search was performed over learning rates of $\{2, 3, 4, 5\} \times 10^{-5}$, warmup steps of $\{10^2, 10^3, 10^4\}$, and total training steps $\{10^2, 10^3, 10^4, 10^5\} \times 5$. The basic training loop is shown here:

```
for step, batch in enumerate(cycle(train_loader)):
    # turn to training mode and calculate loss for backpropagation
    torch.enable_grad()
    optimizer.zero_grad()
    word_attributes, attention_masks, word_subset_counts, real_labels = batch
    word_attributes = word_attributes.to(device)
    attention_masks = attention_masks.to(device)
    logits = model(word_attributes, attention_masks)[0]
```

```
logits = torch.squeeze(logits)
L = loss(logits, labels)
L.backward()
if args.clip_grad:
    nn.utils.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()
scheduler.step()
running_loss += L{}.item()
```

6.3 Quantitative Evaluation

The best performing models from both approaches were chosen, and were evaluated via recall on the test set.

Recommendation Model	Recall @ 1000 (%)
RANKFROMSETS	**53.1**
BERT	46.6

Table 1: RFS outperforms BERT in an offline evaluation on the test dataset when predicting which articles would be featured at The Browser

Additionally, the time taken to train the models is shown:

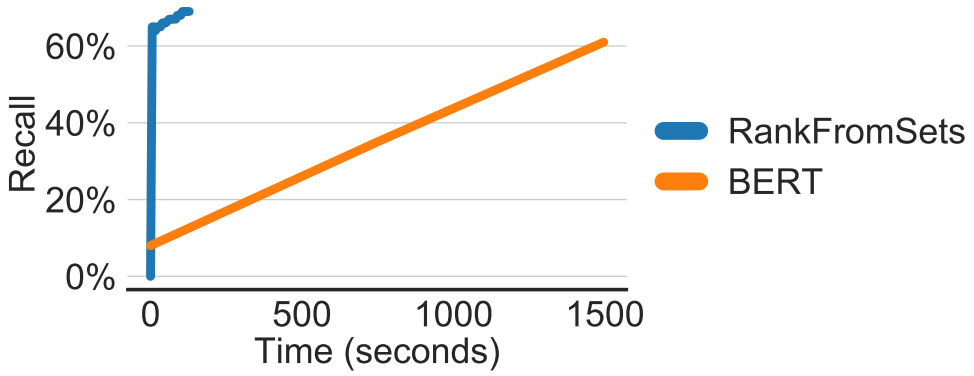

Figure 9: RFS achieves better performance faster than BERT in terms of validation recall during training.

7 Discussion

My hypothesis that BERT would outperform RFS was proved to be incorrect, as RFS actually performed over 6% better as shown in Table 1. However, my other inference that RANKFROMSETS would overfit faster was shown to be correct by Figure 9.

The results were obviously statistically significant, which can easily be deduced by the breakdown of the datasets (Refer to Appendix E) . Based on this data distribution of the test set, a completely random sorting of the articles would result in an anticaptory recall @ 1000 of 0.15% or roughly 2 true positives. The performance of both models was much greater than this, indicating that the models were able to learn some intrinsic qualities of the text that allowed them to generate predictions on untrained data. However, the difference between the two models' performance is still debatable and could simply be due to random error, something which was indicated by the qualitiatative analysis of the predictions by editors at The Browser. Despite this, even comparable results by RFS indicate a better result than BERT, due to the drastically reduced training and prediction time and the simplistic mathematical framework.

I was, however, intrigued by my initial hypothesis being incorrect, as BERT should theoretically have outperformed RFS. There were a few possible explanations. Firstly, machine learning approaches sometimes struggle on certain tasks because of the nature of the dataset, so it is possible that BERT was simply unable to fit the data due to an inherent mathematical hindrance in its workings, although this is unlikely [8]. It is also possible that BERT overfit the training set in an atypical manner. Because of BERT's inherent complexity, it can be prone to overfitting a dataset due to the assumption of more complicated relationships between the variables and labels. In contrast, RANKFROMSETS assumed a simple, linear relationship which requires more data to overfit [8]. It is also possible that there is no direct causational relationship between the words of an article and its "quality", thus the relationships learned by both models were specific only to the dataset chosen. This is also unlikely, as the performance seen was significantly higher than random, however it is a common issue with machine-learning. Finally, BERT typically requires huge loads of data to effectively train, due to its 100 million+ parameters. There is a chance that the training data was simply not large enough for BERT to effectively fit on [9].

