University of Southern Denmark

Department of Mathematics and Computer Science

SPDM801: Master's thesis in Computer Science

Creating a Pre-Trained Transformer Specific for News Articles

Author

Andreas Thomsen
antho17@student.sdu.dk

Supervisor

Konrad Krawczyk
konradk@imada.sdu.dk

July 22, 2023

Acknowledgements

I would like to acknowledge Konrad Krawczyk for supervising this project, aiming it in the right direction, and providing the training data used throughout the project. I would also like to acknowledge Hugging Face for providing a vast collection of free libraries, tutorials and forum posts, which eased the workload of pre-training multiple language models.

Abstract

Natural language processing (NLP) becomes increasingly relevant as recent language models become advanced enough to be useful to the average user as is the case with OpenAI's *ChatGPT* and other similar systems. The recent surge in language model applications increases the relevance in shining light on the applied technologies, with the aim of this project being to analyse the effect of domain-specific data used in the training of Transformer models. More specifically, investigating how training a Transformer model using only news articles, affects the performance of the model both in the context of transfer learning and text generation. For transfer learning, using the Hugging Face libraries, a Transformer model, with an architecture similar to that of RoBERTa, was pre-trained using only news articles. The result was a model which was almost capable of performing on par with the generically trained RoBERTa model, when both models were evaluated on their performance on news-specific, down-stream NLP tasks. The news-specific model was able to perform as it did, despite being pre-trained on, comparatively, very little data.

Using text analysis methods like term frequency and sentiment analysis, it also became apparent that the data used to train the generative Transformer model (GPT-2) impacted the generated output of the model, with the news-trained model being noticeably more negative in sentiment than the model pre-trained using generic data. The main takeaway from these results is that a Transformer model might not need excessive pre-training, if its use case is within a specific data domain. The second takeaway might not be very surprising, but it underlines the fact, that a bias is attached to a corpus, which easily affects a language model.

Contents

1 Introduction

1.1 Motivation - Why do text analysis using machines?

Analysing large collections of text can provide quantitative insight into topics, adding a foundation to hypotheses, which might otherwise have remained speculation. Such analyses could include measuring the sentiment with which COVID-19 is mentioned in news [6] or in which ways a viral pandemic might alter the news coverage of vaccines [5]. It may also be possible to find a correlation between what is being written in a certain domain and how people are behaving, and then potentially act on that information. (Think news coverage vs. election results.)

When performing quantitative text analyses similar to the ones referenced above, the sample size matters. 11 tweets do not provide a sufficiently broad representation for a generalizing claim such as *"the 10 most popular swear words used online in 2022"*, when 6000 tweets are shared every second [61]. In a case of quantitative analysis, more data, providing a broader representation of the domain, is better. However, a human cannot, in any useful time frame, manually analyse 6000 tweets, let alone millions or billions, which is why machines are deployed. In short, machines provide the means to perform a quantitative text analysis on a scale which would be impossible without them.

The methods used for text analysis, while all doing the job they were designed to do, vary in levels of complexity. One approach could simply be counting "words" (defined as the group of letters between spaces), while another approach could involve teaching a deep neural network how to classify sequences as being either "negative" or "positive" in their sentiment.

Rather than analysing text for the sake of uncovering any data it may hide, this project mostly aims to provide an analysis of one of the text analysis methods themselves, by measuring how the type of training data used to pre-train a Transformer model might affect its generated text output and its performance on down-stream Natural Language Processing (NLP) tasks. Specifically, the project attempts to determine how pre-training a language model, using samples from the news domain, might affect its performance compared to a model, which was trained using a more generic corpus, not limited to news.

1.2 The Transformer Architecture

The Transformer [2] is a powerful deep neural network architecture for language modeling, and especially its ability to be trained efficiently with parallel processes makes it a fitting architecture to pre-train using millions or billions of model parameters and potentially billions of training samples.

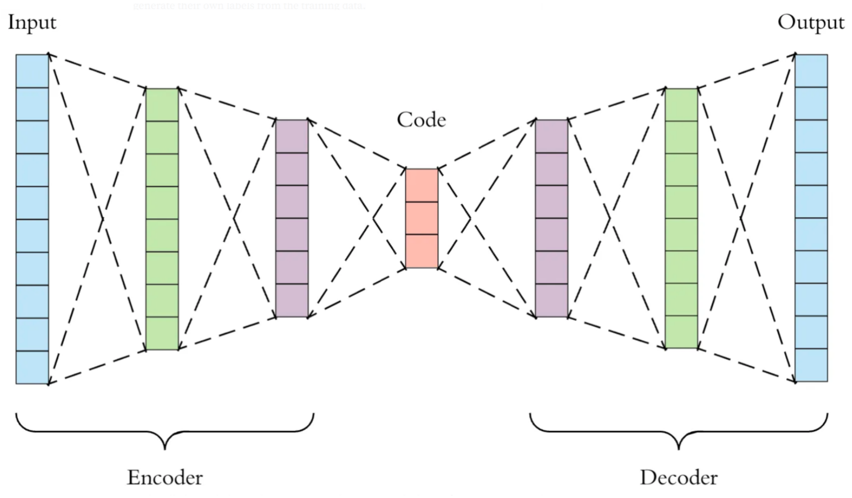


Figure 1: A general overview of an Autoencoder architecture. Source: [14]

1.2.1 Encoder-Decoder Models and Generative Methods

A fully utilized Transformer architecture follows an encoder-decoder structure.

Encoder An encoder takes an input, which could be a text sequence or an image, and produces an encoded representation of the input [64]. The encoder's output vector can, but does not have to [2], be a lot smaller than the initial input. An encoder by itself can be used to create sequence embeddings, which are numeric representations of text, that can be meaningfully compared in a vector space [43]. BERT [4], for instance, is an encoder-only Transformer, which by itself produces contextual sequence embeddings to be used in more specific down-stream NLP tasks.

Decoder A decoder can be considered the opposite of the encoder as it generates an output based on a smaller input. The GPT models are autoregressive decoder-only Transformers [41], meaning they can generate sequences by starting with very little to no input and gradually expand the output by passing the output from step *i-1* to step *i*. *"I"* which becomes *"I am"* becomes *"I am hungry"*, and so on.

Encoder-Decoder An encoder can be combined with a decoder to create an Encoder-Decoder model, where the output of the encoder is passed to a decoder. An example of such a model is the Autoencoder, where the initial input, such as an image, is passed through the encoder, with the output being a smaller representation of the input in "latent space". The encoded representation is immediately passed to a decoder, which produces an output of the same size as the initial input of the encoder. Figure 1 shows a general overview of an Autoencoder. This model architecture has several applications, one of them being denoising [64], in which case the Autoencoder would receive noisy images as training data, where the target output would be the same image without the noise. The goal is that the encoder produces a "latent representation" of the input image, which does not include the noise, so when the decoder reproduces the image, it has generated the input image without noise. Another application of the Autoencoder is lossy compression [8]. The idea is similar to denoising, but with the expectation that the encoder produces a representation that captures as much of the original input image as possible. Given that the encoder's output is smaller than the input, the encoded representation serves as a compressed version of the input image, which can then be reproduced (to a degree, hence the "lossy") by the decoder.

The Autoencoder is one example of an encoder-decoder structure, another is the Transformer as it was originally described in [2]. A Transformer that utilizes both its encoder and its decoder is also called a sequence-to-sequence model, and one of its applications is language translation, which is described in further detail in section 1.2.3.

Generative Methods - CLM vs GAN The word "generative", in the context of deep learning, might once primarily have meant Generative Adversarial Network (GAN) [11], but as already hinted at, the Transformer achieves generative properties differently. Figure 2 shows the structure of a GAN, which consists of two models. Firstly, there is the generator part of the model, which generates an example, such as a an image or a text sequence, given a random input vector. The generator model is trained by introducing an opponent or an "adversary" to the network called the "discriminator". The job of the discriminator is to consider a generated example along with a real example, and then predict which of them is the generated example. If it predicts correctly, it is "rewarded" while the generator is "punished" and vice versa. As the discriminator gets better at predicting, the generator will need to improve in order to "fool" it, and as the generator starts getting better at fooling the discriminator, the discriminator will likewise need to improve. This "battle" continues until the generator can generate examples that look real.

A Transformer, on the other hand, (if the goal is to generate text) practically learns a language by using a training objective called language modeling. Specifically, Causal Language Modeling (CLM) is a well-suited training objective, if the goal is to generate text. The word "causal" is the key. In CLM the goal is to predict the next token based only on the words leading up to it, because that is how a sentence is generally constructed during a conversation. Yes, the speaker has a plan of what to say, but the sentence is nonetheless produced one word at a time. Figure 3a illustrates an example of CLM. In the exampe, the goal is to predict the fourth word, therefore the word, along with all potential subsequent words, has been masked to hide it from the model, forcing the model to predict the word based only on the three preceding words. A loss will be calculated based on the prediction, and the model optimized accordingly.

CLM is not the only type of language modeling. Masked Language Modeling (MLM),

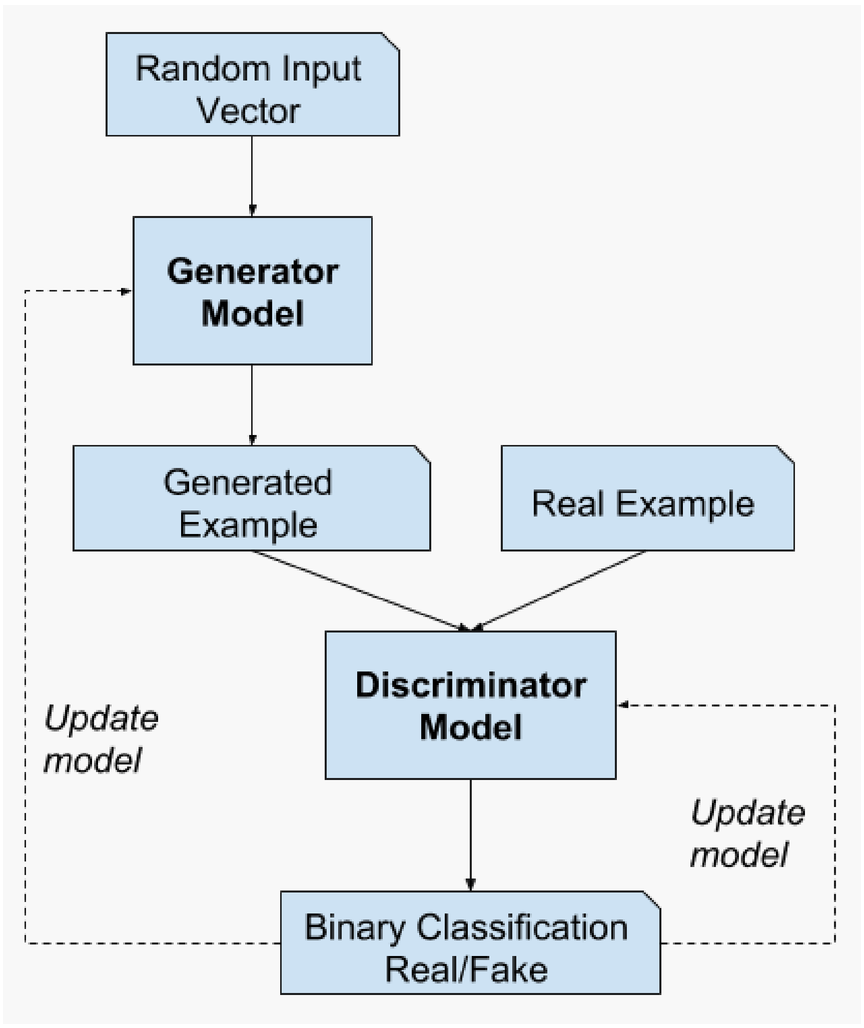
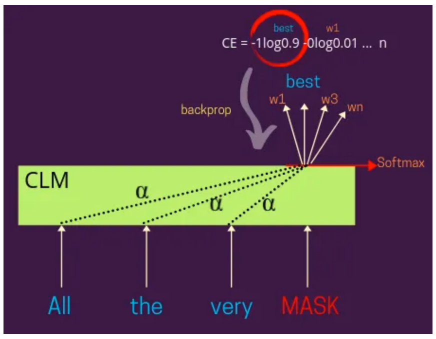
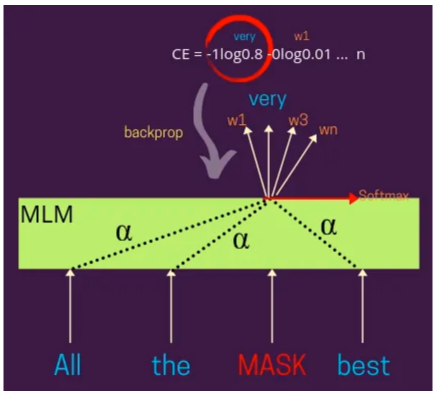


Figure 2: A general overview of a GAN architecture. Source: [11]



(a) CLM



(b) MLM

Figure 3: Illustrations of CLM and MLM with loss. Source: [46]

illustrated in figure 3b, works very similarly to CLM, with the exception that it also considers words beyond the masked token when predicting it. This exception is why MLM is not suited for text generation but luckily it has other applications which are covered in section 1.3.2.

At this point, it has been mentioned multiple times that these language models can predict words based on other words. In practice, these words are really numbers, because, in order to be used in a calculation, they have to be, and in the case of Transformers, these numbers have been heavily manipulated by several embedding layers, not the least of which are the Attention layers.

1.2.2 Attention

While the original Transformer paper, *"Attention Is All You Need"* [2], did not invent the attention mechanism, they did introduce the idea that convolution and recurrence is unnecessary when attention is sufficiently deployed.

The Concept of Self-Attention The idea behind self-attention is to provide context by comparing a sequence to itself, and in doing so, discovering which parts of the sequence deserves the most attention. To achieve an intuitive understanding of the concept, it can help to start with a concrete example while sticking to the simplest implementation of the concept [60]. An example sequence could be: *"The prince enjoyed riding his horse, and he feared the queen."* For this small example, each word from the sequence is represented by a primitive embedding vector, where each vector element is a non-negative number representing how the token scores within a concrete attribute such as "sour" or "food". While this embedding approach may be a bit more simple than what is applied in practice [2], assuming the scores are chosen carefully, it still achieves the desired result, which is that semantically similar words (or "tokens") have embedding vectors with similar values. This simple embedding approach has the benefit of making it easier to imagine how the semantic similarity of tokens can be carried over to a vector space. For this example, the embedding vectors have length 3 and are structured in such a way, that the first element is a score for *royalty*, the second element for *mount* and the last element for *masculinity*. The embedding vectors of a subset of the tokens might then look like the ones in Table 1.

In this case, self-attention is applied simply by calculating the "dot product" between the embedding vectors of all the token pairs. Below are some of the calculated dot products for the "prince" token:

$$prince \cdot prince = 1.8 \cdot 1.8 + 0 \cdot 0 + 2 \cdot 2 = 7.24$$

$$prince \cdot his = 1.8 \cdot 0 + 0 \cdot 0 + 2 \cdot 2 = 4.00$$

$$prince \cdot queen = 1.8 \cdot 1.95 + 0 \cdot 0 + 2 \cdot 0 = 3.51$$

Token	**Vector**
	[roy mou mas]
prince	[1.80 0.00 2.00]
queen	[1.95 0.00 0.00]
horse	[0.05 2.00 1.00]
his	[0.00 0.00 2.00]
the	[0.00 0.00 0.00]

Table 1: Simple embedding vectors for a few example tokens.

$$prince \cdot the = 1.8 \cdot 0 + 0 \cdot 0 + 2 \cdot 0 = 0.00$$

As demonstrated, the dot products between semantically similar tokens produces an amplified score, while the dot products between dissimilar tokens produces the opposite. The results of the dot products between one sequence token and the rest of the tokens can be referred to as "weights", and would be normalized to have a sum of 1. Lastly, the initial embedding vector of each token would be multiplied by the normalized weight of the corresponding position, effectively weighing all tokens towards how they relate to the current token. In the case where "*prince*" is the current token, "*his*" would be multiplied by the normalized value of 4 (a positive value), thus empowering the fact that "*his*" is contextually relevant to "*prince*". Similarly, "*queen*" would be multiplied by a large weight, despite being on the opposite side of the sequence compared to "*prince*", while "*the*" would be multiplied by 0. This process would be applied to each token in the sequence, such that every token has been weighed towards how it relates to all other tokens, leading to the result that the token with the most similarity between all other tokens, would get the highest value, and thus "receive the most attention".

As already mentioned, the above example is a simplified implementation, demonstrating only the very basic principle behind the concept of self-attention. Rather than use a fixed mechanism by relying only on the dot product between pairs of vector embeddings, trainable matrices can be included in the calculation, as well as multiple levels of attention layers.

Making Attention Layers Trainable In practice [2], self-attention is implemented by adding the concept of *keys*, *values* and *query*, all of which are vectors derived from the initial embedding vectors of input tokens. The output of an attention layer is the weighted sum of the values, where the weight for each value is a result of a matrix multiplication between the current query and the corresponding key of the value. Figure 4 illustrates how a weight can be computed and applied to a value using query and key.

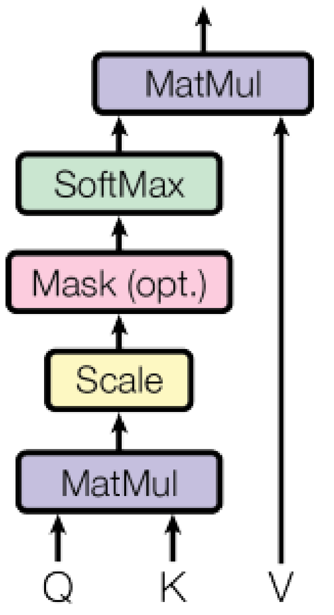


Figure 4: Scaled Dot Product Attention. Source: [2]

Every token in a sequence will be used as query and compared to every key in the sequence, with the resulting weights to be applied to the keys' corresponding values. The resulting sum of those values is the new embedding vector for the token that was used as query in the given calculation. The final result is new embedding vectors for all tokens, all of which differ from the initial embedding vectors by the alterations made as a result of their similarity (or dissimilarity) to every other token, including themselves. Figure 5 illustrates how a query, derived from one input token's embedding vector, is applied to all other input tokens and eventually results in a new vector.

Something figure 5 does not explicitly illustrate, although it is implied, are the components which make the layer trainable. In the example illustrated by figure 5, *input #1* is multiplied by a 3x4 matrix, which is what results in the *key* vector having different size and values compared to the initial input token. Such a matrix is applied to the input to calculate key, value and query for every token, and most importantly the parameters in these matrices are the part of the attention layer that is trained during every optimization step.

Multi-Head Attention The attention layers illustrated in figure 7 are called "Multi-Head Attention" layers, which simply means that multiple attention layers are applied in parallel, with the hope that different heads of the Multi-Head Attention layer will be trained to focus the attention on different parts of the sequence or in other words *"jointly attend to information from different representation subspaces at different positions"* [2].

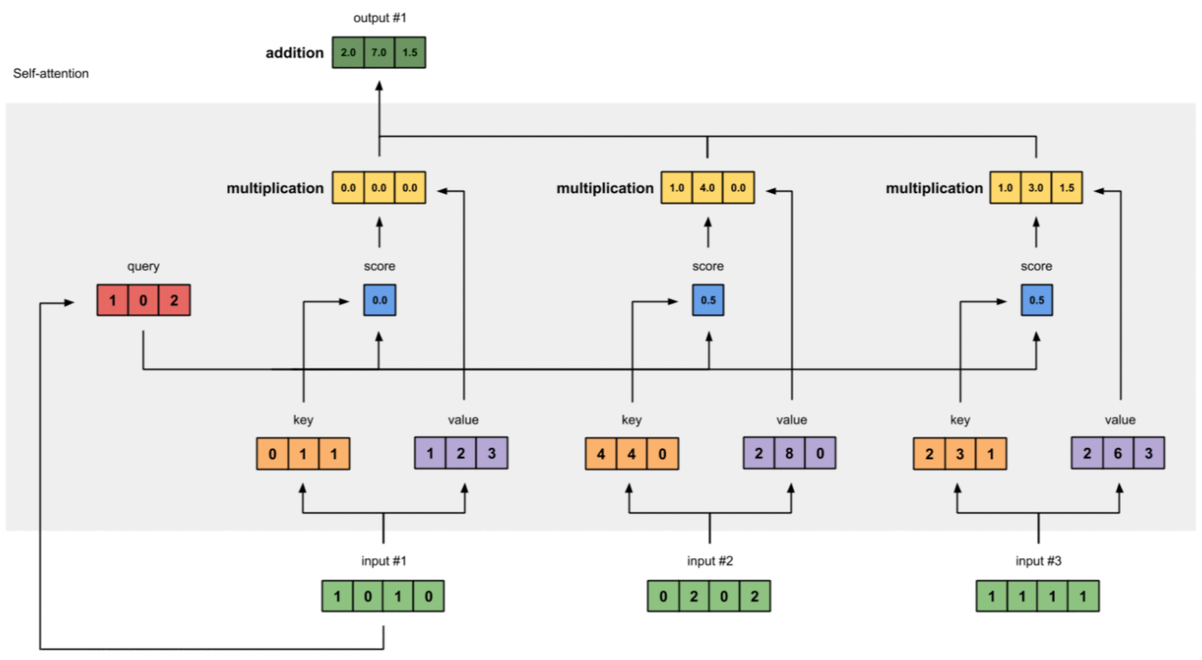


Figure 5: Detailed illustration of a 3-token sequence example of the attention mechanism using query, keys and values. Source: [44]

Figure 6 illustrates this process, showing how the vectors produced by the h attention layers are concatenated into a single matrix.

Positional Encoding Focusing on attention layers only, and thereby, for instance, removing recurrent layers, will rob the architecture of an important contextual property, which is the relative position of tokens. This issue can be solved by explicitly including a positional encoding layer between the embedding layer and the decoder or encoder stack. [2]

One option for the implementation of such a layer is a fixed approach [12]. [2] describes the use of sine and cosine functions of different frequencies for the positional encoding:

$$PE(pos, 2i) = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \tag{1}$$

$$PE(pos, 2i + 1) = cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \tag{2}$$

where pos is the position of a token and i is the dimension. The hypothesis was that the model could *"learn to attend by relative positions"*, because for every offset k, PE_{pos+k} could be represented as a function of PE_{pos}. However, a non-fixed, completely learned approach is also possible and achieves almost identical results. [2]

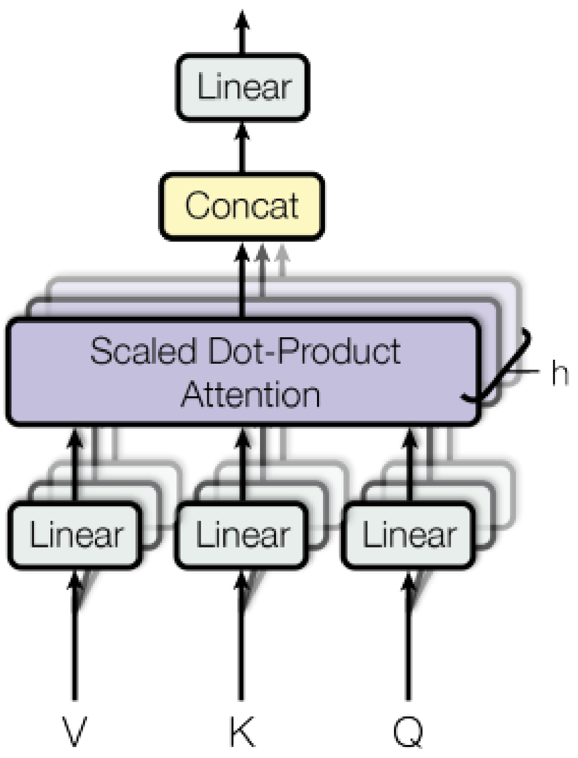


Figure 6: Illustration of multi-head attention. Source: [2]

1.2.3 Combining Concepts

With an understanding of encoders, decoders, attention, and positional encoding, figure 7, illustrating the Transformer architecture, is easier to understand. The architecture can be broken up into two parts, the left stack being the encoder and the right stack being the decoder. Language translation as a general example works well to describe the full capabilities of a Transformer with both the encoder and the decoder stack active. In the case of translating a French sequence into English, the full French sequence would (after the initial tokenization) go through the *"Input Embedding"* layers before the encoder stack. These layers can either be trained along with the rest of the model or they can be added as fully capable embedding layers from the start. In any case, the purpose of the embedding layers are to convert the sequence tokens of semantically similar meaning into vectors with numerically similar values.

Next, positional encoding is applied to the embedded input, and the "final initial embedding" is passed to the encoder stack, where it goes through the *Multi-Head Attention* block and continues through regular *Feed Forward* layers. The result of the encoder is an encoded representation of the initial French sequence, which hopefully captures the essence of the sequence very well, once the model parameters, including the attention layers, have been thoroughly trained.

Concurrently, during training, the decoder model would receive a "target" or "output"

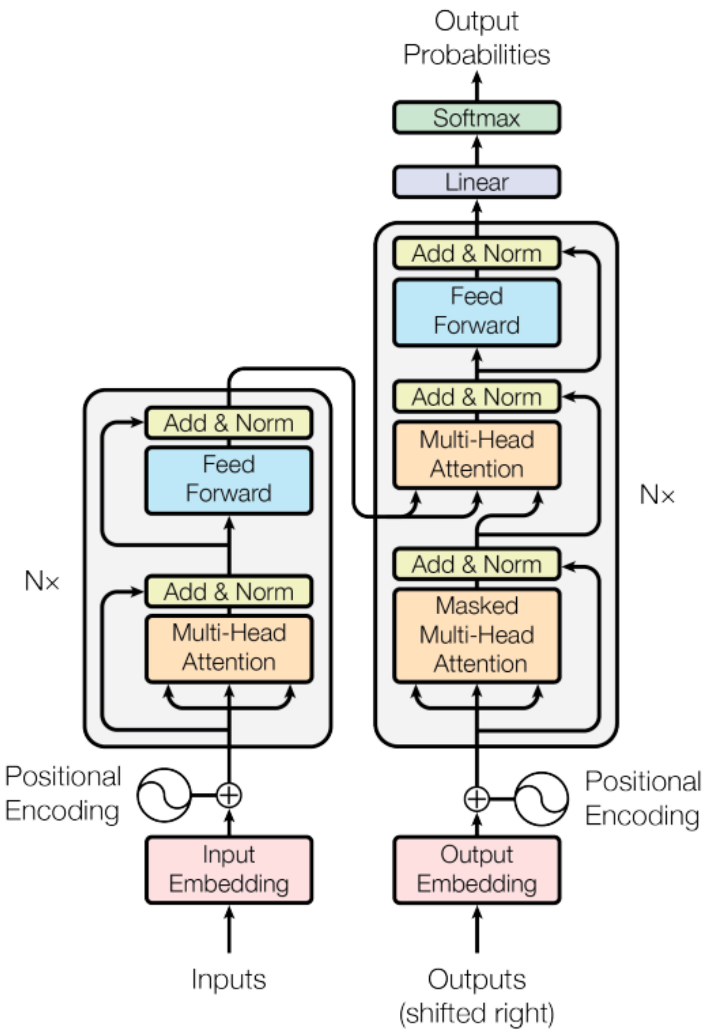


Figure 7: Transformer Architecture. Source: [2]

sequence, which would be the correctly translated English sequence. The output sequence would be shifted one position to the right, which is one of the steps taken to avoid a situation where the decoder knows the token it is trying to predict. The shift is usually implemented by including a special *"beginning of sentence"* token at the beginning of the sequence. This tokenized and shifted sequence would go through a procedure similar to the input, where it is passed through embedding and positional encoding layers, before reaching the decoder stack. Rather than a regular *Multi-Head Attention* block, the shifted output is passed to a *Masked Multi-Head Attention* block, which is the second step that ensures that the prediction of position i only depends on the outputs of positions $< i$.

