WILFRID LAURIER UNIVERSITY

DEPARTMENT OF MATHEMATICS

MA695 - MAJOR RESEARCH PROJECT

Enhancing Low-Resource Named Entity Recognition Results with Data Augmentation via Abstractive Summarization

---

Author
Douglas BOWEN

Supervisors:
Dr. Yang LIU
Dr. Xu (Sunny) WANG

Second Readers:
Ilias S. KOTSIREAS

July 24th, 2022

Abstract

News and other articles can provide valuable info - but need to be read, analyzed, and processed. In the past, this was done manually and could be costly and time-consuming. In recent years, Natural Language Processing (NLP) has become mainstream in its ability to extract a variety of information from structured and unstructured text, without human intervention. This provides a much quicker and more cost-effective tool than manual labour.

Named Entity Recognition (NER) is one such information extraction method, allowing for the classification of named entities to be made automatically. Due to its unique characteristics and the large variance in different domains, NER has been widely studied in the past few decades.

For models to extract named entities from articles, a large amount of NER labelled data is required to learn from and sufficiently make predictions. However, finding data that is labelled can be challenging. Though some common tag sets and domains have plenty of data available to train on already that have been compiled by the public (e.g. Wikipedia, Twitter, News), custom NER tags and niche domains often have little to no data available, making training quite challenging. As a result, steps must be taken to create a large dataset from scratch or a small low-resource set. Data augmentation (DA) is a technique that attempts to alleviate this. Through various approaches, individuals can create artificial data to supplement the original and help increase NER model performance without the need for costly human data curation.

This project examines various approaches to data augmentation in low-resource domains wherein artificially produced training data is required for decent model training. The dataset simulates a low-resource scenario for data augmentation, and a new generation approach using abstractive summarization is used to create sentences with completely new structures from the original data.

Techniques to create a sufficient sample set of augmented NE data include (but are not limited to) weighting source article sentences by entity count prior to shuffling, cutting sentences with no entities, and shuffling sentences randomly to create new samples.

Results show that utilizing an abstractive summarization technique to augment data provides a significant boost over the original unaugmented data for low-resource data. Summary results are similar to paraphrased results, but lower than rules-based results. As the data size increases, the boost provided diminishes for all methods as expected.

Acknowledgements

I would like to extend my sincere thanks to everyone who provided support during this project. Firstly, to my advisors Dr. Xu (Sunny) Wang and Dr. Yang Liu, for supporting my efforts and for providing their expertise as well as helping guide my research with valuable suggestions and insights. Secondly, to my colleagues Yanan Chen, Ian Fraser, Jason Lechter, and Herteg Kohar who worked with me, helped in researching, and provided support in tackling issues throughout the project. Lastly, I would also like to acknowledge the funding that was provided by the Natural Science and Engineering Research Council (NSERC), the MITACS Accelerate Program, and The Quantum Insider (TQI).

Contents

List of Tables

List of Figures

Chapter 1: Introduction

In the digital age news comes out at an incredible rate. Be it live tweets, online articles, published research, or anything in-between, so much information is coming in at one time and has become impossible for humans to process it all.

In some domains (such as quantum technology), the amount of data has steadily increased with the number of news releases growing every day. Subsequently, information from these releases can often require domain expertise and thus cannot be quickly understood without sufficient background research.

Given the timely and costly manual labour required to review and research for a proper understanding in these domains, along with the general size of training data required for NLP tasks, data augmentation (DA) is a growing field of research that aims to lessen said burden. DA utilizes a variety of techniques that create "new" artificially created data points from the original data. However, Named Entity Recognition (NER) is one such information extraction task that has more stringent requirements in both the structure of the data and the information required from models. For NER tasks, the structure of a sentence is considered and from it, words are designated either as an entity or not. The sequence of the sentence provides the information to designate an entity and is thus important to maintain. As a result, general DA techniques that have been developed for other tasks tend not to apply as well to NER ones due to the contextual requirements.

This project aims to examine data augmentation techniques that can be utilized for NER models in low-resource domains. Though entity classifications are not complicated, the information surrounding them can be confusing for computers. Examining the following snippet from a quantum technology article, "demonstrating the early application of NISQ solutions to the energy sector", a model would likely consider NISQ to be a company rather than scientific acronym if it was not trained in the domain. An augmented version using generation could alter the sentence to the form "NISQ solutions demonstrated promising results within the energy sector" which is still coherent in structure as required by NER models and yet provides new sentence structure to learn on. By examining and applying DA techniques, the time and subsequent cost that arise from requiring individuals with sufficient expertise to provide labelled data can be reduced significantly. With this information, inferences and changes in the industry can be more readily tracked without constant monitoring.