For future work, it would be important to introduce regularization into the training

to further penalize incorrect predictions. This may hinder RFS' ability, but could boost BERT. Collecting more data could also provide more insight into whether a lack of examples contributed to the final results.

8 Conclusion

The purpose of this experiment was to compare two modern machine-learning approaches to NLP in their ability to recommend writing. The theoretical aspects of both models were explored, including the mathematical framework, and grid-searchs were run to determine the best-performing models. It is difficult to draw a direct conclusion in this type of exeriment because of the limited dataset, and the other potential limitations discussed above.

However, when answering the initial research question, RANKFROMSETS outperformed BERT in recall at 1000 on the test set and generated better results significantly faster. A novel set-based dot-product approach was able to suggest "high-quality" writing at a rate much higher than random. RFS can also be implemented in an online application or command-line tool to help readers and curators filter large amounts of new articles to the more relevant and interesting reads.

References

[1] Jay Alammar. *A Visual Guide to Using BERT for the First Time*. Nov. 2019. URL: http://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/.

[2] Jay Alammar. *The Illustrated Transformer*. June 2018. URL: http://jalammar.github.io/illustrated-transformer/.

[3] Carl Allen and Timothy Hospedales. "King - man + woman = queen: the hidden algebraic structure of words". In: *The University of Edinburgh* (July 2019). URL: https://www.ed.ac.uk/informatics/news-events/stories/2019/king-man-woman-queen-the-hidden-algebraic-struct.

[4] Jaan Altosaar, Wesley Tansey, and Rajesh Ranganath. "RankFromSets: Scalable Set Recommendation with Optimal Recall". In: *American Statistical Association Symposium on Data Science & Statistics* (2020).

[5] Richard Bellman. *The Theory of Dynamic Programming*. Defense Technical Information Center, 1954.

[6] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*. MIT Press, 2017.

[7] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Association for Computational Linguistics*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10.18653/v1/N19-1423. URL: https://www.aclweb.org/anthology/N19-1423.

[8] Pedro Domingos. "A few useful things to know about machine learning". In: *Communications of the ACM* 55.10 (2012), pp. 78–87.

[9] Santiago González-Carvajal and Eduardo C. Garrido-Merchán. *Comparing BERT against traditional machine learning text classification*. 2020. arXiv: 2005.13012 [cs.CL].

[10] David Money Harris and Sarah L. Harris. *Digital design and computer architecture*. Morgan Kaufmann, 2015.

[11] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[12] Raimi Karim. *Illustrated: Self-Attention*. Dec. 2019. URL: https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a.

[13] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[14] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. "Natural language processing: an introduction". In: *Journal of the American Medical Informatics Association : JAMIA* (2011). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/.

[15] Christopher Olah. *Understanding LSTM Networks*. Aug. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[16] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[17] Chi Sun et al. *How to Fine-Tune BERT for Text Classification?* 2019. arXiv: 1905.05583 [cs.CL].

[18] Markov Svetoslav and Nikoley Kyurkchiev. *Sigmoid Functions: Some Approximation and Modelling Aspects*. Lap Lambert Academic Publishing, 2015.

[19] Tijmen Tieleman and Geoffrey Hinton. “Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude.” In: *COURSERA: Neural Networks for Machine Learning* (2012).

[20] Ashish Vaswani et al. “Attention Is All You Need”. In: *CoRR* abs / 1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[21] Thomas Wolf et al. “HuggingFace’s Transformers: State-of-the-art Natural Language Processing”. In: *ArXiv* abs/1910.03771 (2019).

A Code Base

All code utilized in this project can be found at https://github.com/rohanbansal12/extended_essay. Both approaches, RANKFROMSETS and BERT, have their own folders in the repository. Crawling/Scraping scripts can be found in the data-collection folder. The files used to build this paper in LaTeX are found in the EE folder. I have included certain examples and relevant scripts throughout the paper and appendices.

B Crawling Scripts

A few sample crawling scripts that were utilized have been included.