The next block of the decoder stack is a regular *Multi-Head Attention* block, but in addition to receiving the output produced by the decoder stack's previous attention block, it also receives the final output of the encoder. The idea is that, although the decoder is only allowed to be aware of the preceding tokens of the output sequence, it is allowed to know the "essence" of the entire input sequence, which hopefully is encoded in the output of the encoder. The desired result is that the decoder provides the model with the means to generate grammatically correct sequences, within its target language, one token at a time, while the encoder nudges the decoder to produce a sequence which encapsulates an essence similar to the input sequence.

A benefit of this type of translation is that the syntax of the two languages are irrelevant. The model does not produce a "direct translation", which is good, because an English sentence might need a few more words, or more likely, present the words in a different order, to convey the same information as a French sentence.

1.2.4 Training Performance

To understand the leap in training performance made by the Transformer architecture, especially when applied to sequential input data, it is necessary to be familiar with the general concept behind the previously leading architectures such as Recurrent Neural Networks (RNNs). RNNs introduce the concept of "memory" to the model, which works well with sequential input such as text, because it allows the input at step i to be taken into consideration for the input at step $i+k$[13]. In other words, the weights applied to word number 3 are altered by word number 1 and 2, thereby adding context. Figure 8 illustrates the concept. One of the limitations of this approach to adding context, is that training is slow due to the recurrent nature of RNNs. The input word at position $i+1$ (x_{t+1} in figure 8) cannot be passed through the layer, until the input word at position i (x_t) has been passed through, an so on. This leads to a situation where RNN training cannot be parallelised optimally by a GPU, which is a limitation not shared by the Transformer. This means that Transformers can be trained much faster than RNNs, making it possible for the models to be a lot bigger, and be trained using a lot more samples, while still providing context via the attention layers. The result is that Transformers produce state of the art

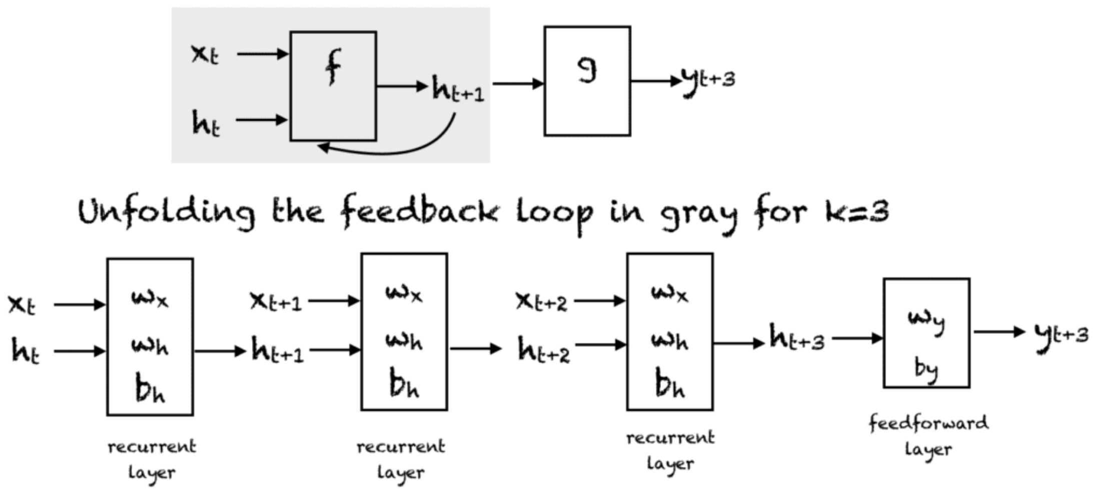


Figure 8: RNN, compressed and unfolded representation. Source: [13]

performance on NLP tasks [2].

1.3 Transformer Applications

1.3.1 ChatGPT

The Transformer has applications besides translation, and, in terms of text generation, a rather popular application is OpenAI's chat bot called "ChatGPT", which OpenAI refers to as a "sibling model" to InstuctGPT [51]. InstructGPT [50] is a model, based on the huge GPT-3 model, which has been fine-tuned to follow instructions rather than simply generating text by continuing autoregressively from a simple prompt. OpenAI are a bit secretive about the architectural details of their models, but ChatGPT has been trained in a way to make it capable of participating in meaningful dialogue, allowing it to accurately perform complicated tasks such as answering questions, translating various languages or creating regular expressions. The GPT-3 model consists of about 175 billion trainable parameters, but alternate versions of InstructGPT were created using only 1.3 billion parameters, although still with the weights of the pre-trained GPT-3 as its base [52]. Surprisingly, the InstructGPT version with 1.3 billion parameters reportedly produced the best responses. For reference, the models trained for this project contain approximately 124 million (not billion) parameters. ChatGPT was specialized to engage in dialogue, such that earlier parts of a chat conversation are taken into account when generating the next response. Correcting one of its earlier responses, for instance, will lead to a meaningful response, where the bot will either admit its mistake or insist on the correctness of its original response.

1.3.2 Transfer Learning

The various GPT models use CLM, because it is a language modeling technique well-suited for generating text, but as mentioned in section 1.2.1 there is an alternative form of language modeling called masked language modeling (MLM). While MLM is not as good for generating text as CLM, and it is not very useful by itself, it is excellent as the base for transfer learning. The BERT paper [4] describes the limitations of a unidirectional approach to language modeling, and introduces "Bidirectional Encoder Representations" by using masked language modeling. The resulting pre-trained language model performs well as the base for a broad set of NLP tasks. Essentially, the language learned by the model is transferred to a downstream task, such as Named Entity Recognition (NER) or sequence classification, resulting in better performances, compared to the case where there is no pre-trained model working as a "base".

1.4 Hypothesis - A Domain-Specific Model

The purpose of this project is to measure the effect of pre-training a Transformer using a specific data-domain. This mainly consists of pre-training a BERT-like[4] Transformer using the SciRide[6][7] corpus, which contains 25+ million news articles consisting of a headline and a sub-header. The goal is to test the hypothesis that: "*a language model pre-trained using data from a specific domain (news), will perform better than a language model pre-trained using a more generic corpus, when both models are fine-tuned and evaluated on a domain-specific NLP task.*"

The specific Transformer model architecture used for this project is the one introduced with BERT (Bidirectional Encoder Representations from Transformers) but using as many of the adjustments introduced with RoBERTa (Robustly Optimized BERT Pre-training Approach) [10] as possible. A second model architecture, GPT-2 [20], will also be trained with the goal of generating news sequences. The GPT-2 model will not be evaluated like RoBERTa, which is evaluated by its performance on down-stream tasks, but rather by using the model to generate a collection of text, and then basing the evaluation on said text. This includes subjectively determining if the resulting text sequences have a "news cadence", as well as performing simple quantitative text analysis to try to determine how the bias most likely embedded in news articles might reveal itself in the news-specific model compared to a more generic model.

2 Methods

The open source Hugging Face (HF) [39] Python libraries were heavily depended upon throughout all parts of the implementation. HF provides easy access to Transformer models, with various model heads for specific down-stream tasks, and the means to easily train them using the Trainer class [25]. The Trainer class implements training loops for models created using the PyTorch machine learning framework [58]. Models created using other machine learning frameworks, such as TensorFlow, are also available on HF, but for this project, PyTorch is the foundation for all models.

HF also provides the Dataset class [23] which is backed by Apache Arrow tables, allowing for fast access to large Datasets, without having to load everything into memory. The functionality of the Dataset class includes filtering, mapping, shuffling and selecting subsets of the Dataset.

Besides the mentioned Python classes, HF also provides the means to freely share the models and Datasets created by users via the HF Hub [24].

The source code implemented during the project is available on GitHub and can be found at [66].

2.1 Data Preparation

All implemented functionality used in the data preparation process mentioned in this section can be found in the *data_preparation.py* file of the source code.

2.1.1 Extraction

Extracing The very first step of preparing the data for training was to collect the data from the SciRide News Mine [7]. SciRide's *processed.tar.gz*-file yields a *release* directory once unpacked, which contains all the front-page news data in the following directory format: *release/{outlet}/per_day/{date}.gz.*

The unpacked *{date}.gz*-file contains all the collected news data for a given date, structured in a JSON-format like the following snippet:

```
{
    article_id:
        {
            "title": str,
            "description": str,
            ...,
            "is_covid": bool
            "description_stem": obj,
        },
    ...
}
```

The *title* is a news headline and the description is the associated sub-header. Each *article_id* is a unique string (per day), created by hashing "*title* + *description*".

The very first step was to remove the outlets, which published articles in languages other than English. Next, the extraction process consisted of traversing the *release* directory, and along the way collecting only the data needed for the rest of the project. This included *title*, *description* and *article_id*, which were collected directly from the JSON-files, as well as the date and outlet, which were collected from the names of the .gz-files. The extracted data was moved to JSONL-files, one for each outlet, which was formatted in a way to seamlessly allow for a HF function to load the data into a HF Dataset.

The JSONL-files were formatted like the following snippet, with an object on each line:

```
{"title": str, "description": str, "meta": {"article_id": str,
"outlet": str, "date": str}\newline
...
{...}\newline
```

Loading With extraction complete, the data was ready to be loaded into a HF Dataset, using the *load_dataset()* function. Initially, the data was loaded into a HF DatasetDict, which is a dictionary of Datasets usually used when the data is split up into train-test splits. It supports many of the same functions as the regular Dataset class, such as *.map()* or *.filter()*, where in those cases, the operations would be performed on one split at a time.

At this stage, the DatasetDict was used to hold a Dataset for each individual outlet, shown in the following snippet:

```
for outlet in english_outlets:
    # load articles from a single outlet into a dataset
    data_folder = DATA_FILES_JSON_DIR_NAME
    data_files = "%s/%s.jsonl" % (data_folder, outlet)
    outlet_dataset: Dataset = load_dataset("json", data_files=data_files)

    # add the outlet_dataset to the dataset dict
    datasets[outlet] = outlet_dataset["train"]
```

The HF Datasets hold the data in a row/column structure, resulting in data tables like table 2.

It is worth noting that the *article_ids* are not truly unique. Articles in the SciRide News Mine were collected from the front-pages of different outlets for all days between 2015-07-18 and 2020-10-17, but a truly breaking or relevant story can appear on the front page for multiple days in a row, resulting in duplicate titles and descriptions, and thus, as a result of hashing, duplicate IDs as well.

title	description	meta
string	*string*	*{"article_id": string, "outlet": string, "date": string}*

Table 2: A table showing the structure of the initial HF Dataset

2.1.2 Cleaning

Before moving on to labeling and training, a few improvements could be made to the Dataset in terms of "cleaning".

Removing Duplicate Rows Mainly as a result of articles appearing on front-pages for multiple days, the Dataset contains a large number of duplicate articles, which need to be removed. The HF Dataset does not provide a function to remove duplicate rows, however, the Pandas DataFrame [53], with its *.drop_duplicates()* function, does. Pandas and HF support conversion between Datasets and DataFrames, so a solution to removing duplicates from a HF Dataset could be to convert it to a Dataframe, where rows with the same title and description could be removed, and converting the resulting DataFrame back into a Dataset.

The problem with using a Pandas Dataframe, when handling large datasets, is that it holds all data in memory, while also using more GBs in RAM than it did on disk. The computer used for this project cannot handle that kind of load, so a few steps were taken to decrease the memory usage. One of those steps was, as mentioned, to initially split the data up into smaller portions, namely creating a Dataset for each outlet. The assumption was that duplicate articles, that are identical because they appear on the front-page for multiple days, would originate from the same outlet. Therefore, removing duplicates within each outlet would remove the majority of duplicate rows, and it did, removing about half of the rows, reducing the Dataset from 26,028,254 rows to 13,484,960 rows. Every other article being a duplicate might seem extreme but it is an average, which is probably increased by articles appearing on the front-pages for more than two days. To give an example, the article with *article_id "78d66dce4c0e768be39cb490e3f76e8e"*, was published on the front-page of *9news.com.au* on the dates 20170916-20170924, which resulted in nine duplicates.

In order to remove the remaining duplicates, the outlet splits would have to be merged into a single DataFrame to apply *.drop_duplicates()* on all rows at once. The remaining 13,484,960 rows were still too many to be loaded into a DataFrame, so the function for removing duplicates was optimized to use less memory. One step in the optimization process was to use the meta column's *article_id* as the "subset" parameter, instead of the *title* and *description* columns, when determining whether two articles were identical. Due to *article_id* being a direct result of *title* and *description*, two articles with the same *title* and *description* would also have the same *article_id*. This allowed for *title* and *description* to be absent from the DataFrame during duplicate removal, reducing memory usage, and now that removing data was an option, *article_id* became its own column, while the rest of the *meta* data was removed, leaving a DataFrame with only a single column. The resulting DataFrame was small enough to fit into memory, but a new problem arose, since the result of *.drop_duplicates()* was missing all the relevant columns. The solution was to add a second column to the DataFrame of *article_id*s before removing duplicates. The second column was an index column starting from 0, cast to a 32-bit *int* to reduce memory usage. When a row was removed due to duplicate *article_id*s, the index column would lose a row as well, and the indices would start skipping numbers. After all duplicate *article_ids* were removed, the index column would be passed to the Dataset's *.select()* function which selects a subset of rows based on a range or a specific list of numbers. *.select()* does not work on a DatasetDict, only on a Dataset, and there was no longer a good reason to have the data split up into outlets, so the DatasetDict was merged into one big Dataset before selecting the unique rows. The last parts of the duplicate removal process is shown in the following snippet.

```
pandas_df: DataFrame = pandas_df.drop_duplicates(subset=["article_id"])

# move the remaining indices to a list
indices = pandas_df["index"].values.tolist()
del pandas_df

# select only the rows associated with an index that was not associated
# with a duplicate id
nodup_dataset: Dataset = dataset.select(indices)
del indices
```

With that, the resulting Dataset would contain no duplicate rows, reducing it from 13,484,960 rows to 13,241,068 rows, but there were still a few things that could be done to remove duplicate articles. Some of the articles were tagged with a date, making it possible that multiple, otherwise identical, articles would appear unique only because of the date. The removal of "junk snippets" is explained in more detail in the next named paragraph, but in short, some dates were removed from the articles before removing duplicates the second time, resulting in a few more duplicates gone, reducing the Dataset from 13,241,068 rows to 13,233,494 rows.

Removing Junk Snippets The term "junk snippets" in this project refers to smaller parts of a *title* or *description*, which is not meant to be there. During cleaning, a snippet was considered "junk" if it did not contribute naturally to the text sequence. The removal of junk snippets was split up into two parts:

Firstly, there was the removal of general patterns such as dates or HTML.

A very common pattern of junk was to attach a date at the end of a sequence, usually of the MM-DD-YYYY format. These types of dates were preceded by a "|" symbol, and were thus caught with the following regular expression:

```
date_pattern = r'\|[^\|]*\d{2}[\.\-\/]\d{2}[\.\-\/]\d{4}[^\|]*'
```

This regular expression allows for any number of symbols (excluding "|") between "|" and the date, which was done to catch examples that included text besides the date, such as "Aired: ". Besides the general pattern for dates, a few custom examples were found, such as:

```
r'.*\d{1,} \(BusinessDesk\) -' # "July 23 (BusinessDesk) -"
```

Other examples of junk were the few HTML elements which were missed during the initial scraping. It was not possible to use a simple generic pattern like the following

```
r'<[^<>]*>'
```

to match the HTML elements, because it would also remove patterns like the one in "JPMorgan Chase <JPM.N>", which it should not. Libraries like Beautiful Soup [55] also matched on those patterns. Instead, the generic pattern was used to find all potential occurrences of HTML elements, and save them in a file. The file was then manually looked through to find all relevant HTML elements, which were then added to a more specific regular expression pattern:

```
html_pattern = r'<\/{0,1}(div|aside|span|table|tbody|thead|tr|td|li|ul|ol
|del|br|strong|i|b|u|p|a|h\d)(\s[^<>]*=[^<>]*>|\/{0,1}>)'
```

The HTML library [56] was used to unescape HTML characters such as "&rsquo;", by replacing them with their Unicode counterpart.

There is not always room for the entire description on a front-page, and a common way for some outlets to solve this problem, was to abruptly end the description using "...". This resulted in the emergence of "fake words", because the "..." would often appear in the middle of a word. This was solved by removing all symbols between the last space and the "...". The "..." were kept because they do indicate that a sentence was not completed.

Secondly, there was the removal of more specific snippets, which were found by combing through all the data and counting occurrences. It was very apparent, that two strings were often associated with junk: " - " and "|". The snippet: "| Daily Mail Online", for instance, appears at the end of 828,999 sequences. Generalizing and removing everything after the "|" character on every sequence is, however, not the way to go, since the character is not always used in the context of junk. Instead, every sequence was split up into snippets, using "|" as the delimiter, while keeping the delimiter as part of the snippet. The snippets were then added as keys to a Python dictionary, with the number of occurrences being the values. The resulting frequency map was sorted by value, saved and looked through manually to find the approximate count threshold, where the snippets started to become junk snippets. This process was performed for both " - " and "|". Every snippet, sorted by length, was then added to a regular expression with the "OR" character separating them. The snippets were sorted by length, because the *.sub()* function only replaces all non-overlapping matches. By making sure the longest snippets were removed first, the case is prevented, where a snippet, which happens to be a substring of another snippet, is removed, causing the longer snippet to no longer match, resulting in some of the junk remaining.

For instance, consider the junk snippets: "| outlet - DK" and " - DK", and the sequence: "Something happened | outlet - DK". Removing the " - DK" would result in the sequence: "Something happened | outlet", but now "| outlet - DK" is no longer a match, causing "| outlet" to remain.

Junk snippets were removed using the Python standard library for regular expressions *re*. Specifically, the *.sub()* function, which takes a pattern and a replacement snippet and substitutes all occurrences in a string, was used.

Dates, and other more general patterns, were removed after the first pass (within outlets) of duplicate removal, due to the fact that duplicate removal was a very fast process, and junk snippet removal was a very slow process. Reducing the Dataset by about 50% before removing junk snippets also cut cleaning time in half.

As mentioned, a second pass of duplicate removal was performed after removing dates, but because duplicate removal was based on the *article_id*, removing snippets from *title* and *description* and then trying to remove duplicates again would result in 0 duplicates removed. For this reason, a new column was added to the Dataset, called "*new_article_id*". The *article_id* with the *meta* data would remain the *article_id* from SciRide, making it possible to find the original article. The *new_article_id*, however, would be the result of hashing the cleaned *title* and *description*. This was done using the hashlib library [57] immediately after cleaning a row. The cleaning process removed $4,484 - 4,315 = 169$ MBs of data.

It needs to be mentioned that the cleaning was not perfect. It was later noticed that there are other, not as common, characters associated with junk, such as a longer version of the dash or a "double dash", which ultimately were not removed. It was also noticed, during manual labeling, that the data includes a special Unicode character for space (U+00A0), which will not be considered a space during tokenization. This was also not removed. Undoubtedly, other things were missed as well. Furthermore, the frequency threshold chosen to remove junk snippets was inevitably influenced by what the author considers junk, injecting a small bias into the data.

Removing Junk Rows A few articles were not cleanable and needed to be removed entirely. Some of the examples found when looking for HTML elements were among those cases. The "<iframe>" element, for instance, was used as an indicator, that a row needed to be removed, because it contained nothing more than a voting option for a poll. Besides HTML, the length of the combined *title* and *description* was also used as an indicator. The number 16 was, somewhat arbitrarily, selected as the threshold, such that only articles longer than that were kept. It turned out that anything of length 16 or less was mostly just names with no context, which is why it was chosen as the threshold, but it could have also been 15 or 19. The removal of junk rows reduced the Dataset from 13,233,494 rows to 13,219,867 rows.

As the last step, duplicates were removed one last time, in the event that the removal of junk snippets had caused new duplicates to appear. It reduced the Dataset from 13,219,867 to 13,118,041 rows. In total, the cleaning and filtering process reduced the Dataset by 12,910,213 rows, approximately 50%.

Pre-Processing the prepared data for training The last step in preparing the data for pre-training was packing the input samples to fit into the models. RoBERTa has a maximum input size of 512 input IDs. Not a lot of the articles actually exceed that limit, but a truncation strategy was necessary nonetheless. One of the considered options was to split the data up into natural sentences, and while it would not absolutely guarantee that all samples contained less than 512 IDs, it would reduce the probability. Potential edge cases could be solved by simply discarding input IDs from index 511 and up. The idea of splitting the articles up into sentences was discarded, because it would be a shame to rob the attention layers of an opportunity to live out their potential. An attention layer should be perfectly capable of providing context intelligently across all 512 tokens, so it should have the chance to do so. Instead, inputs were initially created by adding a period to the title and concatenating the description. This still leaves the truncation strategy. One option, that was initially implemented but ultimately not used, was to remove all ID's after index 511, but rather than discarding them, add them to a separate sample by utilizing the Hugging Face tokenizer's *"return_overflowing_tokens"* parameter. This would create some strange samples, which would start abruptly and not be very long, but no data would be lost.

Ultimately, for the final solution, inspiration was taken from the RoBERTa paper [10], where they describe the packing strategies "Full-Sentence" and "Doc-Sentence". With Full-Sentence, each input is packed with multiple natural sentences sampled contiguously from one of more documents, meaning that if the addition of the next sentence would exceed the limit, the entire sentence is moved to the beginning of the next sample, and if a document has no more sentences, but the input sample still has room, sentences from the next, potentially unrelated, document is added to the sample. Doc-Sentence is very similar, but does not allow for the "crossing of document boundaries". RoBERTa uses Full-Sentence but for this project Doc-Sentence was used, with the reason being, that the articles are usually very short, and if the crossing of document boundaries was allowed, a single input sample would more often than not contain data from multiple different documents. In their testing, the RoBERTa paper concludes that Doc-Sentence performs slightly better than Full-Sentence.

To determine whether a specific article exceeded 512 tokens, it was necessary to train the tokenizer first as explained in section 2.2.1.

So, in summary, the finalized Dataset used for pre-training was cleaned, then the tokenizer was trained to learn the vocabulary of the cleaned data, after which the *title* and *description* from the samples were split up into natural sentences, which were tokenized with the purpose of counting tokens. The tokenized sentences were grouped according to the Doc-Sentence rule, and finally "un-tokenized" to create a single article but with a guarantee, that each article from the resulting Dataset would not exceed 512 tokens, assuming it was tokenized with the appropriate tokenizer. The resulting Dataset had size 13118478, which means the process of repacking the data to avoid abrupt truncation slightly increased the amount of rows by 437.

Normalisation It deserves explicitly mentioning, that while the data is "cleaned", as described above, it is deliberately not normalized. Normalization, which could include removing stop words, converting to lowercase or stemming to reduce variation in similar words, makes a lot of sense for "simpler" text analysis methods such as term frequency, as it would be counter productive to count "dog" and "dogs" separately. But for a language model, which is developed with the purpose of "understanding" the intricacies of a natural language, reducing the language to a normalized one, does not make sense, especially if the goal is to generate text, as the resulting generated text would look something like: "bori johnson won elect" ("Boris Johnson has won the election.")

Access the Data The full, cleaned Dataset, which was used as the basis for training, has been uploaded to the Hugging Face Hub and can be found at [65].

2.1.3 Labeling

Section 2.1.2 describes the very general data preparation process of cleaning up the data, which is enough to begin the pre-training process using the Hugging Face libraries due to the language modeling tasks being self-supervised, however, for the fine-tuning tasks, which were supervised, a significant amount of preparation still remained as the data needed to be labeled according to the tasks.

Single-Label Sequence Classification The single-label sequence classification task, in this case, was a binary classification task, which aimed to predict whether an article referred to COVID-19 or not. COVID-19 was selected as the prediction task because it has a unique identifier, which makes it easy to label the training data automatically.

The data was labeled automatically by mapping over each (temporarily normalised) row in the Dataset and considering two cases: if the *title* or *description* contained the normalised token "covid-19", it would be labeled as 1 for, "*is_ covid*" and returned. If the date from the associated *meta* data was earlier than 2020-01-01, the sequence would be labeled 0 for "*is_ not_ covid*", and returned. Everything else would be discarded. The normalisation, which was only active during labeling, included converting to lower case and stemming using a Porter stemmer from NLTK. [48] The purpose of the case lowering was to include articles where an author might have decided to type "Covid-19" rather than "COVID-19". Most likely, the stemming ultimately did not affect the outcome, but the labeling function was implemented before COVID-19 was decided as the topic, with the goal of implementing a more generic behaviour. The original purpose of stemming was to reduce the necessary amount of keywords to specify, if the topic was something like "vaccination". In that case, rather than instructing the labeling function to look for keywords such as "vaccination", "vaccine", "vaccinating", etc. it would suffice to specify "vaccin".

Once the target size of the labeled Dataset was reached, the labeling process would stop, and the largest class would be reduced to the same size as the smallest class, guaranteeing an even distribution of positive and negative labels. When the labeled Dataset was split into train-test-validation splits, a "*stratify_ by_ column*" parameter was passed to the *train_ test_ split()* function to make sure that all splits contained 50% of each label.

Multi-Label Sequence Classification This project's multi-label sequence classification task was a multi-class classification task, which aimed to predict which politicians / world leaders, if any, a specific article referred to, with the "multi-label" specification meaning that a single article could refer to multiple politicians. Similarly to the single-label classification task, the topic of "politicians" was selected because name + surname makes for a good identifier when labeling data automatically. Table 3 lists the classes used.

An expanded list of the same politicians, containing lists of associated, unstemmed keywords, can be found in the appendix. The lists of unstemmed keywords were used to determine which tokens should be substituted during some variations of evaluation, explained further in section 3.1.1.

Complications The labeling process of the multi-label classification task was similar to the one for the single-label classification task, with the exception that the possibility of multiple labels complicates things. These complications caused the initial implementation to be slightly wrong. The wrong implementation is described as "initial" because the mistake was discovered and fixed, in which case it normally would not make sense to even mention it. In this case, however, the mistake was caught a bit late, and had already had a small impact on the outcome, which was not feasible to reverse. Therefore, the effects of the wrong implementation will be covered, while the finer implementation details, as to

Topic	Stemmed keywords
Donald Trump	["donald trump", "presid trump"]
Boris Johnson	["bori johnson"]
Hillary Clinton	["hillari clinton"]
Joe Biden	["joe biden", "joseph biden", "presid biden"]
Barack Obama	["barack obama", "presid obama"]
Bernie Sanders	["berni sander", "bernard sander"]
Angela Merkel	["angela merkel"]
Emmanuel Macron	["emmanuel macron", "presid macron"]
Vladimir Putin	["vladimir putin"]
Kim Jong-Un	["kim jong-un"]
Justin Trudeau	["justin trudeau"]

Table 3: A table showing the classes used for multi-label sequence classification

how it went wrong, will not.