The new augmentation method being examined relies on generating abstractive summaries as a means of creating sentences with new structure as compared to the original source article. Furthermore, approaches are taken to attempt giving priority to entities in the generated sentences. By utilizing summarization models, the generated text will hopefully contain a proper coherent structure with entities from the original text that is sufficiently unique when compared to the other sentences from the original article. To compare the success of this approach, a baseline will be constructed using other researched methods that have been proven successful in augmenting NER sentences. These are discussed in detail in the literature review.

The research performed here will highlight the performance of abstractive summarization as compared to the original dataset, as well as compared to baselines via rule-based approach and a paraphrasing generative method. Results show that performance of abstractive summarization is quite similar to that of paraphrasing, but below that of the rule-based approach for X-Small, Small, and Medium size batches while the Large batch size has similar performance for all three.

The remainder of this thesis is organized in the following order: Chapter 2, which covers the literature review on relevant research done by others; Chapter 3, which provides background information on underlying mechanisms; Chapter 4, which discusses the specific approach to be taken; Chapter 5, which presents the results of the research; Chapter 6, which discusses potential further work; and Chapter 7, which summarizes the paper.

Chapter 2: Literature Review

This section will address the literature and published research currently available that makes use of data augmentation (DA) techniques on some Named Entity Recognition (NER) process. Though the focus is on utilizing DA for in-domain, low-resource data, the literature of utilizing DA for other purposes is also examined.

2.1 NER-Focused Approach

Though many DA techniques have been proposed for general NLP tasks, the application of these DA techniques is not always possible in the case of NER. This is due to the challenge that NER models present, wherein the labelling process is based on each individual token within sentences.

Critical to understanding NER DA methods, it is important to familiarize oneself with the required structure for NER data. There are three primary points of importance: (1) The sentence of a structure, used to gain insight through learned patterns, (2) The words or "tokens" within a sentence that may or may not be entities, and (3) the associated BIO-labels, indicating either the (B)eginning/(I)nside of a named entity, or (O)utside, for non-entities. Of course, there is also the variety of named-entity categories (entity types) associated with the aforementioned labels, however these can be selected as one sees fit and are non-standard across NER tasks. As a result of these tags, techniques that might transform a token have to avoid also changing their labelling which presents a unique challenge for DA in NER tasks.

2.2 Rule-Based Approaches

The simplest approach is one that focused on tweaking traditional rules-based NLP approaches to DA so that they may be applied to NER situations [1]. Four underlying rules-based methods for augmenting NER data were constructed, for which combinations of these methods could also be utilized. Each of the following methods utilizes a probability distribution to determine whether or not the suggested transformation should take place on the specified token/segment. A success rate p is utilized in the binomial distribution (specifically, the bernoulli distribution for n=1 trial).

- The first method discussed is Label-Wise Token Replacement (LWTR) [1] wherein each token within a sentence is evaluated randomly for success or failure. If the randomly generated number outputs a success, the token is replaced with a randomly selected token designated with the same label. Otherwise, no replacement occurs and the next token is evaluated in the same way.

- The second method discussed is Synonym Replacement (SR) [1]. In a very similar method as to LWTR, each token within a sentence is evaluated randomly. In the case of SR however, if the randomly generated result indicates a success, the token is replaced with a synonym of itself. For synonyms that are multi-word, the associated label is applied in the standard B-Entity, I-Entity order.

- The third method discussed is Mention Replacement (MR) [1]. If a token is marked replacement, an entry with the same entity type is swapped into its place. The BIO-label for the new token being replaced should be maintained (i.e. the label should also be replaced).

- The final method discussed is substantially different from LWTR [1], SR, and MR. Shuffle within Segment (SIS) evaluates each continuous segment of labels within a sentence randomly for success or not. In the case of success, the segment will be randomly shuffled to alter the order of tokens, but not labels.

Lastly, it is possible to use any combination/mixture of the above methods for a more comprehensive approach to data augmentation. Figure 2.1

highlights these methods with an example provided by the original researchers.

	Instance												
None	She	did	not	complain	of	headache	or	any	other	neurological	symptoms	.	
	O	O	O	O	O	B-problem	O	B-problem	I-problem	I-problem	I-problem	O	
LwTR	L.	One	not	complain	of	headache	he	any	interatrial	neurological	current	.	
	O	O	O	O	O	B-problem	O	B-problem	I-problem	I-problem	I-problem	O	
SR	She	did	non	complain	of	headache	or	whatsoever	former	neurologic	symptom	.	
	O	O	O	O	O	B-problem	O	B-problem	I-problem	I-problem	I-problem	O	
MR	She	did	not	complain	of	neuropathic	pain	syndrome	or	acute	pulmonary	disease	.
	O	O	O	O	O	B-problem	I-problem	I-problem	O	B-problem	I-problem	I-problem	O
SiS	not	complain	She	did	of	headache	or	neurological	any	symptoms	other	.	
	O	O	O	O	O	B-problem	O	B-problem	I-problem	I-problem	I-problem	O	