B.1 Longform.org Crawling Script

```
@dataclass_json
@dataclass
class LongformArticleUrl:
    url: str
    title: str
    source: str

class WriteThread(Thread):
    def __init__(self, queue: Queue, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.queue = queue
        def run(self):
        with open(OUTPUT_FILE, 'a') as output_file:
            output_file.write("[\n")
            first_entry = True
                        while True:
                            article = self.queue.get()
                                if article is None:
                                output_file.write("\n]")
                                break
                                article_json = article.to_json(indent=4)
                                if first_entry:
                                first_entry = False
```

```
                        else:
                            output_file.write(",\n")
                            output_file.write(article_json)

class ScrapeThread(Thread):
    def __init__(self, chunk, queue: Queue, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.chunk = chunk
        self.queue = queue
        def run(self):
        for i in self.chunk:
            try:
                print(f'Getting articles from list page {i}')
                article_list_page = get(f"{BASE_URL}{i}")
                soup = BeautifulSoup(article_list_page.text, "html5lib")
                articles = soup.find_all('article',
                    {'class': 'post--single'})
                for article in articles:
                    link = article.find('a',
                        {'class': 'post__link'})
                    title = article.find('span',
                        {'class': 'post__title__highlight'})
                    source = article.find('a',
                        {'class': 'post__permalink'})
                    print(link)
                    if (title is None or
                        title.string is None or
                        source is None):
                        continue
                    article_url = LongformArticleUrl(url=link['href'],
                        title=str(title.string.strip()) or '',
                        source=source['href'])
                    self.queue.put(article_url)
            except Exception as e:
                print(f'Something went wrong when scraping: {e}')
                print("----------------------------------------")
```

B.2 Vox Crawling Script

```
@dataclass_json
@dataclass
class VoxArticleUrl:
    url: str
    title: str
    month: int
    year: int

YEARS = [str(month) for month in range(2014,2020)]
MONTHS = [str(month) for month in range(1,13)]

class WriteThread(Thread):
    def __init__(self, queue: Queue, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.queue = queue

    def run(self):
        with open(OUTPUT_FILE, 'a') as output_file:
            output_file.write("[\n")
            first_entry = True
            while True:
                article = self.queue.get()
                if article is None:
                    output_file.write("\n]")
                    break
                article_json = article.to_json(indent=4)
                if first_entry:
                    first_entry = False
                else:
                    output_file.write(",\n")
                output_file.write(article_json)


class ScrapeThread(Thread):
    def __init__(self, chunk, queue: Queue, *args, **kwargs):
        super().__init__(*args, **kwargs)
```

```
        self.chunk = chunk
        self.queue = queue
    def get_urls(self, year, month):
        page = 1
        while True:
            sleep(1)  # Prevent rate limiting
            print(f'Getting articles for {year}-{month}, page {page}')
            return_data = get(f"{BASE_URL}/{year}/{month}/{page}",
                headers={'Accept': 'application/json'})
            if return_data.status_code != 200:
                print(f"Received status {return_data.status_code}")
                if return_data.status_code != 429:
                    return
                else:
                    sleep(10)
            else:
                page = page + 1
                response = return_data.json()
                soup = BeautifulSoup(response['html'],
                    "html5lib")
                yield soup
                if not response['has_more']:
                    break
    def run(self):
        for year, month in self.chunk:
            try:
                for html in self.get_urls(year, month):
                    h2s = html.find_all('h2')
                    for h2 in h2s:
                        a = h2.find('a')
                        title = a.string
                        url = a['href']
                        print(title,url)
                        vox_url = VoxArticleUrl(title=str(title),
                            url=str(url),
                            month=int(month),
                            year=int(year))
                        self.queue.put(vox_url)
```

```
            except Exception as e: # Best effort
                print(f'Something went wrong when scraping: {e}')
```

C Scraping Scripts

The majority of the scraping utilized the same code for all sources, in contrast to the crawling scripts, which were based on the specific HTML layout of the websites. I did utilize an alternative method for difficult to scrape sites, and both are presented here.