The problem with the wrong labeling implementation, was that it did not correctly ensure an even distribution of classes. The most common classes would be represented more, despite measures to avoid that issue, and the class frequencies ended op as follows:

```
{
0: 15372, 2: 6834, 4: 6189, 1: 6183, 3: 6183, 8: 6095,
10: 5978, 7: 5976, 5: 5965, 9: 5963, 6: 5963
}
```

The distribution was mostly usable, but one class (0, Donald Trump) stuck out, which was also the most common class, by far, before taking measures to even out the distribution. The uneven distribution was caused by the overlap in labels. Class 0 would initially have contained 5963 samples, as was the goal. But then for class 2 to reach the target size, it might include articles, which also contained the label of class 0, including articles which class 0 did not have among its 5963, thus adding extra 0-labels to the total. In an extreme scenario, it might be that class 0 only chose articles with nothing but the 0-label. On the other hand, class 6 might have only chosen articles in which class 0 was also mentioned. In that case, there would be at least twice as many instances of class 0 compared to class 6.

The wrong implementation was used initially to label the data, and those labeled articles were then removed from the data meant for pre-training, after which the pre-training was run. It did not make sense to run pre-training again just because the pre-training Dataset would be slightly changed after using an improved labeling function. Instead, the new labeling function would just "re-label" the already labeled Dataset, such that no new articles were labeled, and the labeled data was still guaranteed to not be included in the pre-training data. This would cause a small subset of the initial labeled Dataset to be useless for both fine-tuning and pre-training but that was a small punishment compared to running pre-training again, which would take weeks. The following goes into detail about the implementation of the improved function.

Implementation The first step was to label the data without adjusting the distribution. In this case, the output of the "wrong" implementation was used but ideally, the distribution of classes should not have been manipulated at all, because the class frequencies would be used for the next part. Rather than a binary label for "true or false", a list was attached to each row containing all the relevant label_ids. The labeled Dataset aimed to contain approximately 0.5% of the full Dataset.

After the initial labeling phase came the "selection phase", which had the job of choosing among the labeled articles such that there was an equal amount of all classes at the end. The first step was to count the frequencies of labels among the labeled data and sort them in increasing order. The idea was that the least common class should be searched for first to make sure it had the least restrictions. The conditions which needed to be satisfied for a candidate article to be selected was be the following:

Condition 1 The candidate article must include the class label currently being searched for.

Condition 2 The candidate article must not cause any class to exceed the target frequency.

Condition 3 The candidate article must not have been selected before.

The labeled data was searched through once for every class, and for the first few classes, there would be no problem with an article having multiple labels. If an article had multiple labels, the dictionary keeping track of frequencies would just increase the count for the additional classes as well. But Condition 2 would become increasingly difficult to satisfy as the classes started to reach the target size, because if a candidate satisfying Condition 1 was found, but it also included a label of a second class, that second class would need to not have reached the target size yet. The last class to be searched for would have to reach the target size by only choosing articles with a single label. Therefore, the least common class should get the opportunity to reach the target size first, and the most common last. In practice, it did not seem to have much of an effect, in which order the classes were searched for, but starting with the least common class had the added benefit of including more multi-labeled articles compared to doing the opposite. The following snippet shows the frequency of articles with different amounts of labels, when starting from least common and most common classes:

```
starting_low  = {1: 31487 , 2: 6046 , 3: 414 , 4: 19 , 5: 1}
```

Type	**Description**	**Yes**	**No**
PERson	Name of a person (no titles).	Joe Biden	Mr. Joe Biden
ORGanisation	Name of an organisation.	EU	Europe
Geo-Political Entity	Name of a country, city, state.	Paris	Eiffel Tower

Table 4: A table showing an overview of the chosen named entity types

```
starting_high = {1: 34165, 2: 4981, 3: 242, 4: 11, 5: 1}
```

To specify the target amount of labeled articles, a percentage was passed to the labeling function to represent how large a fraction of the unevenly labeled data should be selected. In this case, 90% was the goal for the training data, but it would be impossible to reach an even distribution with 100% of the data, so 0.80 was used as a weight for the target fraction, and the target size of each class was calculated as:

$$c\_size = \frac{total\_rows \cdot (0.9 \cdot 0.80)}{num\_classes}$$

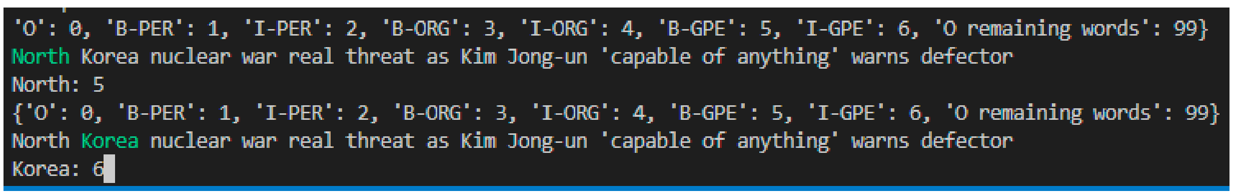
Using *"stratify_by_column"* is not supported when splitting multi-labeled data, so to keep an even distribution of classes when creating the train-test-validation splits, the procedure was run 3 times with a decreasing *c_size* and a collection of "off limits" *article_ids*. The collection of *article_ids* were added to make sure there was no overlap between the splits. The test-split and validation-split selected 9% and 1% of the labeled data, respectively, both of them with the same weight as the train-split used.

Token Classification (NER) For the token classification task, which was a Named Entity Recognition (NER) task, the goal was to fine-tune a model that could identify whether a token was a named entity. There are many types of named entities, an obvious example being the name of a person, and a more obscure one being something like names of weapons. The first step in labeling the data was therefore to decide which types would be considered named entities, which turned out to be "PER" for person, "ORG" for organization, and "GPE" for Geo-Political Entity. GPE is more specific than the more general type "LOC" for location. Table 4 shows an overview of the chosen types along with right and wrong examples.

Labeling data for NER is not possible to do well automatically, so it had to be done manually. To aid in this process, an interactive Python script was written, which was constructed as follows:

1. The already labeled data of politicians was used as the unlabeled data pool, because it was guaranteed to include at least one named entity.

2. Each unlabeled row of *title* and *description* was split up into natural sentences.



```
'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4, 'B-GPE': 5, 'I-GPE': 6, 'O remaining words': 99}
North Korea nuclear war real threat as Kim Jong-un 'capable of anything' warns defector
North: 5
{'O': 0, 'B-PER': 1, 'I-PER': 2, 'B-ORG': 3, 'I-ORG': 4, 'B-GPE': 5, 'I-GPE': 6, 'O remaining words': 99}
North Korea nuclear war real threat as Kim Jong-un 'capable of anything' warns defector
Korea: 6
```

Figure 9: The interactive NER labeling script in action.

3. Each unlabeled sentence was split up into words by separating on the space character.

4. The words were presented to the manual labeler (the user) in a terminal, one at a time, along with the whole sentence for context.

5. For each word, the labeler would type in an integer label corresponding to one of the named entity categories.

6. Each label, typed in by the labeler, would be added to a list at the index corresponding to the labeled word, eventually resulting in a "label list" of the same length as the list of words.

7. Each pair of words and labels, from each sentence, was saved in a .JSON-file, such that if a mistake was made during the interactive labeling, it was easy to fix it afterwards by editing the file. Eventually, the .JSON-file was used to create a HF Dataset, with each row being a natural sentence (split up into words), with the label column being a list of NE labels, where each item was associated with an item in the list of words.

Figure 9 shows the terminal during the interactive labeling process. The "B-" and "I-" attached to the NE types indicate whether the word is at the beginning or inside of the NE, so the words:

```
["Roronoa", "Zoro", "got", "lost", "on", "his", "way", "to", "Paris"]
```

would be associated with the NE types:

```
["B-PER", "I-PER", "O", "O", "O", "O", "O", "O", "B-GPE"]
```

which would be mapped to the integer labels:

```
[1, 2, 0, 0, 0, 0, 0, 0, 5]
```

A few additional pre-processing steps were necessary before the data could be used to train a model, but those steps would be considered part of the tokenization process and therefore part of the training process, so it is described in section 2.2.2.

Because the tokens were labeled manually by a human, the correctness of the labels is subject to human error. It should also be mentioned that the multi-labeled data of politicians, which was used as the basis for the NER data, was the initial labeling that had an uneven distribution of labels. This should no be an issue as the relevant part is whether a token is a named entity at all, not which named entity specifically. Other names, besides the chosen politicians, also appear in the sequences, and these would be few compared to any of the chosen politicians. An uneven distribution of names, in this case, was practically inevitable.

Title-Description Match (Classification) Inspired by next-sentence prediction, which was one of the training objectives used to train BERT [4], an additional fine-tuning task was added to the collection of evaluation metrics. The goal of the "title-description match" task was to predict whether a given *description* belongs to a given *title*. Essentially, it is a sequence classification task to recognize paraphrases, due to *title*s usually being a summary of the *description*.

The labeling process was very simple as it only involved moving descriptions from their original *title* to somewhere else. Specifically, the labeled COVID-19 data was used as the data pool, of which half was labeled 1 for "*does_match*" and the other half was used to create a derangement of *description*s such that no *description* remained at its original position while the *title*s did. The derangement was achieved by mapping over small batches of the *description*s and switching the first half with the second half. If the last batch happened to only contain one article, and therefore could not have the *description*'s position altered, it was simply discarded. The deranged half of the data was labeled 0 for "*does_not_match*".

As the COVID-19 data was already split into train-test-validation splits, the above process was performed on each split individually, rather than doing it on everything at once just to split and stratify again.

2.2 Training

The training portion of the project consists of two parts: the pre-training and the fine-tuning. The main goal of the pre-training was to create a language model, which would act as the base model for the subsequent fine-tuning tasks, while the only purpose of the fine-tuning tasks was to measure the performance of the pre-trained model. As the project evolved, three additional pre-trained models were added to the training plan along with the RoBERTa [33] model mentioned in section 1.4: Firstly, a generative model mimicking the architecture of GPT-2 [20], meant to find out how domain-specific training data would affect the resulting generated text. Secondly, a variant of each of the two previously mentioned models, which would be trained the same way as their counterparts, with the only difference being that they were not trained from scratch. So, the six base-models refereed to in this report would be:

roberta-gen which is just "*roberta-base*" from Hugging Face, a model pre-trained using non domain-specific (general) data.

roberta-news which is the RoBERTa architecture trained using news articles with randomly initialized weights as the starting point.

***roberta-gen*-news** which is the RoBERTa architecture trained using news articles with "*roberta-gen*" as its starting point, rather than randomly initialized weights.

GenGPT which is just "gpt2" from Hugging Face, the smallest version of GPT-2, trained using non domain-specific (general) data.

NewsGPT which is the GPT-2 architecture trained using news articles with randomly initialized weights as the starting point.

GenNewsGPT which is the GPT-2 architecture trained using news articles with "gpt2" as its starting point, rather than randomly initialized weights.

All implemented functionality used in the training process mentioned in this section can be found in the *language_modeling.py* and *fine_tuning.py* files of the source code.

2.2.1 Pre-training

If the evaluation of the pre-trained news-model were to hold any meaning, it would have to be compared to another pre-trained model, and for this task "*roberta-base*" [33] was chosen. Since the purpose of this project was to measure how the data, specifically, affects the performance of a model, it was important to reduce the variance in all other variables as much as possible, which is why the RoBERTa architecture was used as the base for *roberta-news*. During training, *roberta-news* would use the same type of tokenizer, optimizer and optimizer values as *roberta-base*, with the exact same learning objective and masking strategy. Even the amount of warm-up steps were approximately the same, although relative to the total amount of training steps. The only variables that differed were the data, the batch sizes and the learning rate. The same considerations were made for the generative models.

Avoiding Overlap in Pre-Training and Fine-Tuning Data The very first step in realizing the pre-trained models was making sure, that the data used for pre-training was not present in the labeled data used for fine-tuning. It could be considered cheating, if *roberta-news* had already seen the fine-tuning data before, while *roberta-gen* had not.

This was implemented by loading the labeled data for single-label classification task (COVID-19) and the multi-label classification task (politicians) and adding the *article_id*s to a dictionary. The structure of the data allows for the *article_id*s be to be moved from a list to a dictionary without any loss, as the data is not meant to contain any duplicate *article_ids*. There could be an overlap in COVID-19 data and the politician data, but only one would be included in the dictionary. The following snippet shows the process:

```
labeled_ids = {meta["article_id"]: None for meta in labeled_covid[:]["
                                    meta"] + labeled_multi[:]["meta"]}
partial_data = full_data.filter(lambda example: example["meta"]["
                                    article_id"] not in labeled_ids)
```

The *in* function calls the "private" *__contains__()* function of the data structure in question. Moving the data to a dictionary means that the *in* function makes use of the hashing used in dictionaries, which speeds up the filtering process significantly. The filtering process simply removes all rows from the HF Dataset, which appear in the dictionary of *labeled_ids*.

Training a Tokenizer The job of the tokenizer is to transform a natural sequence of text into a list of input IDs used as input for the model. For fine-tuning tasks, the necessary tokenizer is typically attached to the pre-trained model and, using the Hugging Face libraries, loading a tokenizer is as simple as:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

When pre-training a model from scratch, the tokenizer also needs to be "trained" from scratch. The Hugging Face libraries make this process very easy as well, and can be done as follows:

```
old_tokenizer = AutoTokenizer.from_pretrained(checkpoint)
new_tokenizer = old_tokenizer.train_new_from_iterator(training_corpus,
                                    vocab_size=50265)
```

A new tokenizer is created by first defining the configurations. This can be done by either loading an existing tokenizer or defining it manually by specifying the tokenization algotithm and the special tokens. In this case, an existing tokenizer was loaded using a checkpoint, resulting in a tokenizer containing the desired configurations. These checkpoints were *roberta-base* and *gpt2*, which both use the same tokenization algotithm called Byte-Level Byte-Pair Encoding (BPE). As mentioned, an existing tokenizer should not be used when pre-training, because the vocabulary might be different between the two training Datasets. Conveniently, Hugging Face's Tokenizer classes have a *.train_new_from_iterator()* function, to which a function, (returning a Python generator), and a desired vocabulary size were passed. The generator would load 1000 data samples from the training data at a time, thereby removing the need to load all samples into memory at once and crashing the computer. The result of the *.train_new_from_iterator()* function was a tokenizer created by using the same algorithm as the original tokenizer, the difference being the training data, which would cause the vocabulary to turn out differently and more accurately represent the training data.

Two tokenizers would need to be trained before pre-training, one for *roberta-news* and one for *NewsGPT*. The need for two stems from the decision to train the generative model using the full Dataset, meaning the Dataset that includes articles from the labeled data. The generative model was only meant to generate text, and while it could, it was not meant to be used as a base-model for fine-tuning. Therefore, there was no reason to reduce the pre-training corpus for *NewsGPT*, resulting in two different Datasets, and therefore two different tokenizers.

Tokenization Algorithms A tokenizer is really little more than a dictionary mapping from a vocabulary of tokens to the corresponding input IDs (integers starting from 0). A simple concept of such a dictionary could be one achieved by stopping at the pre-tokenization step and simply separating whole words using the space character, counting the frequencies of those words, and making a dictionary of the most common ones. A sequence such as "I do not know.", would be tokenized and result in something like [132, 101, 23, 405]. The problem with this type of tokenizer, would be the amount of words the vocabulary needed to contain to cover all words, which is why it would make use of a special "<unk>" character, in cases where an input word was unknown.

Regular BPE mitigates the above mentioned issue by allowing the vocabulary to consist of sub-words [15]. BPE also uses a pre-tokenization step, splitting the sequences up into whole words, but it continues beyond that. A base vocabulary, consisting of single characters, is built using the training corpus, after which each word is split up into the characters from the base vocabulary. The base vocabulary is then expanded upon by merging the most common symbol pairs found in the corpus words, using symbols from the base vocabulary. With a corpus such as ["had", "mad", "sad", "bat", "bats"], the base vocabulary would be ["a", "b", "d", "m", "s", "t"], with the most common symbol pair being "ad" with three occurrences followed by "ba" with two occurrences. With "ba" (and "ad") added to the vocabulary, "ba" + "t" becomes a possible combination, which happens to be the next most common case, so "bat" is added to the vocabulary. This merging process continues until the desired vocabulary size is reached. Using the trained tokenizer, a word such as "hat" would get the useful tokenization: ["h", "at"], even though "hat" did not appear in the training corpus, however, "glad" would be tokenized as ["<unk>", "ad"], because "g" and "l" are not part of the vocabulary.

Byte-Level BPE [38], which is the tokenizer used in this project, completely removes the need for the special "<unk>" character by having the base vocabulary practically contain all possible characters. If the base vocabulary had to actually contain all possible characters, it would be quite large, so instead, the characters are broken down to a byte-level, meaning the base vocabulary consists of 256 bytes from which all possible characters can be created.

To indicate a space character, the BPE-tokenizers (used by RoBERTa and GPT-2) add a special Ġ character to the beginning of a word. If the word is split up into multiple tokens, only the first of those tokens includes the Ġ. This makes it possible for a list of tokens to be transformed back into the original sequence. Something like WordPiece (used by BERT) flips the logic and assumes there should be a space character between all tokens unless otherwise indicated by the "##" symbols added to all tokens except the first in a word.

Running Training Hugging Face made training a model very approachable with their Trainer class and wide array of readily available model architectures, both for fine-tuning and language modeling.

The Dataset used to pre-train the generative models contained 13,052,885 samples, which is lower than the last reported number in section 2.1.2 because 0.5% was extracted in case it was needed for testing. The Dataset used to pre-train the RoBERTa-like models contained 12,928,029 samples, which is smaller than the Dataset used to pre-train the generative models because some data was extracted for fine-tuning.

RoBERTa The steps for running pre-training for the RoBERTa-like language models is shown in the following Python snippet:

```python
# tokenize the raw dataset using the trained tokenizer
tokenized_datasets = data_preprocessing(datasets, tokenizer)

# create a data collator, in charge of batching the data
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=
                                     True, mlm_probability=0.15)

# define the training arguments, including learning rate, evaluation
                                     strategy and more
training_args = TrainingArguments(...)

# create a Trainer object by passing the untrained model, data, training
                                     arguments, etc.
trainer = Trainer(
        model=base_model,
        args=training_args,
        train_dataset=tokenized_datasets["train"],
        eval_dataset=tokenized_datasets["validation"],
        data_collator=data_collator)

# start training
trainer.train()
```

The overall training process was almost identical for the other training objectives, both for pre-training and fine-tuning, so the step by step process will only be written up this once. The difference between them was the implementation of the *data_preprocessing()*

function, the type of *data_ collator* used and the type of model being trained. Those will be explained for each objective.

Preprocessing for *roberta-news* only involved tokenization using the custom trained tokenizer, and for *roberta-gen-news* the pre-trained tokenizer from *roberta-base* was used. Note that "preprocessing", in this case, does not refer to the cleaning and packing processes described in section 2.1.2. At this point, the data is assumed to be cleaned.

The DataCollatorForLanguageModeling [16] was utilized for all language models. For the RoBERTa models, which used masked language modeling, the *mlm* parameter was set to *True* and the *mlm_probability* was set to 0.15 to mimic the original RoBERTa. The *mlm_probability* being 0.15 means that 15% of the tokens will be used in the masking process. 80% of those 15% will be masked with the special "<mask>" token, 10% will be replaced with a random token, and the last 10% will remain unchanged. This is the same masking process used for the models described in the RoBERTa paper [10]. The corresponding label of the masked tokens would be the same as before being masked, and every token that was not masked would have -100 as its label to indicate it should be ignored during loss calculation and evaluation. Besides masking and labeling, the data collator also made sure that the samples were padded properly with a special padding token in each batch, such that all samples had the same length as the longest sample in the batch.

The model used was the RobertaForMaskedLM [35], which loads a RobertaModel for the base and adds a RobertaLMHead to the head of the model. This model has a custom implemented *.forward()* function which calculates the loss and returns an output according to the learning objective.

The training arguments were chosen to mimic those from the original RoBERTa for better comparison. The *roberta-base* model was trained using batches of size 8,000 for 500,000 steps [33], which was not possible to mimic with the data and the GPU used for this project. The GPU ran out of memory for anything above 4 samples per batch, so gradient accumulation was used to effectively increase the batch size to 16 by adding 4 gradient accumulation steps. As a result, the model was trained using batches of size 16 for approximately 800K steps per epoch. The original *roberta-base* used a learning rate of 6e-4, which was initially mimicked for the training of *roberta-news*, however, the loss began to oscillate early in the training and some experimentation was needed to find a more suited learning rate. Ultimately, a learning rate of 2e-5 was used for *roberta-news* and because it was essentially a fine-tuning, a smaller learning rate of 2e-6 was used for the training of *roberta-gen-news*. The original *roberta-base* used 24k warm-up steps to start low and work up towards the chosen learning rate. For the RoBERTa-like models trained in this project, 50k warm-up steps were used. Initially, *roberta-gen-news* was not meant to be trained with warm-up steps, due to being a fine-tuning process, and the attention layers should therefore already have been "warmed up", however, it proved to be more effective to include them. Possibly, starting with a low learning rate when fine-tuning a language model would prevent "shocking" the already learned parameters but that is pure speculation at this point and was not investigated further.

The full list of training arguments can be found in the appendix.

GPT-2 As previously stated, the training process for the GPT2-like models was very similar to the training process for the RoBERTa-like models. The few differences are described here.

The generative models used the same prepossessing function as the RoBERTa models, although with a different tokenizer as explained in "Training a Tokenizer".

The generative models used DataCollatorForLanguageModeling like the RoBERTa models but with the *mlm* parameter set to *False*, due to the learning objective being causal language modeling (see section 1.2.1), which means no masking process is used and the labels are simply the same as the input.

Rather than RobertaForMaskedLM, the generative models used GPT2LMHeadModel [21], which is structured similarly by adding the base GPT2Model and a head.

NewsGPT and *GenNewsGPT* used the same training arguments as *roberta-news* and *roberta-gen*-news, respectively.

Training Duration Due to the masked language modeling objective randomly masking 15% of the tokens, there is a good chance that the second epoch of training would have other tokens masked, meaning that the training data would essentially be new. For this reason, *roberta-news* was trained for 4 epochs, at which point the loss (both validation and training) saw very little improvement. Training for 4 epochs took approximately 16 days of non-stop training using a GeForce GTX 1070 GPU. The loss during fine-tuning of *roberta-gen-news* stopped improving very early and training was stopped after 1 epoch, lasting approximately 4 days. *NewsGPT* and *GenNewsGPT* were both trained for 1 epoch, lasting 4 days each. In total, training lasted approximately 28 full days.

Checkpoints were saved every 10,000 steps (out of 800,000 per epoch), but were manually deleted to only keep checkpoints for every 40,000 steps due to storage limits.

Access the Pre-Trained Models The pre-trained models were uploaded to the Hugging Face Hub and can be found at [67].

The model pages include an interactive input field where a user can try the inference of the models via a GUI.

2.2.2 Fine-Tuning

The fine-tuning process used the same training process as the pre-training, although with different learning objectives than language modeling.

Single-Label Sequence Classification For the fine-tuning tasks a tokenizer was loaded directly from the pre-trained model about to be fine-tuned. The data used for the single-label classification task needed no preprocessing besides tokenization.

The sequence classification tasks used the DataCollatorWithPadding [18] to prepare batches. It padded the samples like DataCollatorForLanguageModeling without masking or labeling.

The model used for sequence classifications was RobertaForSequenceClassification. [36] When loaded like demonstrated in the following snippet,

```
model = RobertaForSeqeunceClassification.from_pretrained(
                                    pretrained_checkpoint, ...)
```

some of the pre-trained model's layers were automatically removed and a new, untrained classification head was attached.

Even though the learning rate used for language modeling was 10 times smaller when fine-tuning (*roberta-gen-news*) compared to pre-training (*roberta-news*), the learning rate used for these fine-tuned classification models was not, as it proved to be inefficient. Training of the classification tasks used a learning rate of 2e-5 with no warm-up.

Multi-Label Sequence Classification Preprocessing for multi-label classification was similar to that of the single-label classification with the exception that an extra step was performed to reformat the labels. The labels for each sequence was initially a list of label IDs like [0,3,4]. Calculating the loss between [0,3,4] and [1,4] is not well defined so the labels were reformatted to be binary in structure. For each sequence, a list was created large enough to fit all possible labels. Each element in the list would either be 0.0 or 1.0: 1.0 if the index appeared in the original list of label IDs, 0.0 if not. A reformatting could be like the following example, where 5 is assumed to be the largest label ID:

```
[0,3,4] -> [1.0, 0.0, 0.0, 1.0, 1.0, 0.0]
```

The numbers were explicitly added as floating point because they needed to be used in a loss function and the libraries used did not convert the labels to floating point automatically.

Token Classification (NER) The data used for token classification also needed additional preprocessing besides simply tokenization. As described in section 2.1.3, the token classification data was labeled one whole word at a time, so when tokenizing the words, thus splitting the words up into smaller tokens, the labels would no longer match. Luckily, the Hugging Face libraries had a solution for that, and their token classification tutorial was followed rather closely. [40]

Firstly, because the sequences were already split up into words, the *is_split_into_words* parameter of the tokenizer was set to *True* to inform the tokenizer, that the pre-tokenization step of splitting on space was not necessary. When loading the tokenizer, the *add_prefix_space* parameter would be set to *True* to add a space before each token as there would have been if the sequence was tokenized normally with the BPE-tokenizer. This had the added benefit of adding a space before the first word in a sequence, which normally would have none. Adding a space to the first word in a sentence, means that a given word is going to be treated the same as if it appeared anywhere else in the sentence, where it naturally would be preceded by a space, assuming it is a word, which always starts with a capital letter.

Secondly, after the usual tokenization of each batch of sequences, the *.word_ids()* function was used to return a list of indices for each token which would correspond to the index of the original word before it was tokenized. For example, the following list of words:

```
words = ["These", "words", "are", "exciting."]
```

could become the following tokens (ignoring special tokens, using regular G as space):

```
tokens = ["GThese", "Gwords", "Gare", "Gexcit", "ing", "."]
```

In that case, the *word_ids* of that sequence would be:

```
word_ids = [0, 1, 2, 3, 3, 3]
```

This information was used to recreate the list of labels by iterating the *word_ids*, using each index to look up the original label, and pairing that with the token. If the token was the first token of a word, it received the same label as the original word, and if it was not it received -100 as a label to indicate that it should be ignored. Continuing the example above, the new labels would look like this:

```
labels = [0, 0, 0, 0, -100, -100]
```

Using *-100* as a label for some of the tokens will prevent symbol tokens, such as "." or "'s", to be considered part of a named entity. The downside is that it may sometimes cut off a useful part of a word or consider "Ġ(" to be a named entity, if the named entity was mentioned in parentheses.

For token classification the DataCollatorForTokenClassification [17] was used. Besides padding the inputs, like DataCollatorWithPadding, it also pads the labels.

For token classification the RobertaForTokenClassification [37] model was used.

Title-Description Match (Classification) The Hugging Face tokenizers allow for two sequences to be passed to a tokenizer simultaneously. This adds a special separator token between the sequences, such that a model has a way to "tell them apart". The tokenizers used in this project used the same token to indicate both separation and "end of sentence" (EOS), resulting in the actual indicator for separating sequences was two SEP tokens: one added due to the first sequence ending, the other added as a separator. Besides using two separate sequences during tokenization, the preprocessing for the title-description match task was practically the same as the other single-label sequence classification task.

2.3 Model Evaluation

This section describes the methods used to evaluate the performance of the models. This mostly applies to the fine-tuning tasks but a method was used to evaluate the pre-trained models directly as well.

All implemented functionality used in the fine-tuning evaluation process mentioned in this section can be found in the *evaluation.py* and *model_ comparison.py* files of the source code.

2.3.1 Pre-Trained Models

When training a model using the Hugging Face Trainer, a *compute_ metrics()* function could be passed, which would be run during an interval specified by the *eval_ steps* parameter and be printed along with with the validation loss, reported as *eval_ loss*. For the models trained with a MLM training objective, a function was created to prepare the data for an accuracy calculation, which was passed to the Trainer and run during training. For the CLM-models, only the *eval_ loss* was reported during training but other methods were used to evaluate the generated output after training.

RoBERTa The accuracy of the unmasking capabilities of the RoBERTa models were calculated by extracting the indices of the token labels, which did not have the value −100. These indices were used to extract only the token predictions, which had had the input token masked, meaning only the predictions, where the model did not know the answer. These predictions were paired with the corresponding token labels, and passed to a function to calculate the accuracy score. [26][54] Thus, the resulting accuracy was a metric for how many of the masked tokens, which the model had predicted correctly in a given batch.

During training of *roberta-news*, the unmasking of the validation data reached an accuracy of approximately 58%, which at first does not sound that impressive but being correct more than half the time is not bad, considering each masked token had more than 50,000 possibilities based on the size of the vocabulary. In comparison, *roberta-gen* had an accuracy of about 71% during the beginning of fine-tuning the language model to news. The reason the accuracy in this case is being reported as an approximate, is that it changed a lot between evaluation steps during training, probably as a result of the random masking process. One evaluation step may have masked the validation data in a way that was relatively easy to unmask, while the next had more difficult tokens masked.

The accuracy was only used to monitor the performance of the model during training, along with the loss, and not as metric of evaluation afterwards. The reason is that it may not be a very good metric to compare the pre-trained models against. As mentioned above, the generic model reached a much higher accuracy when evaluating using the validation data compared to the news-trained model, despite the validation data consisting of news. It may be the case, that the generic model, which was trained using a huge amount of generic data, is very good at predicting "generic" words, which is the majority of a sentence. Words such as "the", "he", "they", "was", "a", "may", "which", etc. This would result in a higher accuracy score, even if the news-trained model was more accurate with the news-relevant words, such as names of viral politicians or phrases such as "brexit", "COVID-19" or "scandal".

The actual metrics used to evaluate the pre-trained RoBERTa models would be the evaluation of the tasks they were fine-tuned to do.

GPT-2 The generative models would not be evaluated by fine-tuning them to perform down-stream NLP tasks. Perplexity, which is a common metric for evaluating a causal language model, was briefly considered as an evaluation metric, however, perplexity is not an ideal metric, when the goal is to compare the performance of different models [28], which is why it was ultimately not used in this project. Instead, the generative models were evaluated by analysing their generated output. Two types of analysis were performed to evaluate the output: a qualitative analysis and a quantitative analysis. The qualitative analysis was a very limited and biased evaluation of asking and answering the question: "Does the output look like news?". A few generated examples is shown in section 3.7.1. The quantitative text analysis is described in section 2.3.2

2.3.2 Text Analysis for Generated Text

The quantitative text analysis consisted of a sentiment analysis and a term frequency analysis. The aim was to determine whether a model trained using news articles was biased in any way compared to models trained using generic text.

All implemented functionality used in the text analysis process mentioned in this section can be found in the *text_generation.py* and *text_analysis.py* files of the source code.

Text Generation The first part of analysing the generative models was to choose a decoding strategy, write a collection of news worthy prompts and then generate a bunch of text.

Decoding Strategy The decoding strategy used for the quantitative part of the text analysis was top-k sampling in conjunction with top-p sampling, also called "nucleus sampling". [22]

A decoding strategy decides which tokens are chosen during text-generation after a model has been trained. With a greedy strategy the next token with the largest probability is chosen on every regressive step of the text generation. The greedy strategy results in a deterministic output for the same prompt, as the reached probabilities do not change

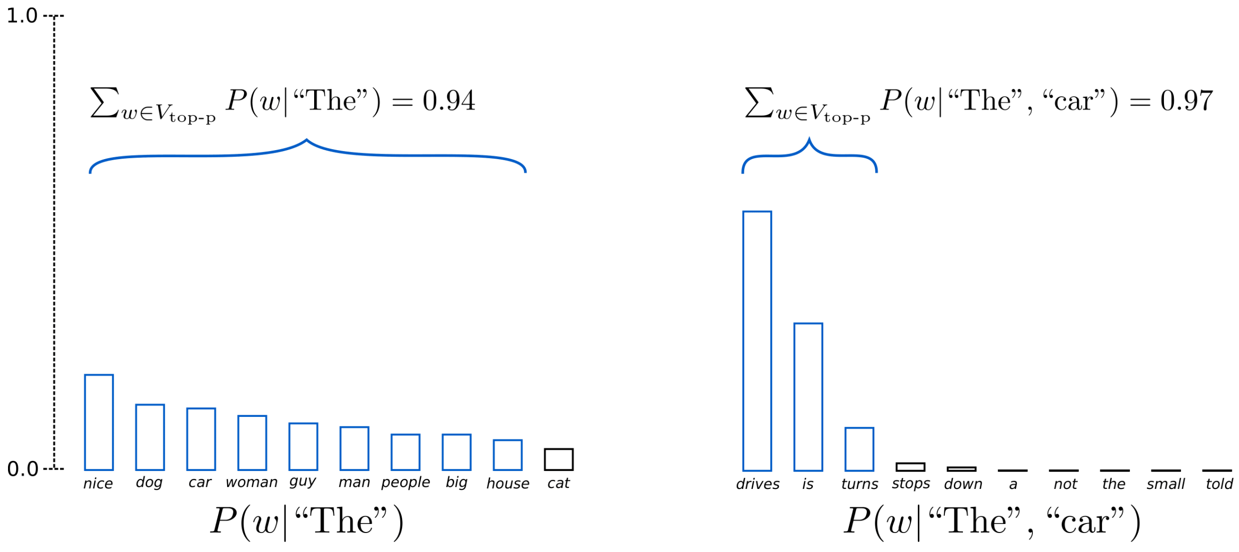


Figure 10: Top-p sampling. Source: [22]

after the model has completed training. To add some variance to the output, a randomized approach was chosen. Rather than choosing the most probable token every time, a random token is chosen. Obviously, with no limits introduced, this strategy would produce incoherent nonsense, so rather than choosing from among all possible tokens, a subset of good candidates were chosen. Top-k sampling selects a fixed number of the most probable tokens to randomly choose from, and top-p sampling selects a variable amount of the most probable tokens to choose from.

The chosen value of k was 5, meaning the next word was randomly chosen from the top 5 most probable tokens. While some variation is necessary to generate a large amount of samples with a limited amount of prompts, it would not be good to grant the generation process too much freedom, as the outputs would stray too far from the learned probabilities and thus not represent the corpus very well, which is counter productive when the goal is to determine how a model might learn a bias.

For top-p sampling the default Hugging Face value of 1.00 was chosen. This means, that for each token selection step, the smallest subset of tokens were chosen such that their combined probability was at least p. Figure 10 illustrates the concept well.

In order for the following inequality to hold

$$\Sigma_w P(w|''The'') \geq 0.92$$

the pool of candidate tokens needed 9 tokens. On the other hand, for

$$\Sigma_w P(w|''The'', ''car'') \geq 0.92$$

the pool of candidate tokens needed only 3 tokens. For this project, top-k and top-p were combined such that in the above example, if top-p sampling resulted in a subset of 9 candidates, top-k sampling would reduce it to the top-5.

Prompt Topics To add some structure to the text generation process, a JSON-file was created to contain a collection of topics, and its associated prompts. The following JSON-snipppet illustates the structure.

```
{
    "Donald Trump": {
        "prompts": [
            "Donald Trump",
            "Donald Trump is",
            "Donald Trump has",
            "Donald Trump will"
        ]
    },
    ...
}
```

In this case, Donald Trump is the topic and the associated prompts include his name and some verbs to initiate a sentence. All prompts and topics can be found in the appendix.

Generation The JSON-file of prompts was loaded and the outputs were generated using the Hugging Face pipeline. [32] The pipeline is simply an easy way to perform NLP-tasks using a pre-trained model. It includes all the necessary steps to go from text-input to text-output such as tokenizing the input text (natural language), passing the input through the model, decoding the output token IDs back to natural language, and in the case of text generation, applying the decoding strategies based on the passed function parameters. This would include randomly choosing among the token IDs with the top-5 highest probabilities and stopping when the target length of a sentence is reached. The pipeline also accepts a *checkpoint* parameter, which is where the pre-trained models were passed. The following Python snippet shows the steps needed to generate text with a Hugging Face pipeline:

```
generator = pipeline("text-generation", model=model_checkpoint, do_sample
                                       =True, top_k = 5, top_p = 1.0)
generated_text = generator(prompt, num_return_sequences=10, max_length=50
                                       )
```

After adjusting the structure of the output of the generator a bit, the output of the text generation function had the structure of the following JSON-snippet:

```
{
    "Donald Trump": {
        "greedy": [
            "1 of #prompts greedily generated sequences",
            ...
        ],
        "sampling": [
            "1 of (#prompts * variants) randomly generated sequences",
            ...
        ]
    },
    ...
}
```

The JSON-file included generated output from using both the sampling method and the greedy method. The greedy method was included for the qualitative analysis, to find out what the model would generate if it could follow the path of the absolute largest probability without sampling. Since the greedy method generates the same output for the same prompt, the collection of "greedy outputs" was a lot smaller than the "sampled outputs", and were not used in the quantitative text analysis.

Term Frequency Analysis The JSON-file containing all the generated output was used as the basis for the quantitative text analyses.

Absolute Term Frequency Within each topic, all natural sentences were first normalized by converting to lower case. A list of stop words provided by NLTK [48] was combined with a list of punctuation provided by the string library. The text was stemmed one word at a time using a PorterStemmer from NLTK [49] to make sure that plurals, and other variations of the same words, were not counted separately. During the stemming process, tokens from the combined *stops* list were ignored and not included. The result was a list of stemmed sentences in lower case with stop words and most punctuation removed. From the list of normalized sentences, three new lists were created, each with the sentences split up into tokens consisting of uni-, bi- and trigrams.

Each list of n-grams was used to count the frequencies of n-grams separately to create three top-n lists of terms. A list of "*topic keywords*" was passed to the counting function to make sure that tokens that appeared in the topic was not counted, as these would obviously be in the top-n list due to appearing in every prompt.

The absolute counting resulted in JSON-files of top-n term frequencies for each n-gram group within each topic and overall.

TF-IDF In an attempt to reduce the amount of very common words (not stop words) in the top-n lists, TF-IDF [62] was added as a frequency metric rather than simply using absolute term frequency. TF-IDF is Term Frequency weighted against the Inverse Document Frequency, resulting in high frequency terms, within a single document, scoring lower, if the term is common in other documents. To calculate the scores, the TfidfVectorizer from sklearn [63] was used.

First, the documents were normalized the same way as when counting the absolute frequencies. The list of unigram-sentences, returned by the normalization function, was turned into a list of normalized strings by joining the unigrams on the space character. The Vectorizer was "fitted" on all normalized documents, meaning the IDF scores were calculated and the vocabulary was learnt. After fitting, the Vectorizer's *.transform()* function was used on the documents from a single topic, resulting in a TF-IDF-weighted document-term matrix, where each term ranged from unigrams to trigrams, defined by the *nrgam_range* parameter of the Vectorizer. The Vectorizer had to be fitted on all documents before transforming each topic, because each topic had a custom set of topic keywords which needed to be passed to the Vectorizer's *stop_words* parameter. In the document-term matrix, each row represented a document and was the length of the vocabulary. The values in each row were 0, if the corresponding term of the index did not appear in the document, and a positive TF-IDF score if it did. The terms were summed along the columns and sorted in a decreasing order to find the highest scoring terms within each topic for each n-gram group.

The TF-IDF analysis resulted in similar JSON-files as the absolute term frequencies.

Sentiment Analysis Similarly to the term frequency analysis, the sentiment analysis was performed within each topic and overall. Once again, the Hugging Face pipeline [32] was used to make the process faster. This time, by passing the "sentiment-analysis" task parameter and a model trained for the task. The main model used for sentiment-analysis for this project was SiEBERT [42][3]. For sentiment analysis there was no good reason to normalize the input the same way it was done for the term frequency analysis. The only normalization which could be needed in some cases, was converting to lower case if the model was trained using uncased data. The default model for sentiment analysis using Hugging Face is "DistilBERT base uncased finetuned SST-2" [19] which is uncased. SiEBERT used roberta-large [34] as its pre-trained base which is not uncased.

To get the sentiment scores for each topic, all sequences generated within a topic were passed in a list to the pipeline, producing a list of sentiment outputs. If a sequence was negative it would come out of the pipeline with a "NEGATIVE" label and a positive value ranging from 0 to 1.0. If the sequence was positive it would be labeled "POSITIVE", also accompanied by a positive value. The value of the negative sequences were multiplied by -1 to make the value negative and saved in a list of negative values. The values of the positive sequences were saved in a separate list without manipulation. Three average values were then calculated: One for the negative values, positive values, and overall. The average negative and positive values turned out not to be very useful. The model was trained using data, which was labeled either positive or negative with no in between, which results in the model being very confident in its prediction. *"This is a neutral sentence."*, for instance, gets a POSITIVE score of 0.998. The result of this was that the average positive and negative scores were both very high for all topics, and it was not very meaningful to compare them. The overall average ended up being a lot more useful. Even though the result was calculated as if to find the average sentiment score, the result was more of an indicator of which label appeared the most, since all scores were almost 1 or -1 for POSITIVE and NEGATIVE sequences respectively.

With average sentiment scores from each topic of generated text, generated by models trained using news articles and a model trained using generic data, it was possible to determine whether models trained using news articles generally had a bias towards being negative or positive, comparatively.

2.3.3 Fine-Tuned Models

While the pre-trained RoBERTa models were evaluated during training as standalone unmasking models, the primary evaluation took an indirect approach, evaluating the models based on how they performed on NLP tasks after being fine-tuned for the job.

All fine-tuned tasks were evaluated by calculating an accuracy score and an F1 score using a subset of the test data extracted during labeling. Accuracy is a metric of how many predictions a model gets right compared to the total amount of predictions, which is why an even distribution of labels was chosen for all possible tasks. For the COVID-19 data, for instance, that meant 50% of each label. Assuming 95% of articles were about COVID-19, an accuracy score of 0.95 could be reached by always predicting 1. An even distribution was forced on the data, even though it may not represent reality, such that the model would have to learn the content and could not cheat by learning the distribution of labels. To keep it consistent, the distribution was maintained across splits. Accuracy is calculated as follows [26]:

$$accuracy = \frac{TN + TP}{TN + TP + FN + FP}$$

The evaluations were calculated using the Hugging Face libraries referenced in this section using something close to the following Python snippet:

```
metric = evaluate.load("accuracy")
results = metric.compute(references=[0, 1, 0, 1, 0], predictions=[0, 0, 1
                                                 , 1, 0])
```

The functions calculating the results do not return the confusion matrix, so the specific positive and negative values will not be reported.

Multi-Label Classification For the multi-label data, even though the distribution of labels were even, such that there were no surplus of one politician, the necessary reformatting created a situation where there was a large abundance of negatives. When the labels were spread out to a binary structure as explained in section 2.2.2, the majority of the values ended up as 0, because most articles only mention one or two politicians. Therefore, accuracy is not a very useful metric for this data, but luckily F1 score is.

The F1 score is the harmonic mean of the precision and recall. [27] Neither precision [29] or recall [30] takes into account the TN (True Negatives), which the multi-label classification model would easily produce a lot of. The F1 score is calculated as follows:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precission + recall}$$

$$recall = \frac{TP}{TP + FN}$$

$$precision = \frac{TP}{TP + FP}$$

The reason why the multi-label classification model would produce many true negatives was the way the logits were handled. The initial result of the model would be prediction scores for each label. These were passed to a Sigmoid function which transformed the scores into probabilities from 0.0 to 1.0. It is up to the user of a model to decide on a threshold which would turn the probability score into a binary yes or no. This number could be anything but some experimentation resulted in 0.33 being the chosen threshold. The majority of labels would not reach that threshold, and ideally only the correct ones would, which resulted in a lot of negative predictions, which is correct according to the labels, resulting in a large amount of TN.

Token Classification (NER) The test data for the token classification task was not chosen based on some distribution like the other tasks. Instead, two of the politician names were removed from the training data, and used as test data to see whether the model would be able to recognize a named entity it had not encountered during training. The accuracy and F1 metrics were calculated using seqeval [47][59] via Hugging Face [31]. Seqeval does not return accuracy "per NER type", only overall, so in those cases, only F1-score will be reported.

Evaluation Approach The reported findings for each task are a result of an average of multiple fine-tunings. When initialising a model head for a specific NLP task, the initial model parameters might be "lucky", resulting in accurate results after training. For this reason, the fine-tuning for each task was performed 10 times, each time with different initial parameters controlled by the *seed* parameter accepted by the TrainingArguments class. A loop was created, iterating through indices 0 to 9. In each iteration, a model was fine-tuned and saved, after which the model was loaded and evaluated. The index was used as the seed for each iteration, resulting in different initial model parameters for each iteration's model head. The evaluation results were then saved, and an average of each metric was calculated. After fine-tuning a pre-trained model 10 times, the next pre-trained model was chosen as the base and the procedure repeated. Due to the indices being used as seeds, each pre-trained model would get the same 10 initial model heads as the other pre-trained models, making the comparison fair.

A seed was initially passed to the Trainer (wrapped in the TrainingArguments class) along with the model, which means that the seed had no influence on the model, because the model was already initialised before the seed was set. Therefore, rather than passing an initialised model directly, a *model_ init()* function was defined and passed as a callback to the Hugging Face Trainer. The Hugging Face Trainer calls the *model_ init()* function after setting the global Torch seed, which means the model initialisation, in that case, is affected by the seed.

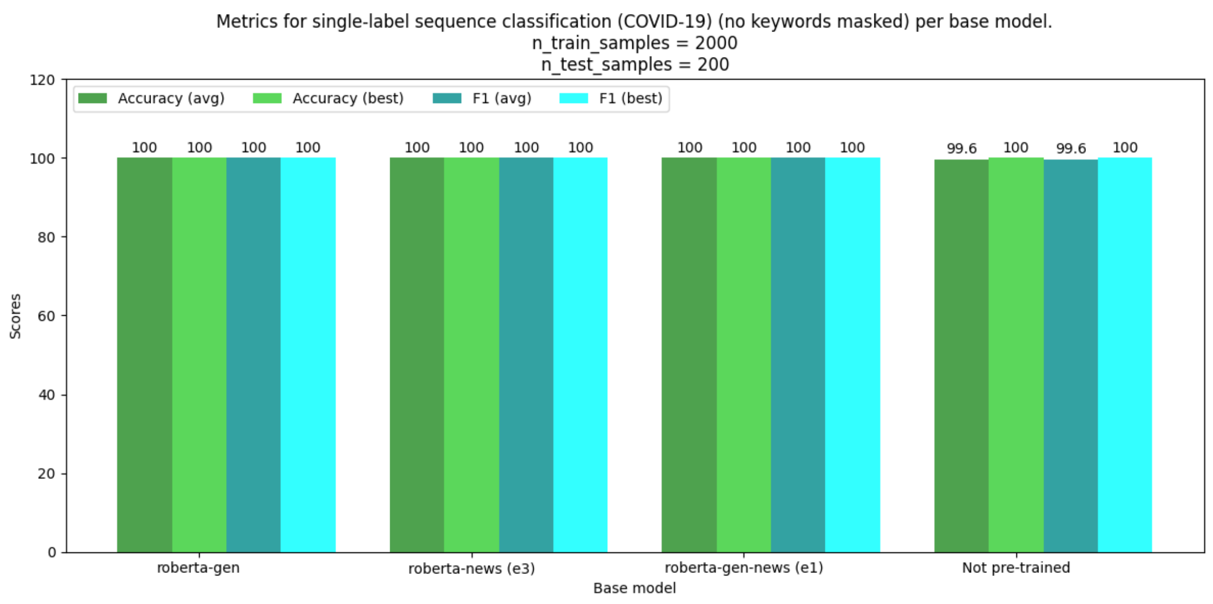


Figure 11: Fine-tuning results for single-label sequence classification after 10 separate fine-tunings.

3 Findings

For the fine-tuning and evaluation of the RoBERTa models, the training checkpoint at the third epoch of *roberta-news* (indicated by "e3") was used because it provided the best results. Accuracy and F1 scores are multiplied by *100* to make the charts look more neat.

The charts were produced using the *data_ visualiasation.py* file of the source code.

3.1 Single-Label Sequence Classification

The large amount of training data, which was extracted and labeled for sequence classification (about 60,000 samples), was chosen to remove any doubt, that there would be enough data for fine-tuning, because after pre-training, it would be too late to extract more data. Ultimately, the amount of samples was too high, given that fine-tuning had to be run 10 times for multiple tasks for multiple pre-trained models, so it had to be reduced. Simply using the *.select()* function did not suffice, as it would not maintain the even distribution of labels, so a few extra steps had to be taken.

Unless otherwise stated for the specific task, the pre-trained models were fine-tuned using 2000 samples and then evaluated using 200 different samples.

3.1.1 Evaluation Results

As illustrated in Figure 11 the single-label classification task was too easy to make any meaningful comparisons between the base models. Even with an untrained transformer

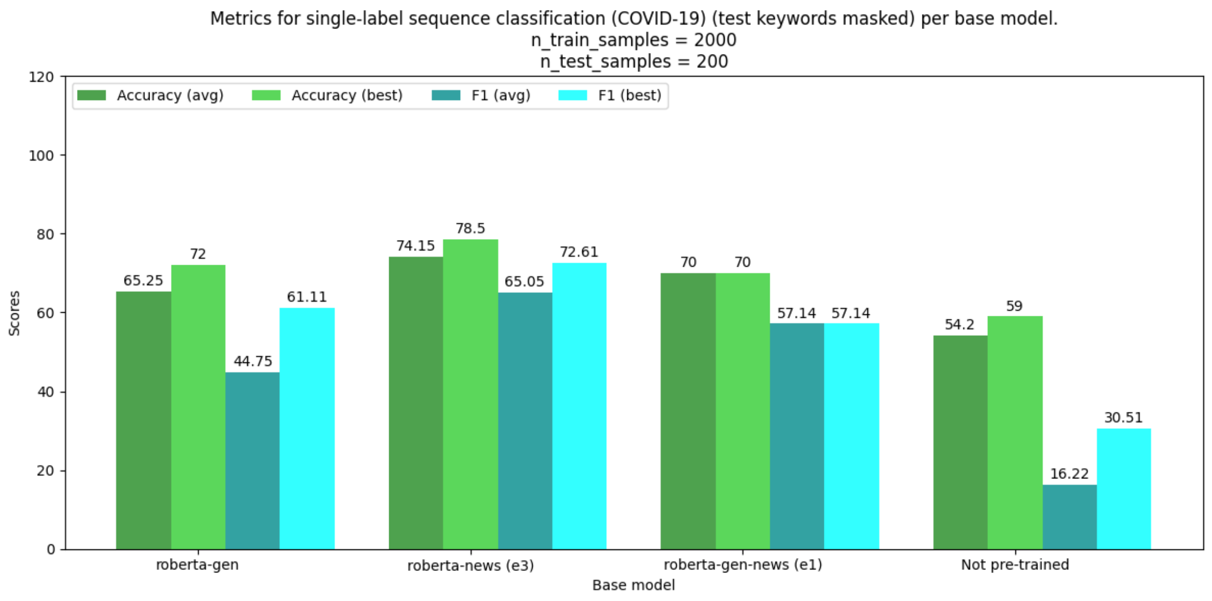


Figure 12: Fine-tuning results for single-label sequence classification after 10 separate fine-tunings.

used as the base model, the average accuracy was almost 100. For that reason, measures were taken to handicap the model. It makes sense that a fine-tuned transformer would accurately be able to classify whether an article was about COVID-19, because it was easy for a simple Python script to do it, simply by looking for the "COVID-19" keyword. To prevent the model from locking onto that keyword, the "COVID-19" keyword was substituted with "<mask>" and the attention mask was adjusted such that the indices corresponding to the masked tokens were 0.

Figure 12 shows the results for the single-label classification task, where the model was trained without masking the keywords but evaluated with the keywords in the test data being masked. This approach made it much harder for the model to accurately predict the labels. The model might have been completely focused on the "COVID-19" tokens when calculating its predictions, so to make it a bit easier, the model was trained with the keywords masked, giving it a chance to learn to focus on something else.

Figure 13 shows the results of training where both the train data and the test data had the "COVID-19" keywords masked. With this approach the base models can be meaningfully compared. The models still perform quite well, which could be because they just learned to latch onto another common keyword such as "coronavirus".

Just for good measure, the last possible training-evaluation combination was run as well. Figure 14 shows the results of training where the train data had the COVID-19 keywords masked but the test data did not.