C.1 Vanilla Scraping Script

A Multi-Threaded Scraper that utilizes the simple Requests and Newspaper packages.

```
import json
from dataclasses import dataclass, field
from dataclasses_json import dataclass_json
from datetime import datetime
from newspaper import Article
from bs4 import BeautifulSoup
from typing import List
from queue import Queue
from threading import Thread

@dataclass_json
@dataclass
class ScrapedArticle:
    title: str = ''
    text: str = ''
    url: str = ''
    source: str = ''

class WriteThread(Thread):
    def __init__(self, queue: Queue, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.queue = queue
```

```
    def run(self):
        with open(OUTPUT_FILE, 'a') as output_file:
            output_file.write("[\n")
            first_entry = True
            while True:
                article = self.queue.get()
                if article is None:
                    output_file.write("\n]")
                    break
                article_json = article.to_json(indent=4)
                if first_entry:
                    first_entry = False
                else:
                    output_file.write(",\n")
                    output_file.write(article_json)


class ScrapeThread(Thread):
    def __init__(self, urls, queue: Queue, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.urls = urls
        self.queue = queue
    @staticmethod
    def scrape(url):
        article = Article(url['url'])
        article.download()
        article.parse()
        soup = BeautifulSoup(article.html, 'lxml')
        ga = ScrapedArticle()
        ga.url = url['url']
        ga.title = url['title']
        ga.text = article.text
        ga.source = url['source']
        return ga
    def run(self):
        for url in self.urls:
            try:
```

```
                print(f"scraping {url['url']}")
                article = ScrapeThread.scrape(url)
                self.queue.put(article)
            except Exception as e: # Best effort
                print(f'ScrapeThread Exception: {e}')
```

C.2 Selenium Scraping Script

A linear scraper that utilizes Selenium headless Chrome browser and the Newspaper package.

```
import pandas as pd
from dataclasses import dataclass, field
from dataclasses_json import dataclass_json
from newspaper import Article
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.chrome.options import Options

@dataclass_json
@dataclass
class ScrapedArticle:
    title: str = ''
    text: str = ''
    url: str = ''
    source: str = ''

# get necessary capabilites and options for selenium Chrome webdriver
def generate_driver_settings(proxy):
    # generate chrome options for proper user-agent
    chrome_options = Options()
    chrome_options.add_argument("--headless")
    # chrome_options.add_argument("--no-sandbox")
    chrome_options.add_argument("--ignore-certificate-errors")
    prefs = {
        "profile.managed_default_content_settings.images": 2,
        "profile.default_content_setting_values.notifications": 2,
```

```
            "profile.managed_default_content_settings.stylesheets": 2,
            "profile.managed_default_content_settings.cookies": 2,
            "profile.managed_default_content_settings.javascript": 1,
            "profile.managed_default_content_settings.plugins": 1,
            "profile.managed_default_content_settings.popups": 2,
            "profile.managed_default_content_settings.geolocation": 2,
            "profile.managed_default_content_settings.media_stream": 2,
        }
        chrome_options.add_experimental_option("prefs", prefs)
        chrome_options.add_argument("--log-level=3")
        # generate PROXY capabilites
        capabilites = webdriver.DesiredCapabilities.CHROME.copy()
        capabilites["acceptInsecureCerts"] = True
        """
        capabilites["proxy"] = {
            "httpProxy": proxy,
            "proxyType": "MANUAL",
            "autodetect": False,
        }
        """
        return chrome_options, capabilites

    # basic scrolling function for selenium infinite scroll pages
    def scroll(driver, timeout, count, pixels):
        # Get scroll height
        last_height = driver.execute_script("return document.body.scrollHeight")
        for x in range(count):
            # Scroll down to bottom
            driver.execute_script(f"window.scrollTo(0, {pixels});")
            # Wait to load page
            time.sleep(timeout)
            # Calculate new scroll height and compare with last scroll height
            new_height = driver.execute_script("return document.body.scrollHeight")
            if new_height == last_height:
                # If heights are the same it will exit the function
                break
            last_height = new_height
```

```
# get chrome_options and capabilites for selenium
chrome_options, capabilites = generate_driver_settings(args.proxy)
driver = webdriver.Chrome(
    args.chromedriver_path, options=chrome_options,
)

def get_entries(links_df, driver, proxies, headers):
    for idx, row in enumerate(links_df.itertuples()):
            if "www" in row.link_url:
                base = "https://www." + str(row.url)
            else:
                base = "https://" + str(row.url)
            try:
                url = row.link_url
                header_copy = headers.copy()
                driver.get(url)
                scroll(driver, 5, 1, 500)
                content = driver.page_source
                soup = BeautifulSoup(content, features="html5lib")
                article = Article("")
                article.set_html(content)
                article.download_state = 2
                article.parse()
                ga = ScrapedArticle()
                ga.url = base
                ga.title = row.title
                ga.text = article.text
                ga.source = row.source
                return ga
```