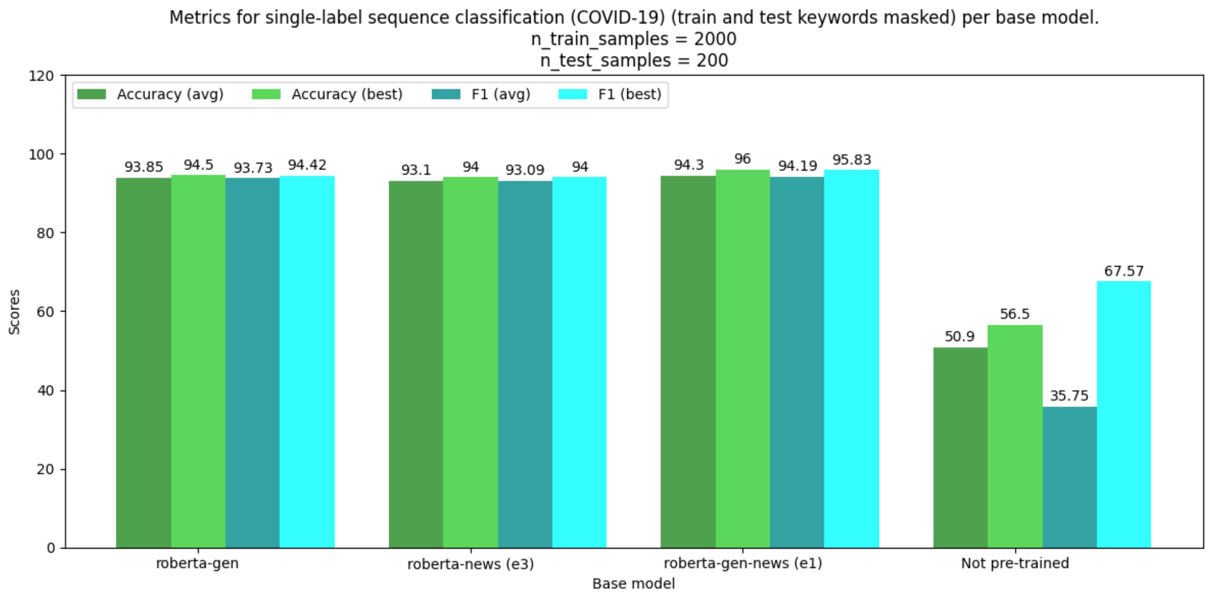


Figure 13: Fine-tuning results for single-label sequence classification after 10 separate fine-tunings.

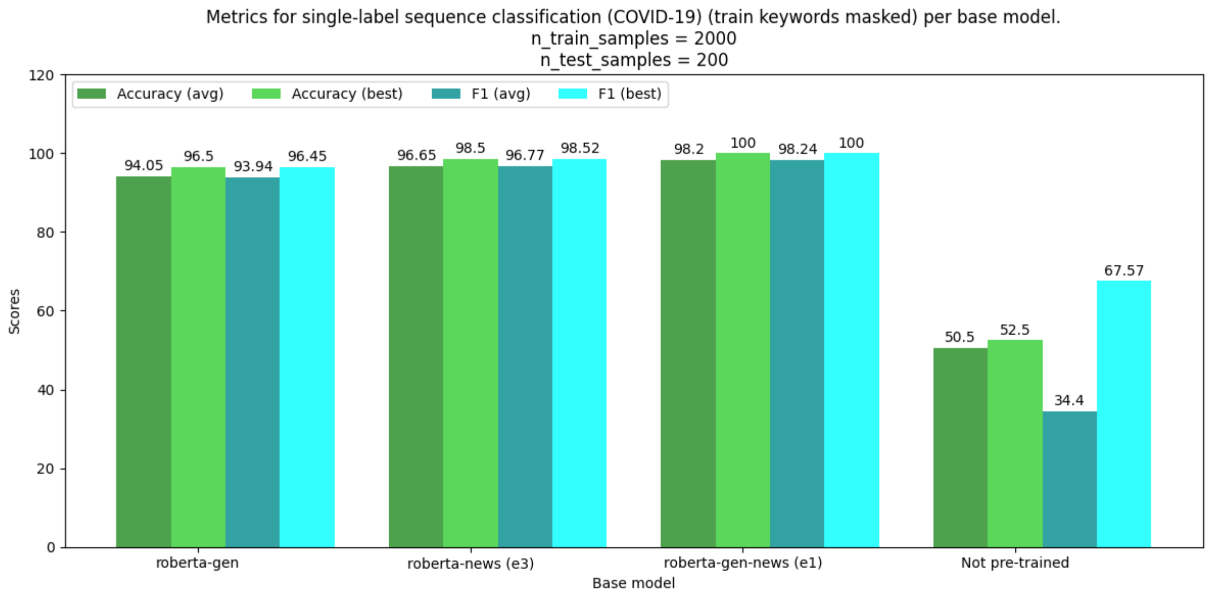


Figure 14: Fine-tuning results for single-label sequence classification after 10 separate fine-tunings.

3.1.2 Discussion

Looking at the results where either the train data or the test data had been masked, *roberta-news* performed slightly better than *roberta-gen* 2 out of 3 times. Both those times the "COVID-19" keywords were present in either the test data or the train data. When the "COVID-19" keywords were masked completely, *roberta-gen* did slightly better than *roberta-news* with *roberta-gen-news* outperforming them both by less than 1 percent.

All results, but especially the ones where the keywords were not masked completely, should be taken with a grain of salt. *roberta-base* (*roberta-gen*) was trained before COVID-19 was a thing, so the keyword is not part of its vocabulary and it has not seen it used in context during pre-training. Therefore, *roberta-news* and *roberta-gen-news* might have an advantage.

The "Not pre-trained" base model was included to demonstrate, that the pre-training does have an effect on the performance, which is probably obvious but it mitigates the notion that the models performed similarly, just because they were fine-tuned on a sufficient amount of data.

3.2 Multi-Label Sequence Classification

For the multi-label dataset, selecting the desired amount of samples exactly was not trivial so the train and test data ended up containing a little more than 2000 and 200 at 2026 and 207 samples, respectively.

3.2.1 Evaluation Results

The same four combinations of masking test and train data, which was performed for the single-label classification, was also performed for the multi-label classification task. The results can be seen in figures 15-18.

3.2.2 Discussion

The results of the multi-label classification task all illustrate the same thing: *roberta-gen-news* performs slightly better than *roberta-gen*, which performs slightly better than *roberta-news*. Looking at the results of the "Not pre-trained" model, it becomes quite obvious that accuracy is not a fantastic metric for multi-label classification, as discussed in section 2.3.3. Even with an F1-score of 0, the accuracy is still almost 90.

3.3 Token Classification (NER)

The token classification data was labeled manually, which was a time-consuming task, so only 120 articles ended up being labeled, resulting in 281 labeled sentences, 247 of which were used for training. The small amount of training samples resulted in token classification

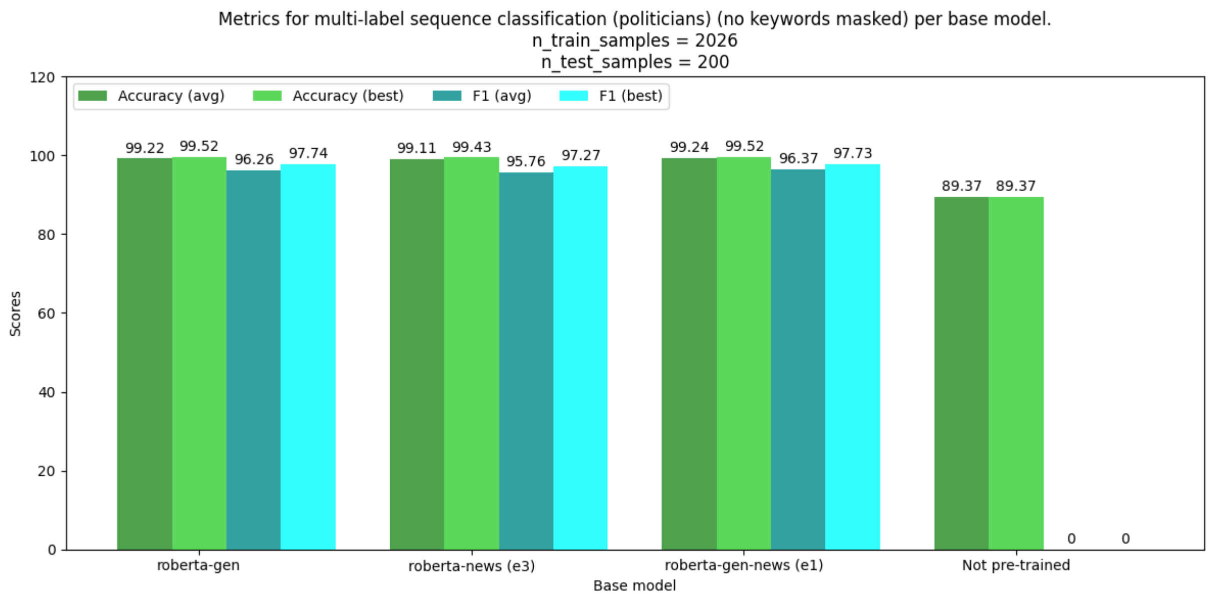


Figure 15: Fine-tuning results for multi-label sequence classification after 10 separate fine-tunings.

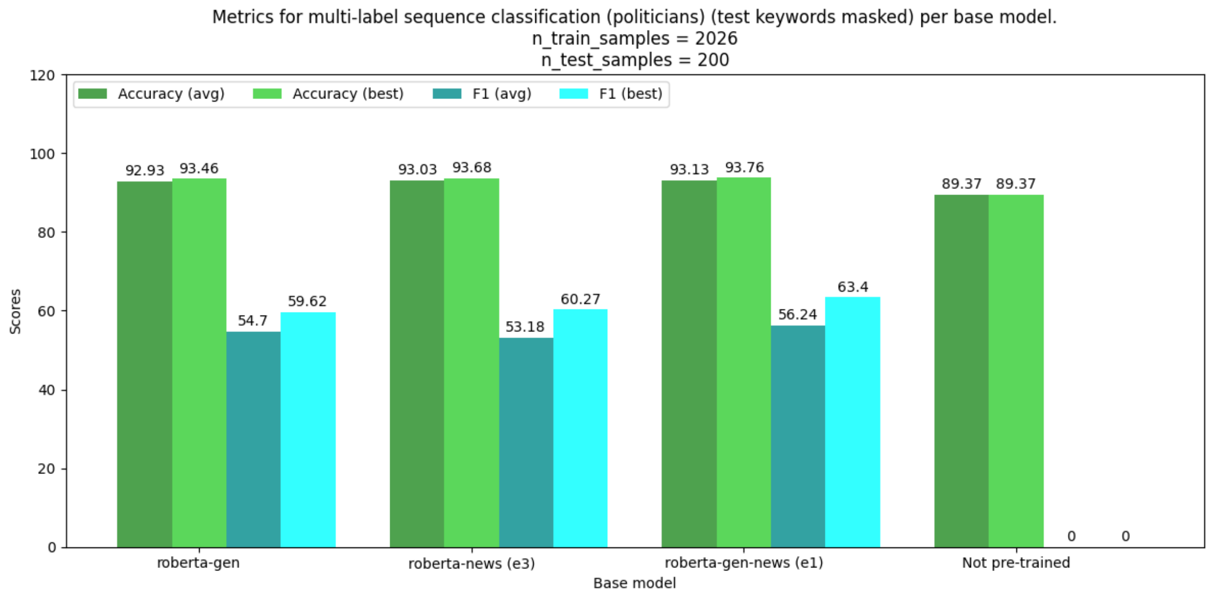


Figure 16: Fine-tuning results for multi-label sequence classification after 10 separate fine-tunings.

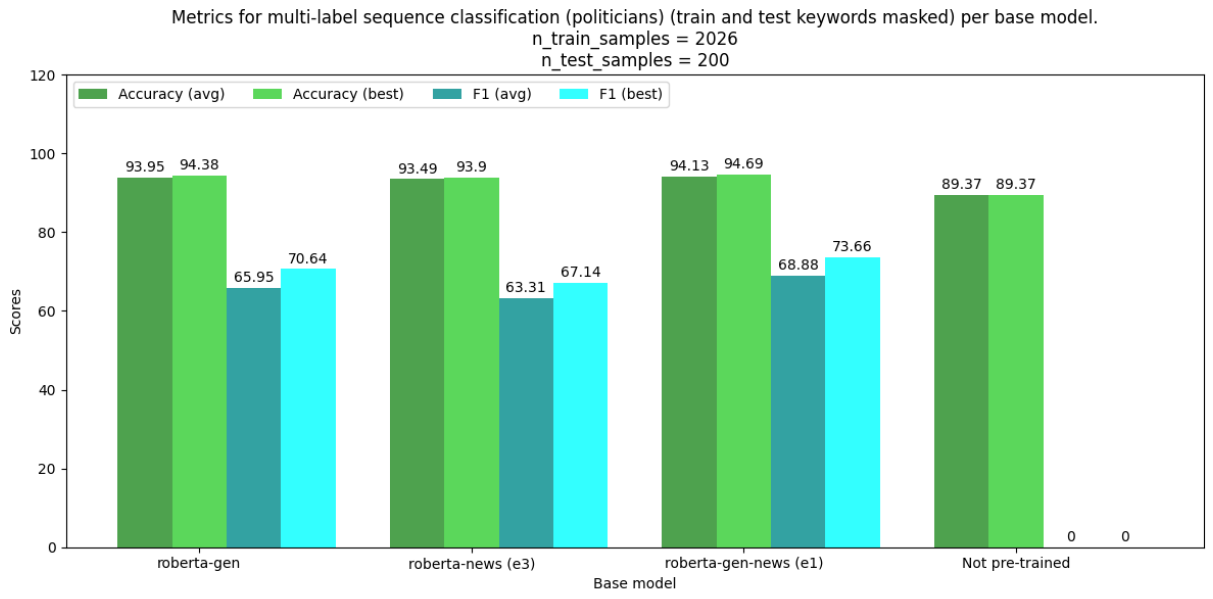


Figure 17: Fine-tuning results for multi-label sequence classification after 10 separate fine-tunings.

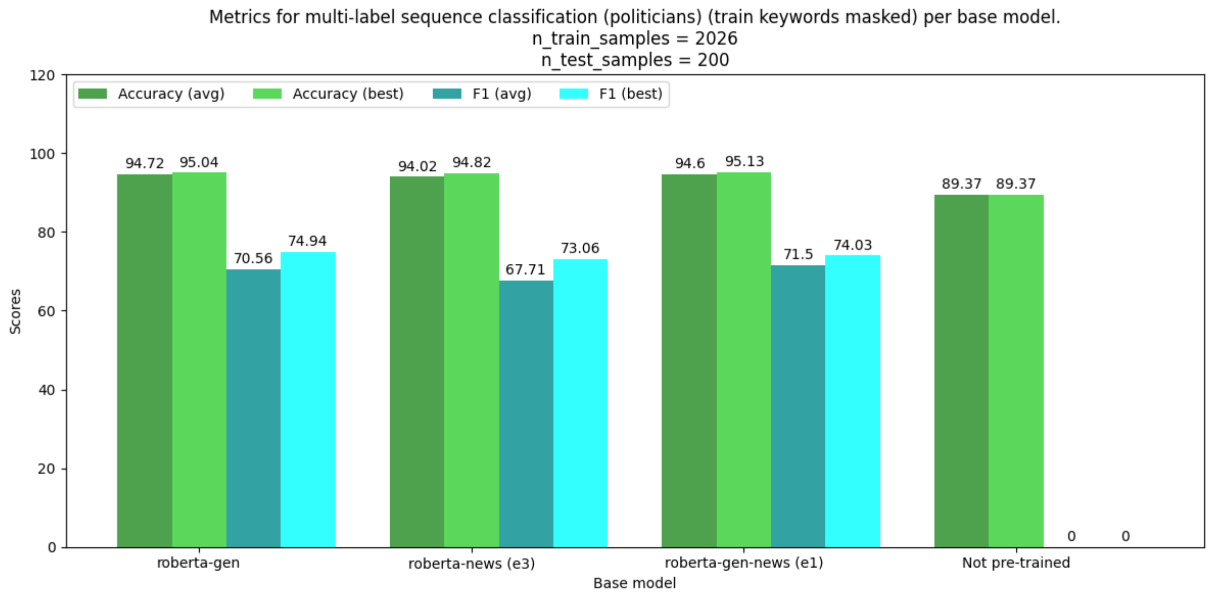


Figure 18: Fine-tuning results for multi-label sequence classification after 10 separate fine-tunings.

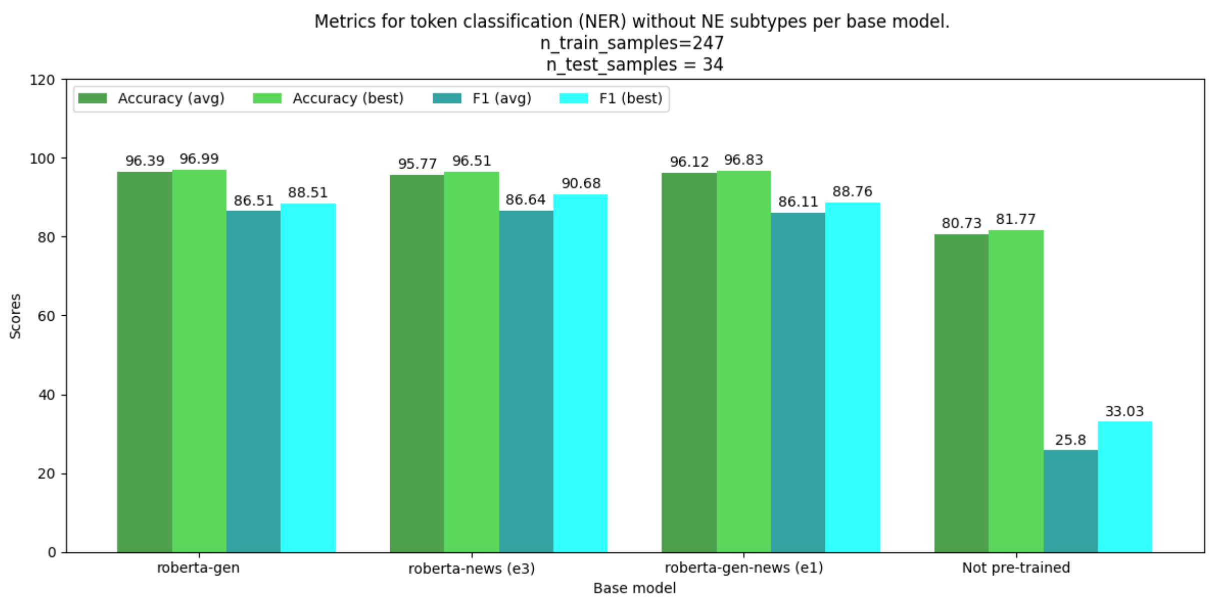


Figure 19: Fine-tuning results for token classification (NER) after 10 separate fine-tunings.

being the only task for which it was necessary to run fine-tuning for multiple (3) epochs to achieve any non-zero results.

3.3.1 Evaluation Results

Figure 19 shows the results of fine-tuned token-classification models, in which the token labels had been generalised, meaning that any of the three labeled named entity types had been labeled 0 or 1 (and 2 for "I-" tokens) to indicate whether they were a named entity without specifying subtype.

Figure 20 shows the results, where the data was not generalised. Looking at the results of each subtype of named entities, it became apparent that ORG and GPE was not able to be identified at all with F1 scores of 0, which makes sense, as there was a clear abundance of PER entities in the dataset.

Figure 21 shows the results of the PER entities only. Keep in mind that these metrics represent how well the models performed in identifying PER entities, which they, for the most part, was not fine-tuned to recognize.

3.3.2 Discussion

All token classification results show *roberta-news* performing slightly better than *roberta-gen* when considering the F1-score, which is the most interesting metric for a task with an abundance of negatives. It even beat *roberta-gen-news* 2 out of 3 times. The fact that *roberta-news* performed well on a token classification task might be due to it having a custom tokenizer, which was trained using the same training data. Even so, the fact that

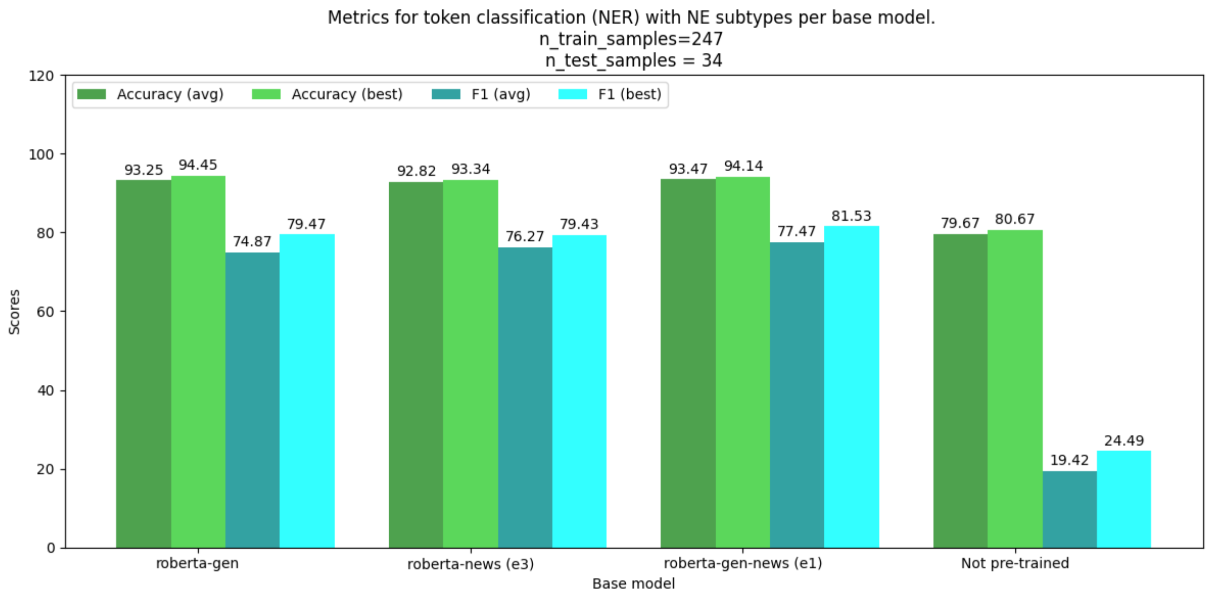


Figure 20: Fine-tuning results for token classification (NER) after 10 separate fine-tunings.

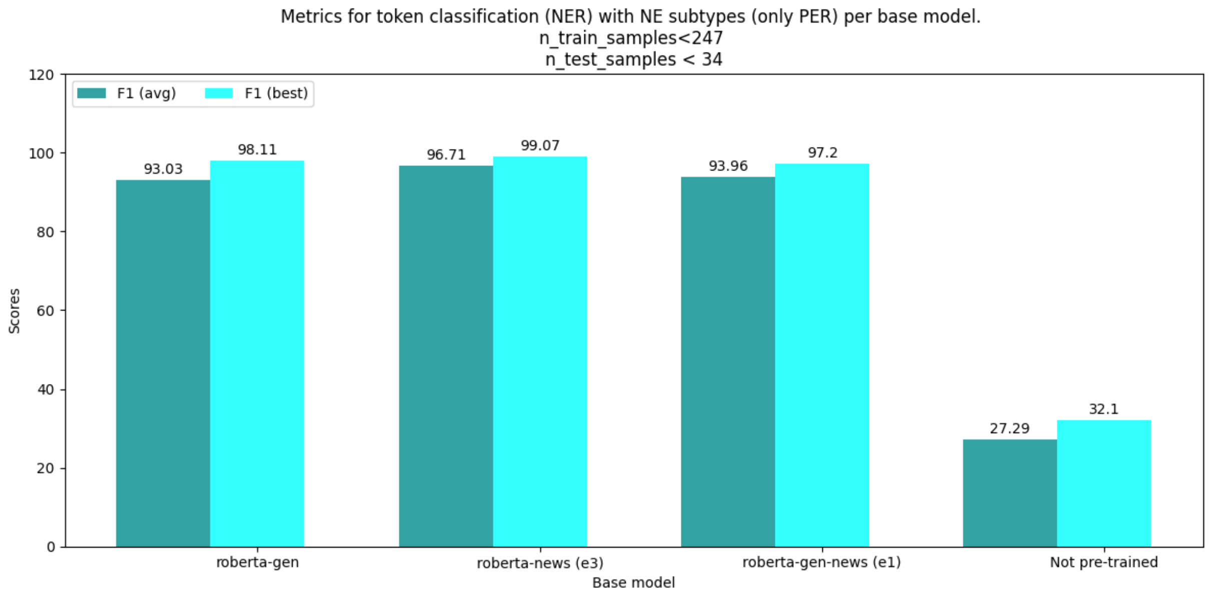


Figure 21: Fine-tuning results for token classification (NER) after 10 separate fine-tunings.

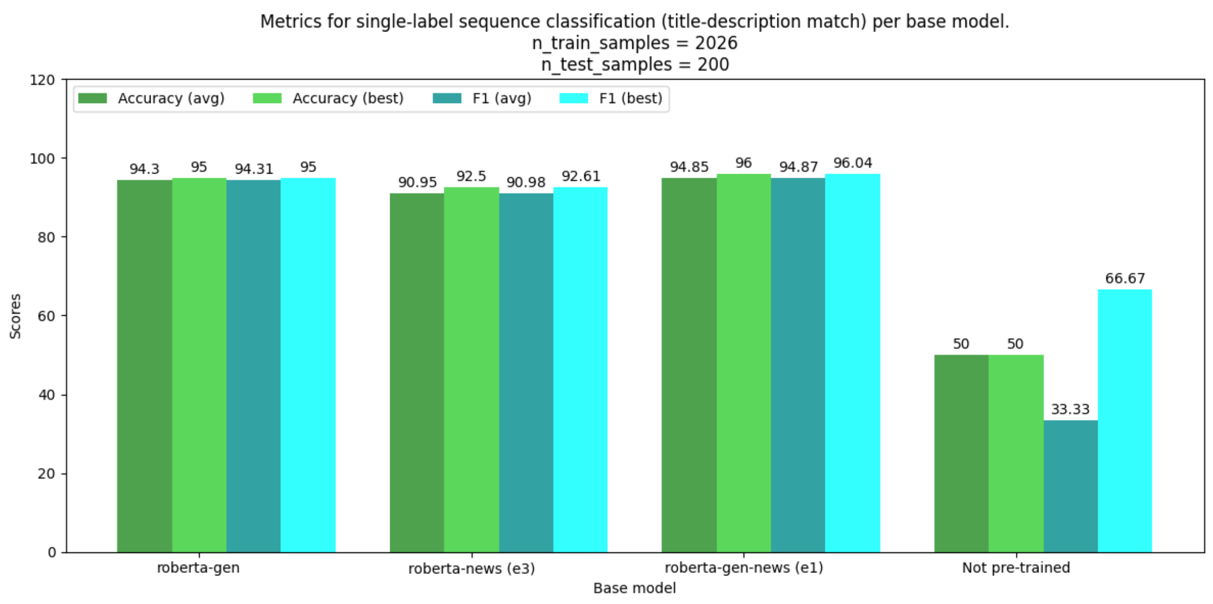


Figure 22: Fine-tuning results for single-label sequence classification of "title-description match" after 10 separate fine-tunings.

roberta-news performs better than *roberta-gen*, while having been pre-trained using only a fraction of the amount of data, which was used to train *roberta-gen*, is quite impressive. It is also worth emphasising that all of the pre-trained models performed very well, even with a small fine-tuning Dataset of 247 samples, which really speaks to the benefits of transfer learning.

3.4 Title-Description Match (Classification)

3.4.1 Evaluation Results

Figure 22 shows the results of the evaluation of the title-description match task.

3.4.2 Discussion

The results of the title-description match task follows the general trend of the other evaluations: *roberta-gen-news* performs best, *roberta-gen* second best and *roberta-news* performs the worst. The expectations for this task might have been that *robera-news* would have an advantage over *roberta-gen*, because it was pre-trained using concatenated titles and descriptions. As will be shown in section 3.7, the generative models pre-trained using news articles seem to learn the cadence of a news article, with the first sentence sounding like a title, and the next sentence elaborating on the details of the first sentence. This "knowledge" of the broad structure of the input might have given *roberta-news* an advantage but alas, the large train Dataset of *roberta-gen* seems to have had a larger impact on the results.