D Data Processing

I created an Articles class that made dataset handling easy. It includes a built-in mapping function (map_items), in addition to other methods, such as sampler creations, that were utilized in the rest of the experiment.

```
import numpy as np
from tokenizers import BertWordPieceTokenizer
import torch
import ujson as json
import collections
from datetime import datetime, timedelta
from dateutil import parser

tokenizer = BertWordPieceTokenizer("bert-base-uncased.txt", lowercase=True)

class Articles(torch.utils.data.Dataset):
    def __init__(self, json_file, index_file=None):
        super().__init__()
        with open(json_file, "r") as data_file:
            self.examples = json.loads(data_file.read())

        if index_file is not None:
            with open(index_file, "r") as file:
                indices = [int(index.rstrip()) for index in file.readlines()]
            self.examples = [self.examples[i] for i in indices]

    def __getitem__(self, idx):
        return self.examples[idx]

    def __len__(self):
        return len(self.examples)

    def create_positive_sampler(self, target_publication):
        prob = np.zeros(len(self))
        for idx, example in enumerate(self.examples):
            if example["model_publication"] == target_publication:
                prob[idx] = 1
        return torch.utils.data.WeightedRandomSampler(
            weights=prob, num_samples=len(self), replacement=True
        )

    def create_negative_sampler(self, target_publication):
        prob = np.zeros(len(self))
```

```
        for idx, example in enumerate(self.examples):
            if example["model_publication"] != target_publication:
                prob[idx] = 1
        return torch.utils.data.WeightedRandomSampler(
            weights=prob, num_samples=len(self), replacement=True
        )

    def map_items(
        self,
        tokenizer,
        url_to_id,
        publication_to_id,
        filter=False,
        min_length=0,
        day_range=None,
    ):
        min_length_articles = []
        for idx, example in enumerate(self.examples):
            encoded = tokenizer.encode(example["text"],
                                       add_special_tokens=False).ids
            self.examples[idx]["text"] = encoded
            self.examples[idx]["url"] = url_to_id.get(
                example["url"], url_to_id.get("miscellaneous")
            )
            self.examples[idx]["model_publication"] = publication_to_id.get(
                example["model_publication"],
                publication_to_id.get("miscellaneous")
            )
            if filter:
                if day_range is not None:
                    dated = parser.parse(example["date"])
                    now = datetime.now()
                    last_month = now - timedelta(days=day_range)
                    if (
                        len(self.examples[idx]["text"]) > min_length
                        and last_month <= dated <= now
                    ):
                        min_length_articles.append(self.examples[idx])
```

```
            else:
                if len(self.examples[idx]["text"]) > min_length:
                    min_length_articles.append(self.examples[idx])
    return min_length_articles
```

E Data Distribution

The scraped data was converted into train, test, validation sets through a 70%, 15%, 15% split. The fake-news corpus was then added only to the validation and test sets, making them much larger and more negative heavy.

Data Distribution			
Dataset	Total Length	# of Positive Examples	# of Negative Examples
Train	100797	18598	82199
Validation	272447	4039	268348
Test	272448	4049	268399

Table 2: Dataset Breakdowns for training, test, and validation sets.

F Training Scripts

A collection of some of the most important training scripts that were utilized when performing the experiments.

F.1 Evenly-Split Batch Generator

For improved stability of the training process and because of the skewed nature of the datasets, I chose to feed in batches that had an even number of positive and negative labels. To do this, I created a subclass of the PyTorch Sampler [16].

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

# Create batches with even splits
# positive samples in first half and negative examples in second half
```

```python
class BatchSamplerWithNegativeSamples(torch.utils.data.Sampler):
    def __init__(self, pos_sampler, neg_sampler, batch_size, items):
        self._pos_sampler = pos_sampler
        self._neg_sampler = neg_sampler
        self._items = items
        assert (
            batch_size % 2 == 0
        ), "Batch size must be divisible by two for negative samples."
        self._batch_size = batch_size