3.5 Summarized Fine-tuning Results

The general trend of the fine-tuning results shows *roberta-gen-news* performing the best, which is not surprising, as it had the advantage of being pre-trained using a huge amount of generic data, while also having been fine-tuned using news articles. Next best was *roberta-gen* which beat *roberta-news* most of the time, and if it did not beat it, it could possibly be explained by *roberta-news* having an advantage in recognizing specific tokens due to having a custom tokenizer. That being said, even though there was a pretty clear hierarchy of pre-trained models, the differences in scores were not that big, and the fact that a model pre-trained using approximately 39 million inputs through 3 epochs (*roberta-news*) can keep up with, and sometimes even beat, a model pre-trained using approximately 4 billion inputs (*roberta-gen-news*) could be quite significant.

3.6 Examples of Generated Text

This section is only meant to present examples of generated text from a chosen topic. The analysis of said generated text is presented in section 3.7.1.

3.6.1 Topic: Russia

The following are examples of generated text within the topic of "Russia". The bold text indicates the prompt passed to the generative models. The generated text has been manipulated a bit to look more presentable, which includes removing newline symbols.

GenGPT - Greedy

Russia *, which has been accused of meddling in the 2016 election, has said it will not be able to provide a list of the people it has interviewed.The US has said it will not be able to provide a list of the people it*

Russia is *not a country that can be trusted to protect its citizens.The United States has been accused of using its military to protect its citizens from the threat of terrorism.The United States has been accused of using its military to protect*

Russia has *been accused of using its military to support the Syrian government, and has been accused of using its military to support the Syrian opposition.The US has been accused of using its military to support the Syrian opposition, and has been accused of*

Russia will *not be able to take advantage of the situation in Syria to attack the Syrian government," he said."We will not be able to take advantage of the situation in Syria to attack the Syrian government," he said.*

GenGPT - Sampling

Russia *, and the US are trying to get a deal done to end the Syrian conflict."The Assad administration is trying to get a deal done," he told reporters after the meeting, according to AFP.He was referring to a meeting*

Russia is *a nation with a history of violence and a long history of corruption, and that it has been the target of an international criminal court.The United States has been involved in a number of investigations into alleged human rights violations in the region*

Russia has *been working hard to get Russia's cooperation in Afghanistan and other parts of the region back to the United States and to help stabilize the region. But the United States has not fully embraced the new leadership."We are not at peace*

Russia will *not tolerate any further aggression from Iran. Iran will respond by attacking the United States, the United Kingdom, the European Union, the United States and others."Iran will continue its aggression and aggression against the United States and the international*

NewsGPT - Greedy

Russia *'s Putin says U.S. will not be 'in the same place' in Syria. Russian President Vladimir Putin said on Friday that the United States will not be in the same place in Syria, despite the U.S. saying it would*

Russia is *'in the dark' about the US election. The US election is a "in the dark" about the US election, says the Russian Foreign Ministry. The Kremlin has been accused of "in the dark" about the US election. The*

Russia has *'no plans' to send troops to Syria, says US. The US has no plans to send troops to Syria, the US has said. The move comes after the US and Russia announced a military offensive in Syria. The US has said*

Russia will *not be able to use nuclear weapons in Syria, says US. The US has said it will not use nuclear weapons in Syria to help the country's security forces. The US has said it will not use the weapons in the region. Russia*

NewsGPT - Sampling

Russia *, China to meet in Washington on Wednesday, says US. Russia and China will meet on Thursday in Washington to discuss the issue of North Korea, a Russian Foreign Ministry spokesman said, after a meeting in Washington. Moscow has been in the. Russian*

Russia is *'the biggest threat to the US' says Trump. US President Donald Trump has said Russia was the biggest threat to the US in the world after the US... and Russia. In his speech to a rally in Moscow, Trump said he would*

Russia has *no plans to make nuclear war more powerful, says Russian Foreign Minister Sergei Lavrov. Russia has no plans to make nuclear war more powerful and will not have to do so, Russian Foreign Minister Sergei Lavrov said on Thursday, saying the country has no*

Russia will *not be 'un-American' in the 2016 election, says Donald Trump. The US president has claimed Moscow was a "un-American" in 2016. He also said the U.S. should not be "un-American"*

GenNewsGPT - Greedy

Russia *'s Putin says he will not be 'banned' from meeting US President Donald Trump. Russian President Vladimir Putin said on Friday he would not be banned from meeting US President Donald Trump, as he sought to reassure the world that he was not being*

Russia is *ready to use force to stop Russian military activity in Syria. Russia is ready to use force to stop Russian military activity in Syria, Russian Defense Minister Sergei Shoigu said on Tuesday. Shoigu said the Russian military was ready to use*

Russia has *'no interest' in Syria, says UN chief. The UN Security Council has said Russia has no interest in Syria and that it will not support the Syrian government. The United States and Russia have been accused of supporting the Syrian government. The*

Russia will *not be 'in the business of killing' Russia, Putin says. Russian President Vladimir Putin said on Thursday that the country will not be "in the business of killing" Russia, but that the country will not be "in*

GenNewsGPT - Sampling

Russia *'s military says Russia is preparing to attack Syrian air base. Russia says the Syrian air base in the country has been targeted by Islamic State (ISIL) forces. Russian Defense Minister Sergei Shoigu said on Thursday that Russian military forces had "*

Russia is *not a 'fricty country'. Russia, as a nation, is not aricty country, according to the Russian president, Vladimir Putin. In a new interview with DW, the president said he does not think the Russian Federation is*

Russia has *no intention of changing course after US-Russia summit. Russian President Vladimir Putin is unlikely to change course after meeting with US President Donald Trump, the president's spokesman said on Tuesday. Vladimir Putin, who is also the leader of Russia's main*

Russia will *not allow US troops to enter Syria, Russia minister claims. The US has not invited Russia into Syria since 2014, but Russia says it will not allow troops to enter Syria, the country's foreign minister said on Wednesday. He added*

3.7 Text Analysis of Generated Text

3.7.1 Qualitative Analysis

Section 3.6 shows a few examples of text generated by each of the models. Just by reading the examples, it becomes apparent that *NewsGPT* mimics the data, which was used to train it, by generating text with a "news cadence". The output of *NewsGPT* usually follows a pattern of generating a "title-sounding" first sentence, followed by a sentence which just repeats the first sentence with more details. The repetition is sort of true for *GenGPT* as well but only when using the greedy decoding strategy, which generally has a habit of almost exactly repeating the first sentence multiple times, not with more details. Looking at the examples generated using the sampling strategy, *GenGPT*'s repetition appears to be reduced while *NewsGPT* continues to produce samples of the title-description pattern. *GenNewsGPT* generates sequences of a structure similar to that of *NewsGPT*, suggesting that enhancing a generic language model by fine-tuning on a specific domain is enough to adjust its output noticeably. This repeated structure indicates that the training data has had a significant impact on the news model, which can be good if the goal is to generate news articles, but it begs the question what other biases are carried over from the data to the model.

3.7.2 Quantitative Analysis

Term Frequency Results within individual topics will only be presented here using TF-IDF scores, not absolute term frequencies. More results can be found as raw numbers in the *"/results"* directory of the source code, in files ending with *"_tf.json"* and *"_tfidf.json"*

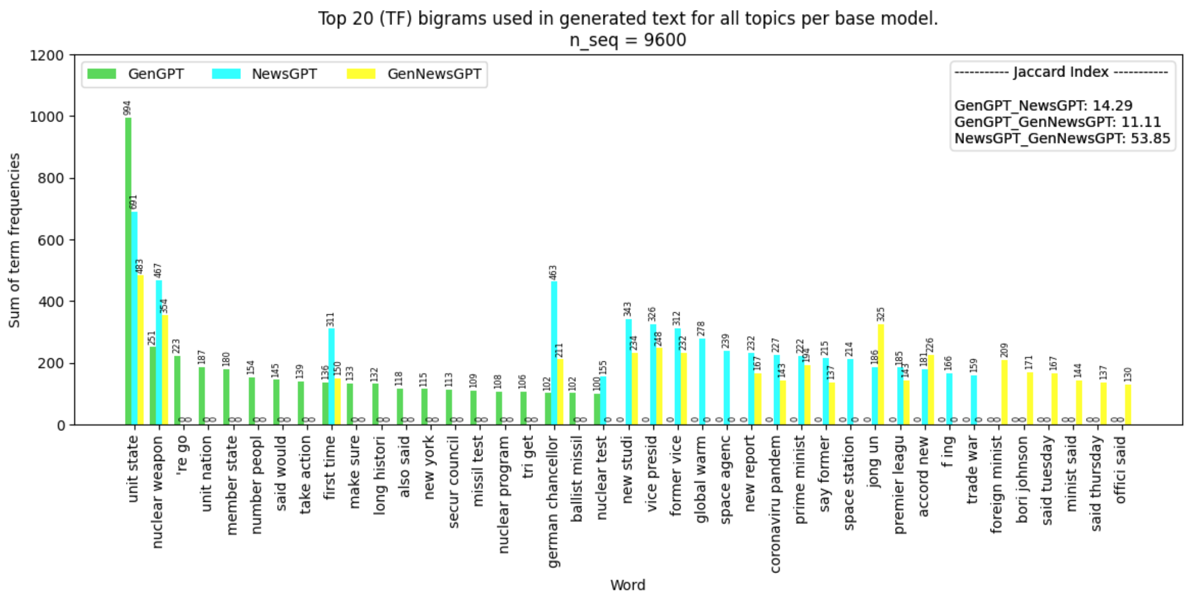


Figure 23: Term Frequency analysis results for generated text in all topics (TF).

and as charts in the *"/bar_charts"* directory but they add little to the conclusions and will therefore not be included in the report. The results in the report only show bigrams for the same reason.

Figures 23-27 show the results of the term frequency analysis performed on the generated text. Each figure contains the top-20 most frequent bigrams within text generated using the pre-trained generative models. For every topic, 100 sequences were generated for every prompt, of which there were four, resulting in 400 generated text sequences per topic, totaling 9600 sequences as indicated by the n_seq in figure 23. The top-20 bigrams are not necessarily the same for each model, so the charts include more than 20 terms. The top-20 terms are shown for all topics and within three chosen individual topics. All the topics can be grouped into three "super topics" which are "world leaders", "geo-political entities" and "the rest". A topic was chosen from each of those super topics to be presented in a chart.

The Jaccard indices attached to each chart show how similar each pair of top-20 lists are, with *Model1_Model2* in the text box meaning "how similar is the top-20 list of *Model1* to the top-20 list of *Model2*. These Jaccard indices range from 0 to 100, where 100 means that the two top-20 lists in question include all the same terms, although not necessarily in the same order, and 0 means that they share no terms. Comparing *GenGPT* to each of the two news models results in low scores compared to the result reached by comparing the two news models with each other. These results indicate that fine-tuning *GenGPT* to generate news manages to affect the model enough to be more similar to *NewsGPT*, which was trained from scratch, and overall, that both news models deviate noticeable from the generic model. A table view of the term frequencies, along with their ranks within each

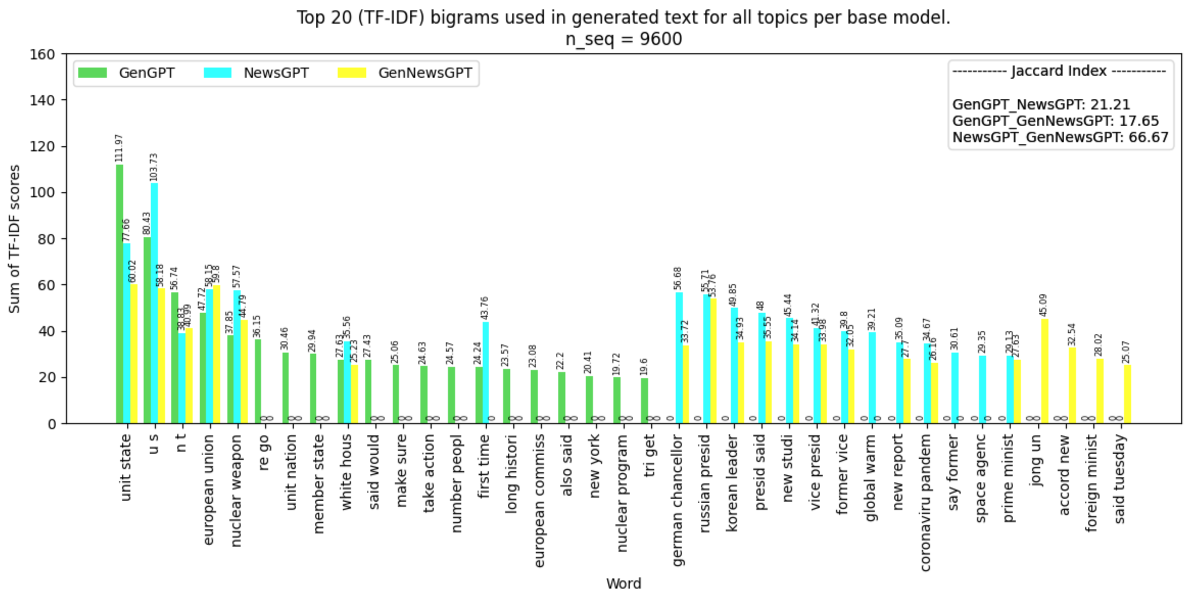


Figure 24: Term Frequency analysis results for generated text in all topics (TF-IDF).

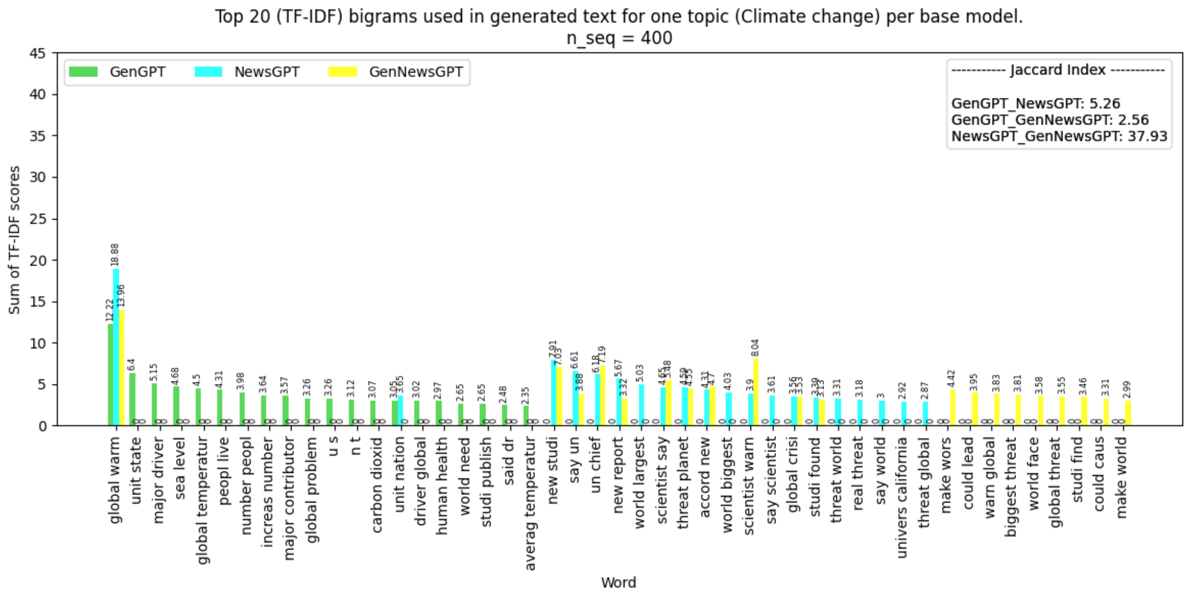


Figure 25: Term Frequency analysis results for generated text in one topic (TF-IDF).

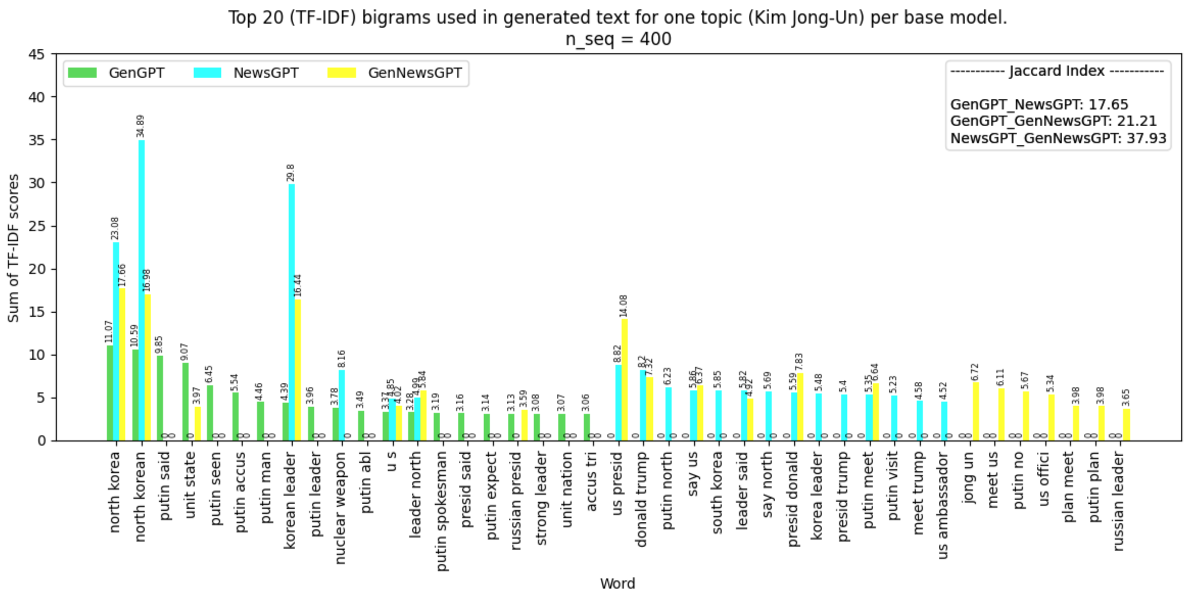


Figure 26: Term Frequency analysis results for generated text in one topic (TF-IDF).

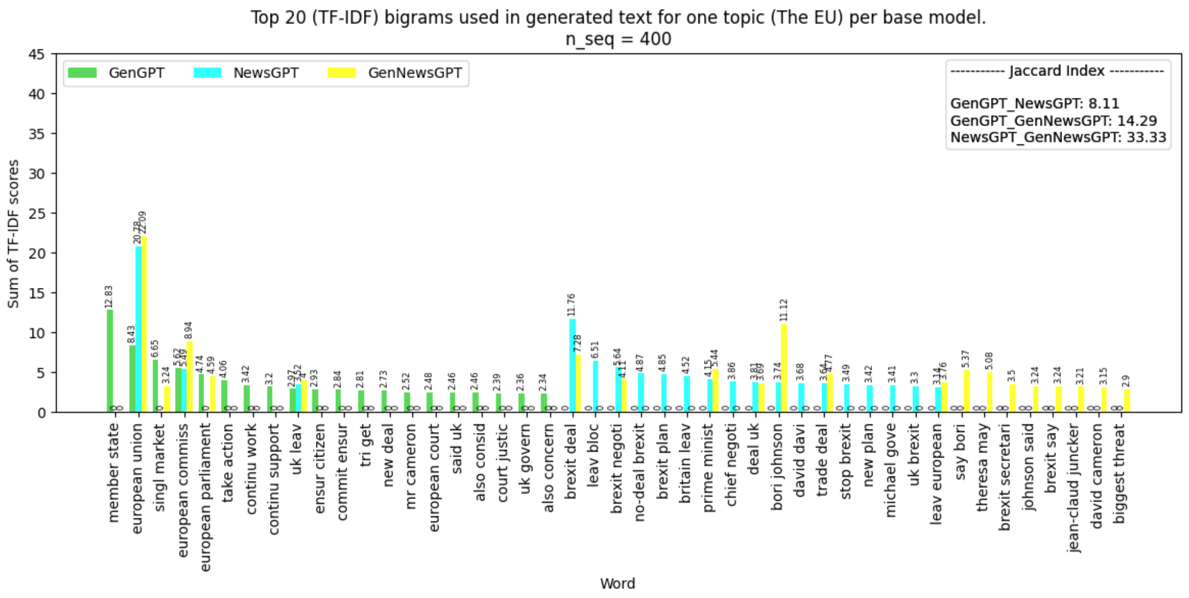


Figure 27: Term Frequency analysis results for generated text in one topic (TF-IDF).

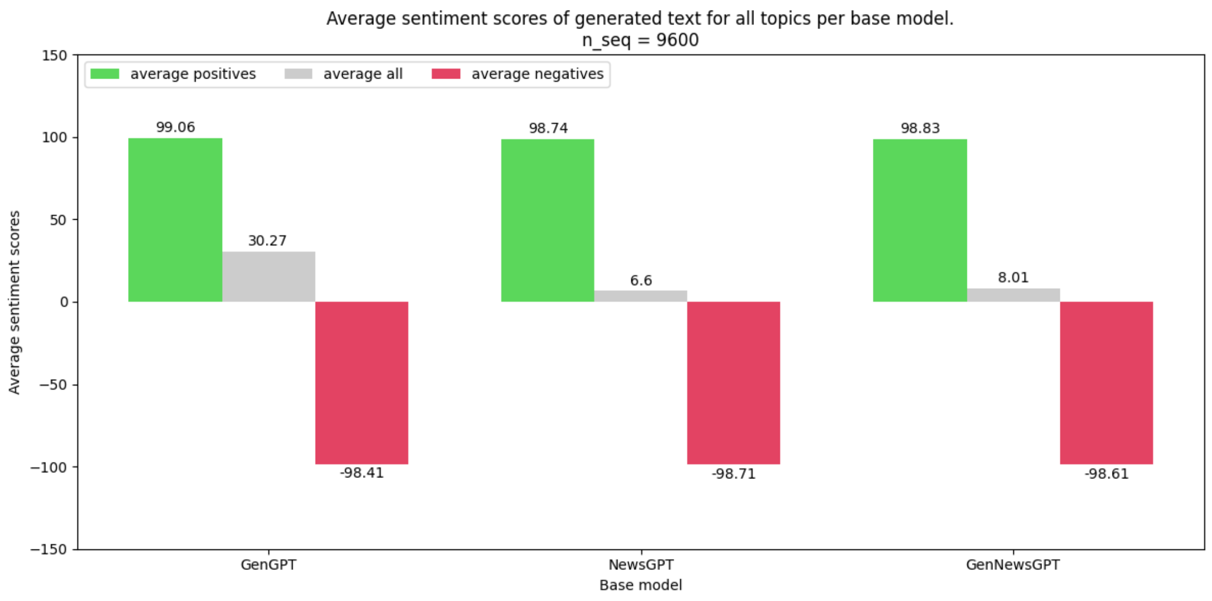


Figure 28: Sentiment analysis for all topics.

base model, can be found in the appendix (figure 32-36).

Looking closely at the individual top-20 terms from each model and trying to determine why a certain term is prevalent in news and not in generic text, and vice versa, is a highly speculative task, which could take into account the political landscape and other factors, and will thus be avoided. That being said, it appears that the top terms generated by *NewsGPT* and *GenNewsGPT* generally have a higher frequency than those generated by *GenGPT*. This could support the conclusion from the qualitative analysis, that *NewsGPT* repeats itself in a title+description format, which would result in repeated terms and thus higher term frequency scores.

Sentiment The sentiment analysis charts presented here are based on results from the SiEBERT [42][3] model, but using the default model [19] gave similar results. Figure 28 shows the average sentiment of the positive, negative, and overall results for all topics. As mentioned in section 2.3.2, the positive and negative averages are very close to 1 due to the models being very confident in their predictions. The more interesting metric is the overall average, which clearly shows, that the news trained models tend to generate more negative examples.

Figure 29-31 shows the overall average sentiment for each topic. The topics were separated into 3 groups, each shown in their own figure.

Like figure 28, these figures show that *NewsGPT* generally produces more negative outputs than *GenGPT*. Out of 23 topics, *NewsGPT* is more negative than *GenGPT* in 20 topics, clearly illustrating how the bias of a model's training data can be carried over to the model and affect its output. *GenNewsGPT* is also more negative than *GenGPT* the

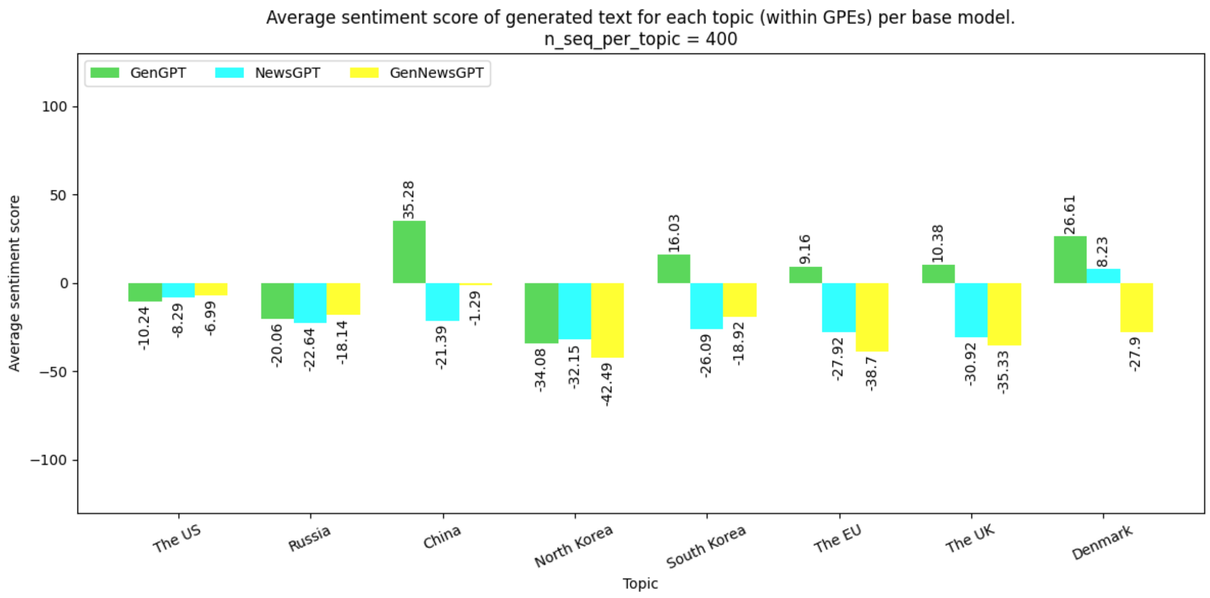


Figure 29: Sentiment analysis for some topics.

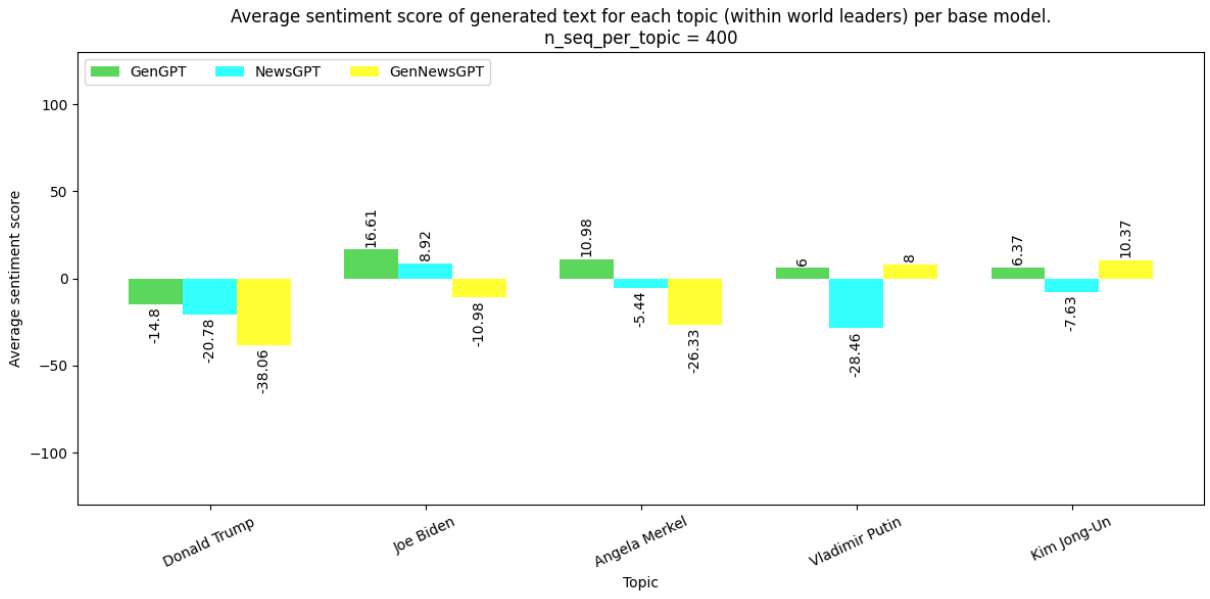


Figure 30: Sentiment analysis for some topics.

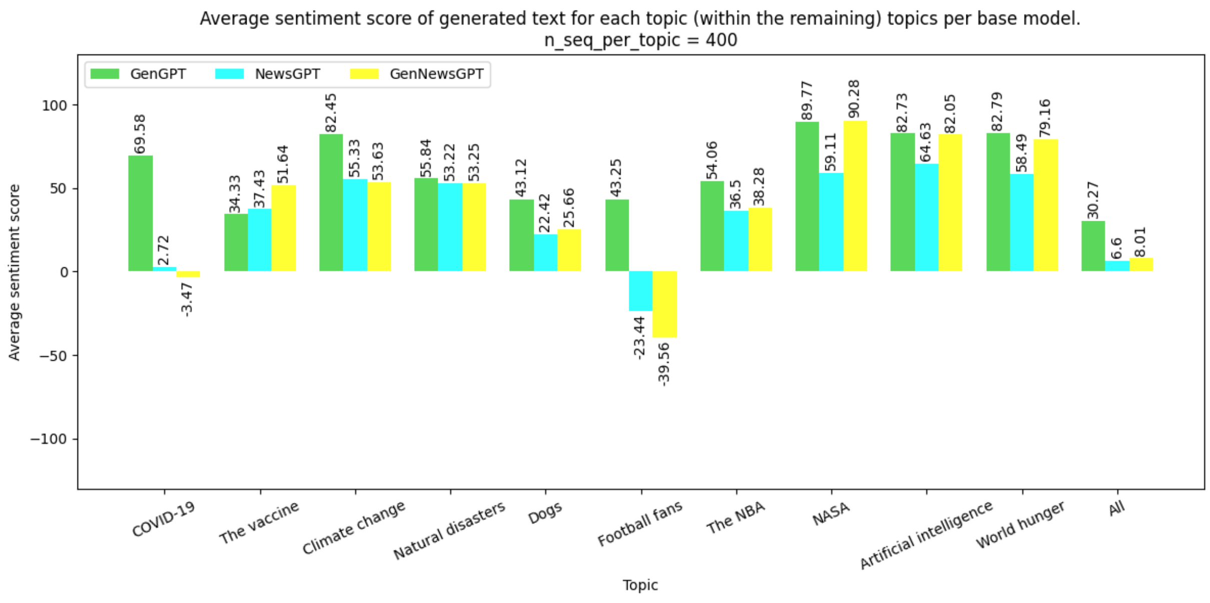


Figure 31: Sentiment analysis for some topics.

majority of the time, while its sentiment relative to *NewsGPT* varies.

4 Conclusion

The results of testing the hypothesis: *"a language model pre-trained using data from a specific domain (news), will perform better than a language model pre-trained using a more generic dataset, when both models are fine-tuned and evaluated on a domain-specific NLP task."* are somewhat inconclusive. Just looking at the numbers from the evaluation of the token and sequence classification tasks, the conclusion would be that a generic model, overall, performs better than a domain-specific model, while a generic model, enhanced by fine-tuning using a specific domain, performs the best. The only exception, where the domain specific model performed slightly better, was when the task was focused on specific tokens, which might be explained by its custom tokenizer. The information from the evaluations alone would disprove the hypothesis, however, it needs to be taken into consideration that the domain-specific model was at a massive disadvantage when considering the amount of inputs used to train it, which was smaller by a factor of 100 compared to the model trained using generic text. The fact that the domain-specific model could almost keep up with the generic model, and sometimes even beat it, could suggest that the hypothesis is true under more equal conditions, especially considering that the enhanced generic model outperforms both of them in most tests. That being said, the fact that the domain specific model can produce similar results to the generic model with comparatively very little data, is interesting and could be worth considering in some niche cases, where the choices are between using an existing pre-trained model and training a new one.

The results of the text analyses of the generated text further supports the conclusion, that the data, with which a model is trained, influences its performance. In the case of text generation, the measured difference was mainly in sentiment and cadence. The generative model trained using news articles generated text sequences which was phrased more like news, with a title and an elaborating description. Secondly, the text generated by the domain-specific model generated noticeably more negative text sequences across a wide variety of news worthy topics, demonstrating the bias which is carried over from the data to the model. While the measured difference in the generative models was not evaluated directly as a performance metric in NLP tasks, such as classification accuracy, it supports the conclusion, that the data domain affects a language model, which might include its performance on down stream NLP tasks as demonstrated in the evaluation of the fine-tuned models.

5 Future Work and Improvements

It would be interesting to see how the fine-tuning results might be different with different amounts of training and testing samples, which would include continuing the manual labeling process of the NER data, and choosing numbers other than 2000 and 200 for the other NLP tasks. Maybe *roberta-gen* would have an advantage on fewer samples due to being heavily pre-trained or maybe *roberta-news* would have an advantage on fewer samples due to already being specialised on the domain, while *roberta-gen* needed to learn it during fine-tuning. Those are questions, which could be interesting to explore.

Pre-training could have been handled better, and if the project were to be repeated in the future, more effort would be made to log and visualise the training process, such that a checkpoint could be chosen based on the best information possible. This is not to say, that pre-training was not monitored. The loss and validation loss were printed to the terminal and monitored for early stopping throughout training but the results were not saved.

Another candidate for future work would be to expand the amount of tested NLP tasks. The results point to the fact, that pre-training a model on a specific domain is an advantage on some NLP tasks, specifically tasks where a custom tokenizer could be the source of improvement, such as token classification. Expanding the types of tested NLP tasks to cover a wider area might create an overview, that could help developers who was trying to decide between fine-tuning a generic pre-trained model and pre-training a new one. This expansion of tested NLP tasks could include reporting the performance of the models on the GLUE benchmarks [1].

It would also be interesting to repeat the work done in this project on a different data domain. Intuitively, the linguistic differences between a crime novel and a news article, should be smaller than, for instance, the differences between a crime novel and a very specific paper of a natural science. It could be relevant to measure whether a more specific domain would produce enhanced results, similar to the *BioGPT* model [9][45].

The results of fine-tuning point towards the models, pre-trained using news articles, inheriting the biases of the data on which they were trained. It would be interesting to continue along the same lines to see how the sentiment in different topics would differ when limiting the training data to articles from individual outlets. On a similar note, it would be interesting to analyse the output of the generative models, when the prompts were written with the intent of provoking the models toward generating news. Transformer models can be incredible flexible, with ChatGPT, for instance, being able to provide answers in multiple languages, depending on the language in which a question was written. If the prompts were written in a way, which could provide GenGPT with some context to indicate, that the generated output should be news, it may perform more similar to NewsGPT.

Lastly, something which could be considered as missing from this project, which would be an interesting task for the future, is testing how the news trained model compares to the generic model when the situation is reversed and the models are fine-tuned on generic data. Does the generic model continue to only perform slightly better, which indicates that the performance is mostly based on the the amount, and not the type, of training data, or does the generic model significantly outperform the news model in that scenario?

Produced Resources

Models: https://huggingface.co/AndyReas
Data: https://huggingface.co/datasets/AndyReas/frontpage-news
Code: https://github.com/Brandeborg/Master-s-thesis-submission/tree/master/Code

References

[1] Alex Wang et al. *GLUE Benchmarks*. URL: https://gluebenchmark.com/. (accessed: April 28th, 2023).

[2] Ashish Vaswani et al. *Attention Is All You Need*. URL: https://arxiv.org/abs/1706.03762. (accessed: January 3, 2023).

[3] J. Hartmann et al. *SiEBERT - English-Language Sentiment Classification*. URL: https://huggingface.co/siebert/sentiment-roberta-large-english. (accessed: April 17th, 2023).

[4] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. URL: https://arxiv.org/abs/1810.04805. (accessed: January 3, 2023).

[5] K. Krawczyk et al. *Quantifying changes in vaccine coverage in mainstream media as a result of COVID-19 outbreak*. URL: https://www.medrxiv.org/content/10.1101/2021.11.07.21266018v1. (accessed: February 3rd, 2023).

[6] K. Krawczyk et al. *Quantifying the online news media coverage of the COVID-19 pandemic*. URL: https://www.medrxiv.org/content/10.1101/2020.12.24.20248813v1. (accessed: January 3, 2023).

[7] K. Krawczyk et al. *SciRide news data*. URL: http://sciride.org/news.html. (accessed: January 3, 2023).

[8] Lucas Theis et al. *Lossy Image Compression with Compressive Autoencoders*. URL: https://arxiv.org/abs/1703.00395. (accessed: February 14th, 2023).

[9] Renqian Luo et al. *BioGPT: generative pre-trained transformer for biomedical text generation and mining*. URL: https://academic.oup.com/bib/article/23/6/bbac409/6713511?guestAccessKey=a66d9b5d-4f83-4017-bb52-405815c907b9&login=false. (accessed: may 24th, 2023).

[10] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. URL: https://arxiv.org/abs/1907.11692. (accessed: January 3, 2023).

[11] Jason Brownlee. *A Gentle Introduction to Generative Adversarial Networks (GANs)*. URL: https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/. (accessed: February 10th, 2023).

[12] Jason Brownlee. *A Gentle Introduction to Positional Encoding in Transformer Models*. URL: https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/. (accessed: February 10th, 2023).

[13] Jason Brownlee. *An Introduction to Recurrent Neural Networks and the Math That Powers Them*. URL: https://machinelearningmastery.com/an-introduction-to-recurrent-neural-networks-and-the-math-that-powers-them/. (accessed: February 10th, 2023).

[14] Arden Dertat. *Applied Deep Learning - Part 3: Autoencoders*. URL: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798. (accessed: February 23th, 2023).

[15] Hugging Face. *Byte-Pair Encoding tokenization*. URL: https://huggingface.co/course/chapter6/5?fw=pt. (accessed: April 4th, 2023).

[16] Hugging Face. *DataCollatorForLanguageModeling*. URL: https://huggingface.co/docs/transformers/v4.27.2/en/main_classes/data_collator#transformers.DataCollatorForLanguageModeling. (accessed: April 11th, 2023).

[17] Hugging Face. *DataCollatorForTokenClassification*. URL: https://huggingface.co/docs/transformers/v4.28.1/en/main_classes/data_collator#transformers.DataCollatorForTokenClassification. (accessed: April 28th, 2023).

[18] Hugging Face. *DataCollatorWithPadding*. URL: https://huggingface.co/docs/transformers/v4.28.1/en/main_classes/data_collator#transformers.DataCollatorWithPadding. (accessed: April 28th, 2023).

[19] Hugging Face. *DistilBERT base uncased finetuned SST-2*. URL: https://huggingface.co/distilbert-base-uncased-finetuned-sst-2-english. (accessed: April 17th, 2023).

[20] Hugging Face. *GPT-2*. URL: https://huggingface.co/gpt2. (accessed: April 11th, 2023).

[21] Hugging Face. *GPT2LMHeadModel*. URL: https://huggingface.co/docs/transformers/v4.27.2/en/model_doc/gpt2#transformers.GPT2LMHeadModel. (accessed: April 11th, 2023).

[22] Hugging Face. *How to generate text: using different decoding methods for language generation with Transformers*. URL: https://huggingface.co/blog/how-to-generate. (accessed: April 16th, 2023).

[23] Hugging Face. *Huggingface Dataset Class*. URL: https://huggingface.co/docs/datasets/main/en/package_reference/main_classes#datasets.Dataset. (accessed: March 3rd, 2023).

[24] Hugging Face. *Huggingface Hub*. URL: https://huggingface.co/docs/hub/index. (accessed: March 3rd, 2023).

[25] Hugging Face. *Huggingface Trainer Class*. URL: https://huggingface.co/docs/transformers/v4.26.1/en/main_classes/trainer#transformers.Trainer. (accessed: March 3rd, 2023).

[26] Hugging Face. *Metric: accuracy*. URL: https://huggingface.co/spaces/evaluate-metric/accuracy. (accessed: April 17th, 2023).

[27] Hugging Face. *Metric: f1*. URL: https://huggingface.co/spaces/evaluate-metric/recall. (accessed: April 17th, 2023).

[28] Hugging Face. *Metric: perplexity*. URL: https://huggingface.co/spaces/evaluate-metric/perplexity. (accessed: May 25th, 2023).

[29] Hugging Face. *Metric: precision*. URL: https://huggingface.co/spaces/evaluate-metric/precision. (accessed: April 17th, 2023).

[30] Hugging Face. *Metric: recall*. URL: https://huggingface.co/spaces/evaluate-metric/f1. (accessed: April 17th, 2023).

[31] Hugging Face. *Metric: seqeval*. URL: https://huggingface.co/spaces/evaluate-metric/seqeval. (accessed: April 17th, 2023).

[32] Hugging Face. *Pipelines*. URL: https://huggingface.co/docs/transformers/main_classes/pipelines. (accessed: April 16th, 2023).

[33] Hugging Face. *RoBERTa base model*. URL: https://huggingface.co/roberta-base. (accessed: April 11th, 2023).

[34] Hugging Face. *RoBERTa large model*. URL: https://huggingface.co/roberta-large. (accessed: April 11th, 2023).

[35] Hugging Face. *RobertaForMaskedLM*. URL: https://huggingface.co/docs/transformers/v4.27.2/en/model_doc/roberta#transformers.RobertaForMaskedLM. (accessed: April 11th, 2023).

[36] Hugging Face. *RobertaForSequenceClassification*. URL: https://huggingface.co/docs/transformers/v4.28.1/en/model_doc/roberta#transformers.RobertaForSequenceCla
(accessed: April 28th, 2023).

[37] Hugging Face. *RobertaForTokenClassification*. URL: https://huggingface.co/docs/transformers/v4.28.1/en/model_doc/roberta#transformers.RobertaForTokenClassi
(accessed: April 28th, 2023).

[38] Hugging Face. *Summary of the tokenizers*. URL: https://huggingface.co/docs/transformers/main/tokenizer_summary. (accessed: April 4th, 2023).

[39] Hugging Face. *The Hugging Face Course, 2023*. URL: https://huggingface.co/course. (accessed: March 3rd, 2023).

[40] Hugging Face. *Token Classification*. URL: https://huggingface.co/docs/transformers/tasks/token_classification. (accessed: April 14th, 2023).

[41] SHRADDHA GOLED. *The rise of decoder-only Transformer models*. URL: https://analyticsindiamag.com/the-rise-of-decoder-only-transformer-models/. (accessed: February 14th, 2023).

[42] Jochen Hartmann et al. "More than a Feeling: Accuracy and Application of Sentiment Analysis". In: *International Journal of Research in Marketing* 40.1 (2023), pp. 75–87. DOI: https://doi.org/10.1016/j.ijresmar.2022.05.005. URL: https://www.sciencedirect.com/science/article/pii/S0167811622000477.

[43] Purva Huilgol. *Top 4 Sentence Embedding Techniques using Python!* URL: https://www.analyticsvidhya.com/blog/2020/08/top-4-sentence-embedding-techniques-using-python/. (accessed: February 14th, 2023).

[44] Raimi Karim. *Illustrated: Self-Attention*. URL: https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a/. (accessed: February 7th, 2023).

[45] Microsoft. *BioGPT*. URL: https://huggingface.co/microsoft/biogpt. (accessed: may 24th, 2023).

[46] Prakhar Mishra. *Understanding Masked Language Models (MLM) and Causal Language Models (CLM) in NLP*. URL: https://towardsdatascience.com/understanding-masked-language-models-mlm-and-causal-language-models-clm-in-nlp-194c15f56a5. (accessed: February 15th, 2023).

[47] Hiroki Nakayama. *seqeval: A Python framework for sequence labeling evaluation*. Software available from https://github.com/chakki-works/seqeval. 2018. URL: https://github.com/chakki-works/seqeval.

[48] NLTK. *Natural Language Toolkit*. URL: https://www.nltk.org/index.html. (accessed: March 29th, 2023).

[49] NLTK. *Porter Stemmer*. URL: https://www.nltk.org/howto/stem.html. (accessed: April 17th, 2023).

[50] OpenAI. *Aligning Language Models to Follow Instructions*. URL: https://openai.com/blog/instruction-following/. (accessed: February 22th, 2023).

[51] OpenAI. *ChatGPT: Optimizing Language Models for Dialogue*. URL: https://openai.com/blog/chatgpt/. (accessed: February 22th, 2023).

[52] OpenAI. *InstructGPT Model Card*. URL: https://github.com/openai/following-instructions-human-feedback/blob/main/model-card.md. (accessed: February 22th, 2023).

[53] Pandas. *Pandas Dataframe Documentation*. URL: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html. (accessed: March 7th, 2023).

[54] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[55] Python. *beautifulsoup4*. URL: https://pypi.org/project/beautifulsoup4/. (accessed: April 28th, 2023).

[56] Python. *HyperText Markup Language support*. URL: https://docs.python.org/3/library/html.html. (accessed: April 28th, 2023).

[57] Python. *Secure hashes and message digests*. URL: https://docs.python.org/3/library/hashlib.html. (accessed: March 27th, 2023).

[58] PyTorch. *Pytorch Documentation*. URL: https://pytorch.org/docs/stable/index.html. (accessed: March 3rd, 2023).

[59] Lance Ramshaw and Mitch Marcus. "Text Chunking using Transformation-Based Learning". In: *Third Workshop on Very Large Corpora*. 1995. URL: https://www.aclweb.org/anthology/W95-0107.

[60] Arjun Sarkar. *All you need to know about 'Attention' and 'Transformers'*. URL: https://towardsdatascience.com/all-you-need-to-know-about-attention-and-transformers-in-depth-understanding-part-1-552f0b41d021. (accessed: February 7th, 2023).

[61] David SayceDavid Sayce. *The Number of tweets per day in 2022*. URL: https://www.dsayce.com/social-media/tweets-day/. (accessed: February 3rd, 2023).

[62] sklearn. *TFIDF Transformer*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html. (accessed: March 29th, 2023).

[63] sklearn. *TFIDF Vectorizer*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html. (accessed: March 29th, 2023).

[64] TensorFlow. *Intro to Autoencoders*. URL: https://www.tensorflow.org/tutorials/generative/autoencoder. (accessed: February 14th, 2023).

[65] Andreas Thomsen. *Cleaned Project Data*. URL: https://huggingface.co/datasets/AndyReas/frontpage-news. (accessed: may 31st, 2023).

[66] Andreas Thomsen. *Project Source Code*. URL: https://github.com/Brandeborg/Master-s-thesis-submission/tree/master/Code. (accessed: may 31st, 2023).

[67] Andreas Thomsen. *Trained Project Models*. URL: https://huggingface.co/AndyReas/. (accessed: may 31st, 2023).

Appendix

Training Arguments

GenNewsGPT and roberta-gen-news

```
training_args = TrainingArguments(
    output_dir=output_dir + "/checkpoints",
    overwrite_output_dir=False,
    evaluation_strategy="steps",
    eval_steps=1_000,
    save_strategy="steps",
    save_steps=10_000,
    warmup_steps=50000,
    learning_rate= 2e-6,
    adam_beta1=0.9,
    adam_beta2=0.98,
    adam_epsilon=1e-6,
    weight_decay=0.01,
    num_train_epochs=1,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=4
)
```

NewsGPT and roberta-news

```
training_args = TrainingArguments(
    output_dir=output_dir + "/checkpoints",
    overwrite_output_dir=False,
    evaluation_strategy="steps",
    eval_steps=1_000,
    save_strategy="steps",
    save_steps=10_000,
    warmup_steps=50000,
    learning_rate= 2e-5,
    adam_beta1=0.9,
    adam_beta2=0.98,
    adam_epsilon=1e-6,
    weight_decay=0.01,
    num_train_epochs=6, # stopped at 4
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    gradient_accumulation_steps=4
)
```

Fine-tuning

```
    training_args = TrainingArguments(
        output_dir=save_dir + "/checkpoints",
        overwrite_output_dir=True,
        learning_rate=2e-5,
        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        num_train_epochs=num_train_epochs, # 3 for token classification,
                                             otherwise 1
        weight_decay=0.01,
        evaluation_strategy="epoch",
        save_strategy="epoch",
        load_best_model_at_end=False,
        seed=seed)
```

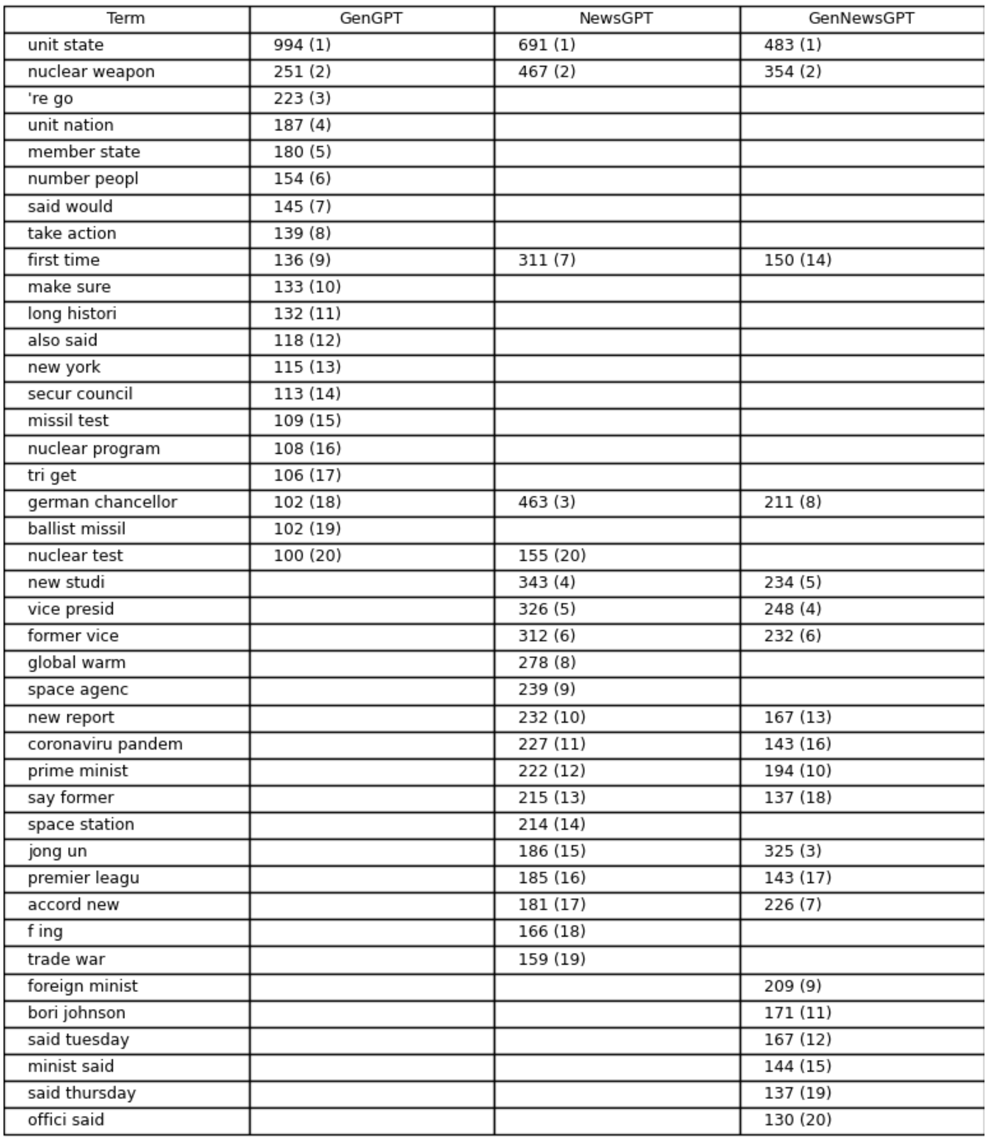
Prompt topics

```
{
    "Donald Trump": {
        "prompts": [
            "Donald Trump",
            "Donald Trump is",
            "Donald Trump has",
            "Donald Trump will"
        ]
    },
    "Joe Biden": {
        "prompts": [
            "Joe Biden",
            "Joe Biden is",
            "Joe Biden has",
            "Joe Biden will"
        ]
    },
    "Angela Merkel": {
        "prompts": [
            "Angela Merkel",
            "Angela Merkel is",
            "Angela Merkel has",
            "Angela Merkel will"
        ]
    },
    "Vladimir Putin": {
        "prompts": [
            "Vladimir Putin",
            "Vladimir Putin is",
            "Vladimir Putin has",
            "Vladimir Putin will"
        ]
    },
    "Kim Jong-Un": {
        "prompts": [
            "Kim Jong-Un Putin",
            "Kim Jong-Un Putin is",
            "Kim Jong-Un Putin has",
            "Kim Jong-Un Putin will"
        ]
    },
    "COVID-19": {
        "prompts": [
            "COVID-19",
            "COVID-19 is",
            "COVID-19 has",
            "COVID-19 will"
        ]
    },
    "The vaccine": {
        "prompts": [
            "The vaccine",
            "The vaccine is",
            "The vaccine has",
            "The vaccine will"
        ]
    },
    "Climate change": {
        "prompts": [
            "Climate change",
            "Climate change is",
            "Climate change has",
            "Climate change will"
        ]
    },
    "Natural disasters": {
        "prompts": [
            "Natural disasters",
            "Natural disasters is",
            "Natural disasters has",
            "Natural disasters will"
        ]
    },
    "Dogs": {
        "prompts": [
            "Dogs",
            "Dogs are",
            "Dogs have",
            "Dogs will"
        ]
    },
    "Football fans": {
        "prompts": [
            "Football fans",
            "Football fans are",
            "Football fans have",
            "Football fans will"
        ]
    },
    "The NBA": {
        "prompts": [
            "The NBA",
            "The NBA is",
            "The NBA has",
            "The NBA will"
        ]
    },
    "NASA": {
        "prompts": [
            "NASA",
            "NASA is",
            "NASA has",
            "NASA will"
        ]
    },
    "Artificial intelligence": {
        "prompts": [
            "Artificial intelligence",
            "Artificial intelligence is",
            "Artificial intelligence has",
            "Artificial intelligence will"
        ]
    },
    "World hunger": {
        "prompts": [
            "World hunger",
            "World hunger is",
            "World hunger has",
            "World hunger will"
        ]
    },
    "The US": {
        "prompts": [
            "The US",
            "The US is",
            "The US has",
            "The US will"
        ]
    },
    "Russia": {
        "prompts": [
            "Russia",
            "Russia is",
            "Russia has",
            "Russia will"
        ]
    },
    "China": {
        "prompts": [
            "China",
            "China is",
            "China has",
            "China will"
        ]
    },
    "North Korea": {
        "prompts": [
            "North Korea",
            "North Korea is",
            "North Korea has",
            "North Korea will"
        ]
    },
    "South Korea": {
        "prompts": [
            "South Korea",
            "South Korea is",
            "South Korea has",
            "South Korea will"
        ]
    },
    "The EU": {
        "prompts": [
            "The EU",
            "The EU is",
            "The EU has",
            "The EU will"
        ]
    },
    "The UK": {
        "prompts": [
            "The EU",
            "The EU is",
            "The EU has",
            "The EU will"
        ]
    },
    "Denmark": {
        "prompts": [
            "Denmark",
            "Denmark is",
            "Denmark has",
            "Denmark will"
        ]
    }
}
```