    def __iter__(self):
        batch, neg_batch = [], []
        neg_sampler = iter(self._neg_sampler)
        for pos_idx in self._pos_sampler:
            batch.append(pos_idx)
            neg_idx = pos_idx
            # keep sampling until we get a true negative sample
            while self._items[neg_idx] == self._items[pos_idx]:
                try:
                    neg_idx = next(neg_sampler)
                except StopIteration:
                    neg_sampler = iter(self._neg_sampler)
                    neg_idx = next(neg_sampler)
            neg_batch.append(neg_idx)
            if len(batch) == self._batch_size // 2:
                batch.extend(neg_batch)
                yield batch
                batch, neg_batch = [], []
        return

    def __len__(self):
        return len(self._pos_sampler) // self._batch_size
```

F.2 Batched Predictions

These were the functions used to take an input batch of examples and generate predictions on them for both approaches. The device refers to a cpu or gpu depending on the hardware and target refers to the intended publication (which was

the quality-writing publication, or id 0, for the entirety of this experiment).

F.2.1 RankFromSets

```
@torch.no_grad()
def calculate_batched_predictions(batch, model, device, target=0):
    model.eval()
    (publications, articles, word_attributes,
    attribute_offsets, real_labels) = batch
    publication_set = [target] * len(real_labels)
    publication_set = torch.tensor(publication_set, dtype=torch.long)
    publication_set = publication_set.to(device)
    articles = articles.to(device)
    word_attributes = word_attributes.to(device)
    attribute_offsets = attribute_offsets.to(device)
    logits = model(publication_set, articles,
                                word_attributes, attribute_offsets)
    final_logits = logits.cpu().numpy()
    return final_logits
```

F.2.2 BERT

```
@torch.no_grad()
def calculate_batched_predictions(batch, model, device, target=0):
    model.eval()
    (word_attributes, attention_masks,
    word_subset_counts, real_labels) = batch
    word_attributes = word_attributes.to(device)
    attention_masks = attention_masks.to(device)
    logits = model(word_attributes, attention_mask=attention_masks)[0]
    final_logits = np.squeeze(logits.cpu().numpy())
    return final_logits
```

F.3 Recall Calculation

This was the loop within a function that was used to generate recall statistics for the model on a new dataset (validation or test) with backpropagationa and gradient calculations disabled.

```
@torch.no_grad()
def calculate_recall(
    dataset, indices, recall_value=1000,
    target_publication, version, writer, step,
):
    rev_indices = indices[::-1]
    correct_10 = 0
    correct_big = 0
    for i in range(recall_value):
        if dataset[rev_indices[i]]["model_publication"] == target_publication:
            if i < 10:
                correct_10 += 1
            correct_big += 1
    print(f"{version} Performance: Step - {step}")
    print(f"Top 10: {correct_10*10} %")
    print(
        f"Top {str(recall_value)}: {(correct_big*100)/recall_value} %"
    )
    print("--------------------")
    writer.add_scalar(f"{version}/Top-10", correct_10, step)
    writer.add_scalar(f"{version}/Top-{recall_value}", correct_big, step)
    return correct_big, correct_10
```

F.4 Generate Rankings List

This was used to create a Pandas DataFrame that ranked results on a new dataset and outputted information on the top 1500 articles.

```
def create_ranked_results_list(final_word_ids, sorted_preds, indices, data):
    df = pd.DataFrame(columns=["title", "url",
            "publication", "date", "prediction"])
    ranked_indices = indices[::-1]
    predictions = sorted_preds[::-1]
    for i in range(0, 1500):
        example = data[ranked_indices[i]]
        prediction = predictions[i]
        title = example["title"]
        url = example["link"]
```

```
        publication = example["publication"]
        date = example["date"]
        df.loc[i] = [title, url, publication, date, prediction]
    return df
```

G Permission Email

G.1 Email Sent to Mr. Uri Bram of The Browser

Hello Mr. Bram,

I am currently working on a machine-learning computer science paper for school. My goal is to compare models in their ability to predict "quality" writing. There is a general scarcity in terms of positive data in this space, and I was wondering if I could utilize your Browser archives. The data will not be published or utilized in any external manner.

Thanks in advance,
[redacted]

G.2 Permission Received from Mr. Uri Bram of The Browser

Dear [redacted],

Thank you so much for your kind email. It sounds like a great project and we would be happy for you to use the Browser archive. Please let me know if I can be of any further assistance.

Best,

Uri Bram CEO, The Browser Ltd