Multi-label topics and keywords

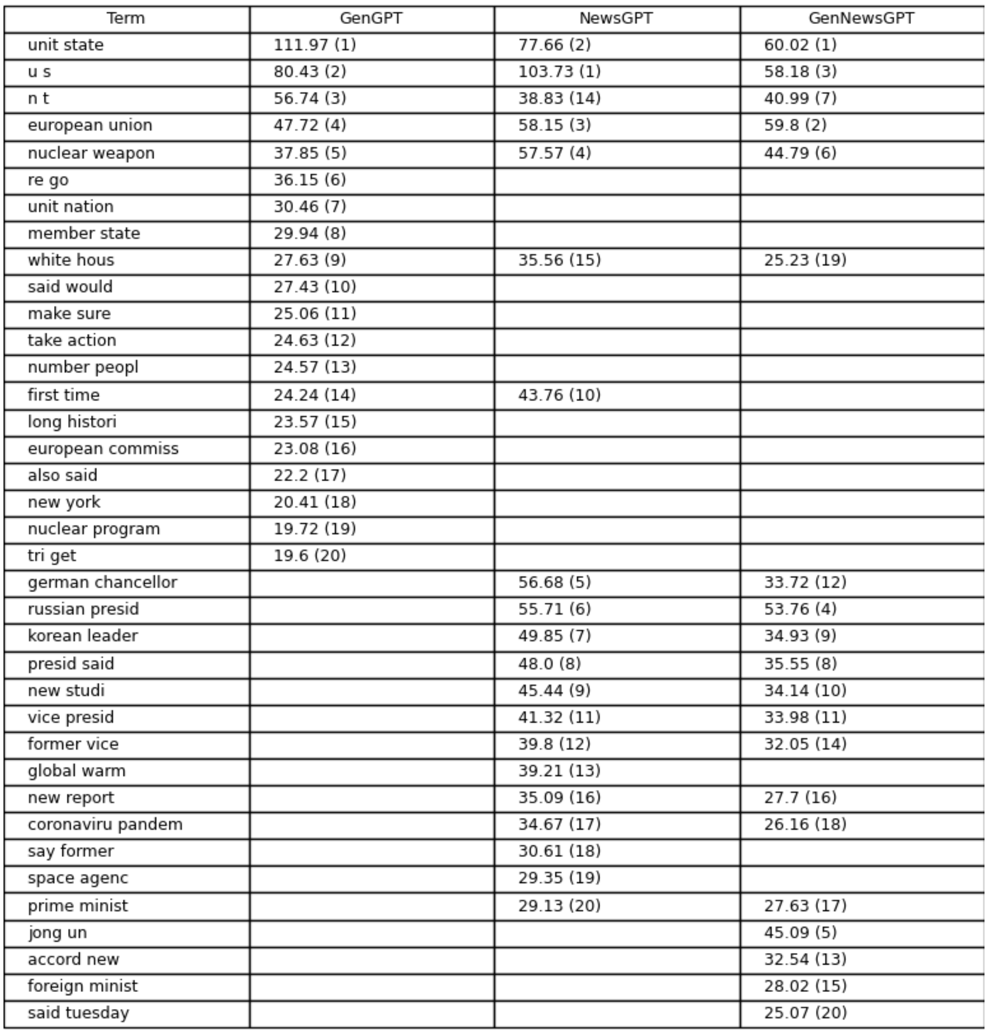
```
{
    "donald trump": {"keywords_stemmed": ["donald trump", "
       presid trump"], "keywords": ["donald trump", "president
        trump", "trump"]},
    "boris johnson": {"keywords_stemmed": ["bori johnson"], "
       keywords": ["boris johnson", "johnson"]},
    "hillary clinton": {"keywords_stemmed": ["hillari clinton"
       ], "keywords": ["hillary clinton", "clinton"]},
    "joe biden": {"keywords_stemmed": ["joe biden", "joseph
       biden", "presid biden"], "keywords": ["joe biden", "
       joseph biden", "biden"]},
    "barack obama": {"keywords_stemmed": ["barack obama", "
       presid obama"], "keywords": ["barack obama", "obama"]},
    "bernie sanders": {"keywords_stemmed": ["berni sander", "
       bernard sander"], "keywords": ["bernie sanders", "
       bernard sanders", "bernie", "sanders"]},
    "angela merkel": {"keywords_stemmed": ["angela merkel"], "
       keywords": ["angela merkel", "merkel"]},
    "emmanuel macron": {"keywords_stemmed": ["emmanuel macron"
       , "presid macron"], "keywords": ["emmanuel macron", "
       macron"]},
    "vladimir putin": {"keywords_stemmed": ["vladimir putin"],
        "keywords": ["vladimir putin", "putin"]},
    "kim jong-un": {"keywords_stemmed": ["kim jong-un"], "
       keywords": ["kim jong-un"]},
    "justin trudeau": {"keywords_stemmed": ["justin trudeau"],
        "keywords": ["justin trudeau", "trudeau"]}
}
```

Term Frequency Tables



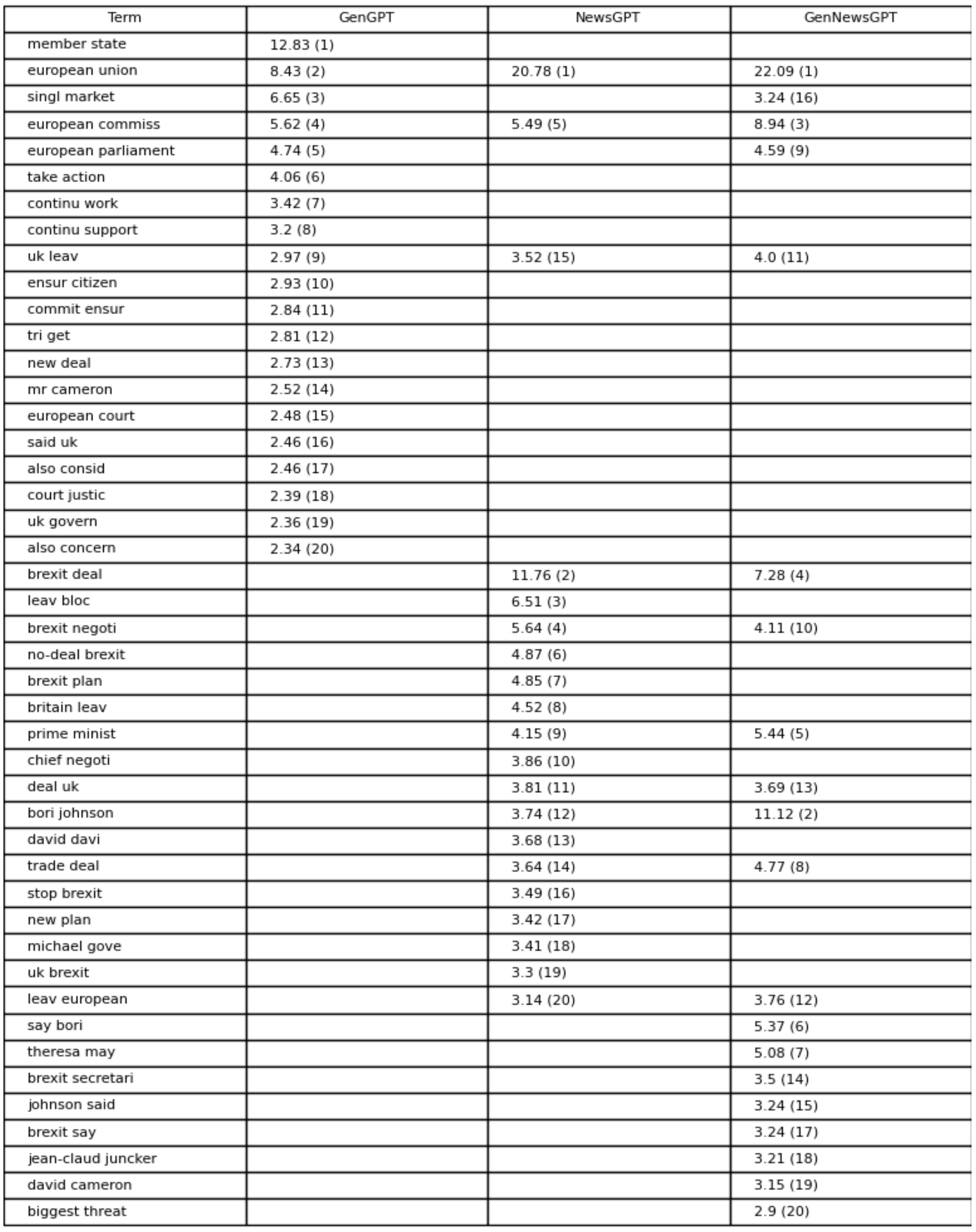
Term	GenGPT	NewsGPT	GenNewsGPT
unit state	994 (1)	691 (1)	483 (1)
nuclear weapon	251 (2)	467 (2)	354 (2)
're go	223 (3)		
unit nation	187 (4)		
member state	180 (5)		
number peopl	154 (6)		
said would	145 (7)		
take action	139 (8)		
first time	136 (9)	311 (7)	150 (14)
make sure	133 (10)		
long histori	132 (11)		
also said	118 (12)		
new york	115 (13)		
secur council	113 (14)		
missil test	109 (15)		
nuclear program	108 (16)		
tri get	106 (17)		
german chancellor	102 (18)	463 (3)	211 (8)
ballist missil	102 (19)		
nuclear test	100 (20)	155 (20)	
new studi		343 (4)	234 (5)
vice presid		326 (5)	248 (4)
former vice		312 (6)	232 (6)
global warm		278 (8)	
space agenc		239 (9)	
new report		232 (10)	167 (13)
coronaviru pandem		227 (11)	143 (16)
prime minist		222 (12)	194 (10)
say former		215 (13)	137 (18)
space station		214 (14)	
jong un		186 (15)	325 (3)
premier leagu		185 (16)	143 (17)
accord new		181 (17)	226 (7)
f ing		166 (18)	
trade war		159 (19)	
foreign minist			209 (9)
bori johnson			171 (11)
said tuesday			167 (12)
minist said			144 (15)
said thursday			137 (19)
offici said			130 (20)

Figure 32: Table of Term Frequency analysis results for generated text in all topics. The number in parentheses in each cell is the term's rank within the associated model's top 20. (TF)



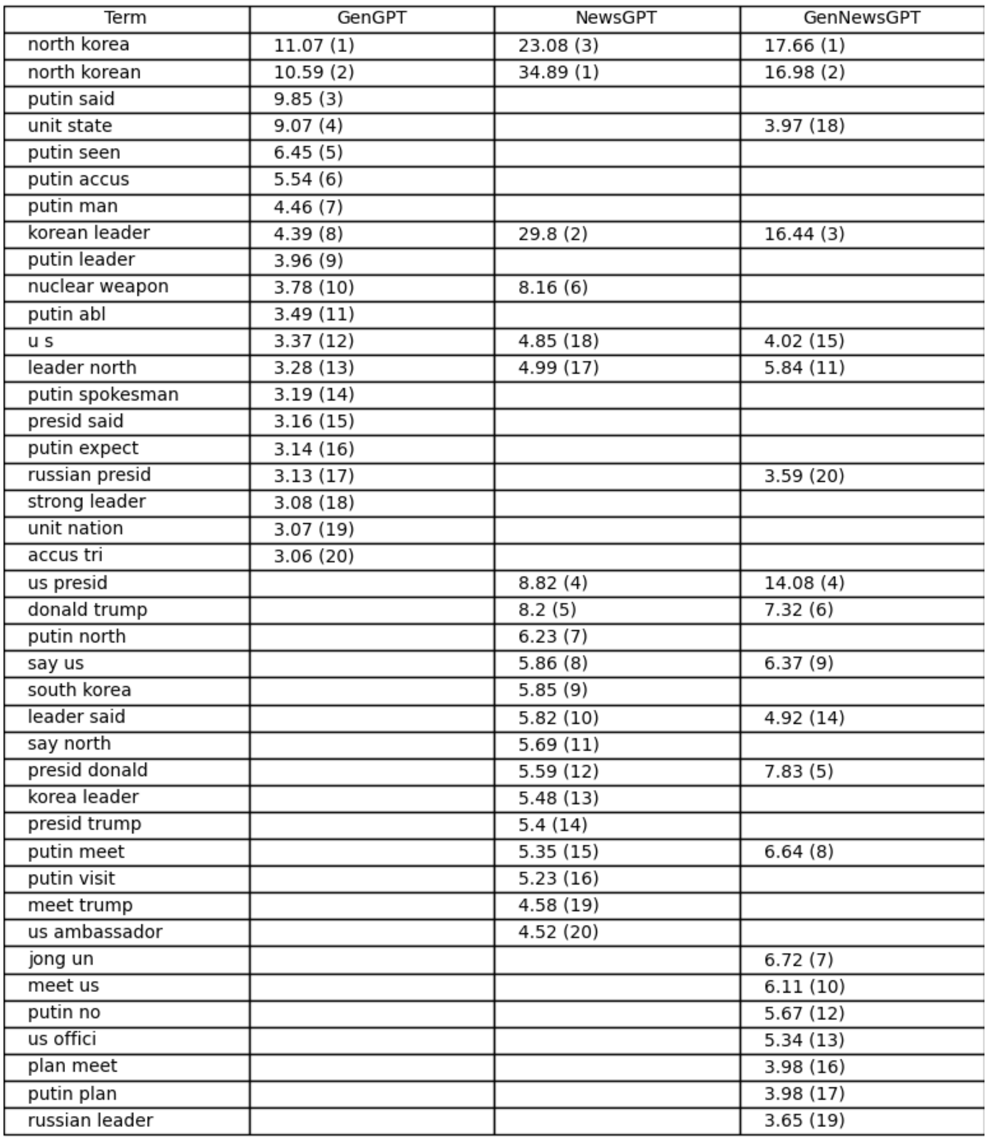
Term	GenGPT	NewsGPT	GenNewsGPT
unit state	111.97 (1)	77.66 (2)	60.02 (1)
u s	80.43 (2)	103.73 (1)	58.18 (3)
n t	56.74 (3)	38.83 (14)	40.99 (7)
european union	47.72 (4)	58.15 (3)	59.8 (2)
nuclear weapon	37.85 (5)	57.57 (4)	44.79 (6)
re go	36.15 (6)		
unit nation	30.46 (7)		
member state	29.94 (8)		
white hous	27.63 (9)	35.56 (15)	25.23 (19)
said would	27.43 (10)		
make sure	25.06 (11)		
take action	24.63 (12)		
number peopl	24.57 (13)		
first time	24.24 (14)	43.76 (10)	
long histori	23.57 (15)		
european commiss	23.08 (16)		
also said	22.2 (17)		
new york	20.41 (18)		
nuclear program	19.72 (19)		
tri get	19.6 (20)		
german chancellor		56.68 (5)	33.72 (12)
russian presid		55.71 (6)	53.76 (4)
korean leader		49.85 (7)	34.93 (9)
presid said		48.0 (8)	35.55 (8)
new studi		45.44 (9)	34.14 (10)
vice presid		41.32 (11)	33.98 (11)
former vice		39.8 (12)	32.05 (14)
global warm		39.21 (13)	
new report		35.09 (16)	27.7 (16)
coronaviru pandem		34.67 (17)	26.16 (18)
say former		30.61 (18)	
space agenc		29.35 (19)	
prime minist		29.13 (20)	27.63 (17)
jong un			45.09 (5)
accord new			32.54 (13)
foreign minist			28.02 (15)
said tuesday			25.07 (20)

Figure 33: Table of Term Frequency analysis results for generated text in all topics. The number in parentheses in each cell is the term's rank within the associated model's top 20. (TF-IDF)



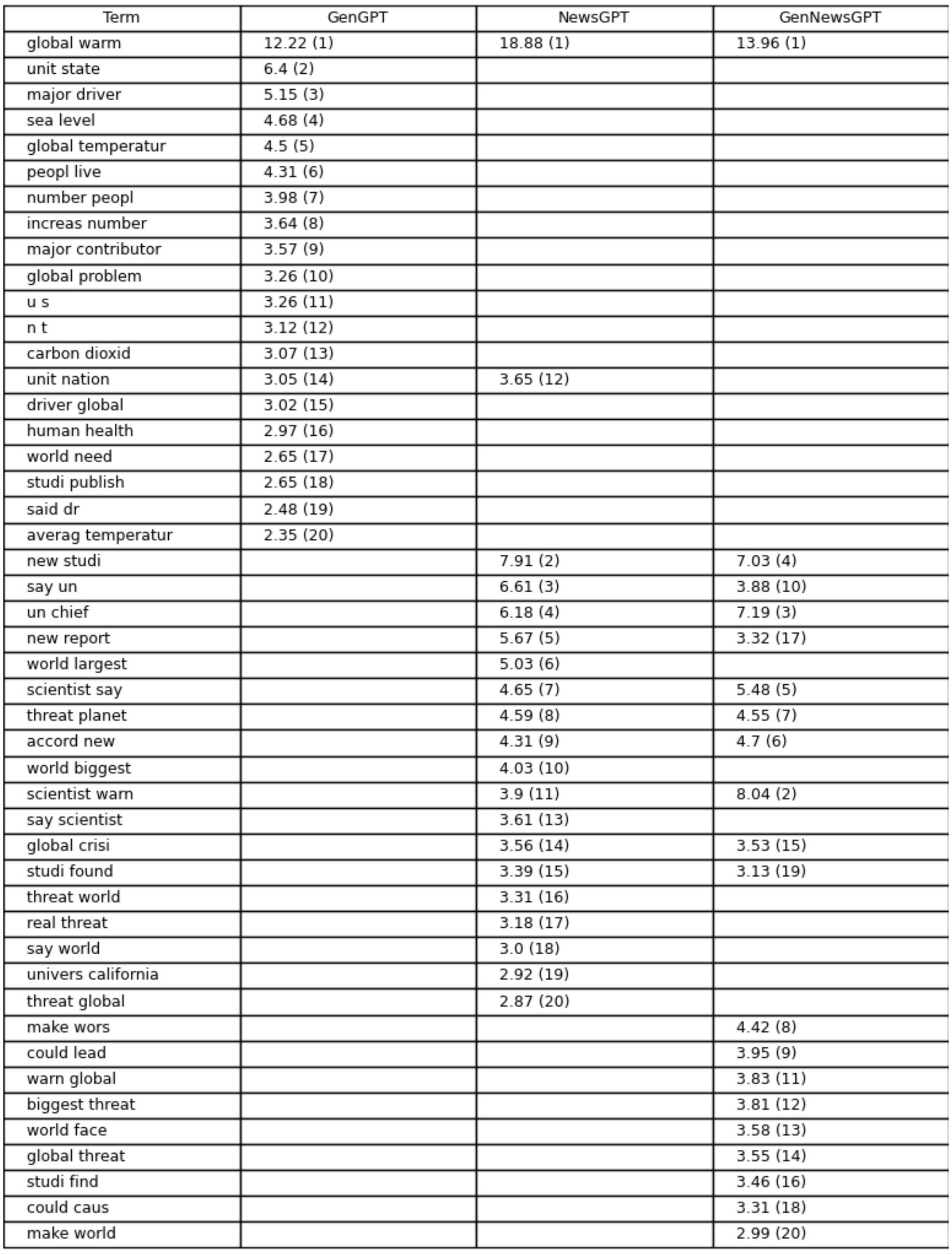
Term	GenGPT	NewsGPT	GenNewsGPT
member state	12.83 (1)		
european union	8.43 (2)	20.78 (1)	22.09 (1)
singl market	6.65 (3)		3.24 (16)
european commiss	5.62 (4)	5.49 (5)	8.94 (3)
european parliament	4.74 (5)		4.59 (9)
take action	4.06 (6)		
continu work	3.42 (7)		
continu support	3.2 (8)		
uk leav	2.97 (9)	3.52 (15)	4.0 (11)
ensur citizen	2.93 (10)		
commit ensur	2.84 (11)		
tri get	2.81 (12)		
new deal	2.73 (13)		
mr cameron	2.52 (14)		
european court	2.48 (15)		
said uk	2.46 (16)		
also consid	2.46 (17)		
court justic	2.39 (18)		
uk govern	2.36 (19)		
also concern	2.34 (20)		
brexit deal		11.76 (2)	7.28 (4)
leav bloc		6.51 (3)	
brexit negoti		5.64 (4)	4.11 (10)
no-deal brexit		4.87 (6)	
brexit plan		4.85 (7)	
britain leav		4.52 (8)	
prime minist		4.15 (9)	5.44 (5)
chief negoti		3.86 (10)	
deal uk		3.81 (11)	3.69 (13)
bori johnson		3.74 (12)	11.12 (2)
david davi		3.68 (13)	
trade deal		3.64 (14)	4.77 (8)
stop brexit		3.49 (16)	
new plan		3.42 (17)	
michael gove		3.41 (18)	
uk brexit		3.3 (19)	
leav european		3.14 (20)	3.76 (12)
say bori			5.37 (6)
theresa may			5.08 (7)
brexit secretari			3.5 (14)
johnson said			3.24 (15)
brexit say			3.24 (17)
jean-claud juncker			3.21 (18)
david cameron			3.15 (19)
biggest threat			2.9 (20)

Figure 34: Table of Term Frequency analysis results for generated text in one topic (the EU). The number in parentheses in each cell is the term's rank within the associated model's top 20. (TF-IDF)



Term	GenGPT	NewsGPT	GenNewsGPT
north korea	11.07 (1)	23.08 (3)	17.66 (1)
north korean	10.59 (2)	34.89 (1)	16.98 (2)
putin said	9.85 (3)		
unit state	9.07 (4)		3.97 (18)
putin seen	6.45 (5)		
putin accus	5.54 (6)		
putin man	4.46 (7)		
korean leader	4.39 (8)	29.8 (2)	16.44 (3)
putin leader	3.96 (9)		
nuclear weapon	3.78 (10)	8.16 (6)	
putin abl	3.49 (11)		
u s	3.37 (12)	4.85 (18)	4.02 (15)
leader north	3.28 (13)	4.99 (17)	5.84 (11)
putin spokesman	3.19 (14)		
presid said	3.16 (15)		
putin expect	3.14 (16)		
russian presid	3.13 (17)		3.59 (20)
strong leader	3.08 (18)		
unit nation	3.07 (19)		
accus tri	3.06 (20)		
us presid		8.82 (4)	14.08 (4)
donald trump		8.2 (5)	7.32 (6)
putin north		6.23 (7)	
say us		5.86 (8)	6.37 (9)
south korea		5.85 (9)	
leader said		5.82 (10)	4.92 (14)
say north		5.69 (11)	
presid donald		5.59 (12)	7.83 (5)
korea leader		5.48 (13)	
presid trump		5.4 (14)	
putin meet		5.35 (15)	6.64 (8)
putin visit		5.23 (16)	
meet trump		4.58 (19)	
us ambassador		4.52 (20)	
jong un			6.72 (7)
meet us			6.11 (10)
putin no			5.67 (12)
us offici			5.34 (13)
plan meet			3.98 (16)
putin plan			3.98 (17)
russian leader			3.65 (19)

Figure 35: Table of Term Frequency analysis results for generated text in one topic (Kim Jong-Un). The number in parentheses in each cell is the term's rank within the associated model's top 20. (TF-IDF)



Term	GenGPT	NewsGPT	GenNewsGPT
global warm	12.22 (1)	18.88 (1)	13.96 (1)
unit state	6.4 (2)		
major driver	5.15 (3)		
sea level	4.68 (4)		
global temperatur	4.5 (5)		
peopl live	4.31 (6)		
number peopl	3.98 (7)		
increas number	3.64 (8)		
major contributor	3.57 (9)		
global problem	3.26 (10)		
u s	3.26 (11)		
n t	3.12 (12)		
carbon dioxid	3.07 (13)		
unit nation	3.05 (14)	3.65 (12)	
driver global	3.02 (15)		
human health	2.97 (16)		
world need	2.65 (17)		
studi publish	2.65 (18)		
said dr	2.48 (19)		
averag temperatur	2.35 (20)		
new studi		7.91 (2)	7.03 (4)
say un		6.61 (3)	3.88 (10)
un chief		6.18 (4)	7.19 (3)
new report		5.67 (5)	3.32 (17)
world largest		5.03 (6)	
scientist say		4.65 (7)	5.48 (5)
threat planet		4.59 (8)	4.55 (7)
accord new		4.31 (9)	4.7 (6)
world biggest		4.03 (10)	
scientist warn		3.9 (11)	8.04 (2)
say scientist		3.61 (13)	
global crisi		3.56 (14)	3.53 (15)
studi found		3.39 (15)	3.13 (19)
threat world		3.31 (16)	
real threat		3.18 (17)	
say world		3.0 (18)	
univers california		2.92 (19)	
threat global		2.87 (20)	
make wors			4.42 (8)
could lead			3.95 (9)
warn global			3.83 (11)
biggest threat			3.81 (12)
world face			3.58 (13)
global threat			3.55 (14)
studi find			3.46 (16)
could caus			3.31 (18)
make world			2.99 (20)

Figure 36: Table of Term Frequency analysis results for generated text in one topic (Climate Change). The number in parentheses in each cell is the term's rank within the associated model's top 20. (TF-IDF)