Figure 2.1: Rule-Based Method Examples [1]

2.3 Basic Generation Approach

Another recent approach to augmenting NER data utilizes generational language models that are trained on linearized labelled sentences [2]. This method was applied to both supervised and semi-supervised data wherein both resulted in performances above baseline. Unlike the rules-based approach which simply applies adjustments to data without significantly altering them, generation-based approaches attempt to create sentences that have completely new structures with different words, contexts, phrases, etc., while still remaining coherent to the human eye. Thus the generated sentences still share similar patterns and trends to the original data that a model can find.

Prior to utilizing a language model, sentences have tags added to the beginning and end of sentences, denoted as [BOS] and [EOS] respectively. Furthermore, named entities have their labels added in front of their respective tokens to linearize the sentence. After the completion of this process, a language model can be utilized. In this instance, a one-layer Long-

Short Term Memory (LSTM) Recurrent Neural Network (RNN) is chosen as the language model. To train the generation model, each token is fed into the model in a linearized sentence order such that the first entry is [BOS] and the last entry is [EOS]. Each token then encounters a dropout layer, LSTM layer, another dropout later, and finally produces a prediction for the next token that should occur. With the trained model, synthetic data is generated by first feeding a [BOS] tag into the LSTM RNN, which then outputs a prediction. This prediction is then fed into the model as the next token in the sentence, and so on until sentence completion. This generated sentence is then de-linearized, with labels being added back into the proper order and [BOS]/[EOS] being removed.

Another generation approach, albeit a bit more involved, is one that utilizes a Sequence-to-Sequence (Seq2Seq) Language Model to back-translate a sentence [4]. As a simple overview, the process involves first splitting a sequence of tokens so as to break it into continuous label segments. Then, segments of 3 tokens or more that do not correspond to an entity type are fed into the model for translation from English to German and back again. The result of this back-translation is a sentence slightly different from the original in most case due to how each model handles the translation.

2.4 Cross-Domain Generation Approach

One method takes a different approach from the aforementioned methods. Wherein those methods focus on data augmentation in low-resource scenarios that are common in niche fields such as Quantum Technology, this approach attempts to leverage data from a high-resource domain and subsequently project it into the low-resource domain by utilizing semantics and patterns inherent to all text, even if textual patterns differ [3]. This approach builds off the work done in the generation approach.

Similarly to the generation approach, this method begins by linearizing sentences, but then pairs a "source" domain sentence with a "target" domain sentence. Each sentence then has noise added through shuffling, dropouts, or masking. Once this is done, the domain-sentence pairs are fed into the model to output into the paired domain, which allows for the models to learn a "mapping" between domains.

2.5 Paraphrasing

A final method is the use of paraphrasing (expressing a sentence's meaning with different word structure) to augment NER data. A Bidirectional Encoder Representations from Transformers (BERT) model is utilized for entity recognition, while the underlying model utilized for the paraphrasing is not present in the published article. Researchers first replaced tokens with their respective entity tags (not "O" tags) and then performed both paraphrasing and back-translation on the sentence to generate newly augmented sentences. Both methods (paraphrasing vs. back-translation) were compared against one another, with paraphrasing proving successful at improving the BERT model at *very* low resource data levels [13].

Chapter 3: Background

This chapter provides a generalized overview of concepts not formally found in papers, but that are relevant to the research conducted henceforth.

3.1 Evaluation Metrics

Precision, Recall, & F1

These are standard metrics used for evaluating mistakes made by models. Precision measures the rate of false-positives, recall measures the rate of false-negatives, and F1 showcases the performance taking into account precision and recall trade-offs.

Recall-Oriented Understudy for Gisting Evaluation

More commonly known as "ROUGE", this is another important metric that will be used, albeit only in baseline construction and testing of the pre-trained Seq2Seq model. ROUGE evaluates the effectiveness of a summary or translation as compared to a human-produced reference.

3.2 Common Model Structure

Seq2Seq models are also commonly referred to as encoder-decoder models. Encoders receive inputs (for example, tokens within a sentence) and use said inputs to find relationships and acquire an "understanding" of the input. This understanding comes in the form of numerical outputs, which are then sent to the decoder and called "context vectors". Once encoding is complete, the context vectors along with a starting token are passed into a decoder. Decoders then read the context vector and starting token inputs and try to predict outputs, token by token. Figure 3.1 and 3.2 display a very basic Seq2Seq overview. The encoder and decoder segments are in actuality often both built on recurrent neural networks, which are either Gated Recurrent Unit (GRU) or Long-Short Term Memory (LSTM) networks. However, this is not the case for transformers.

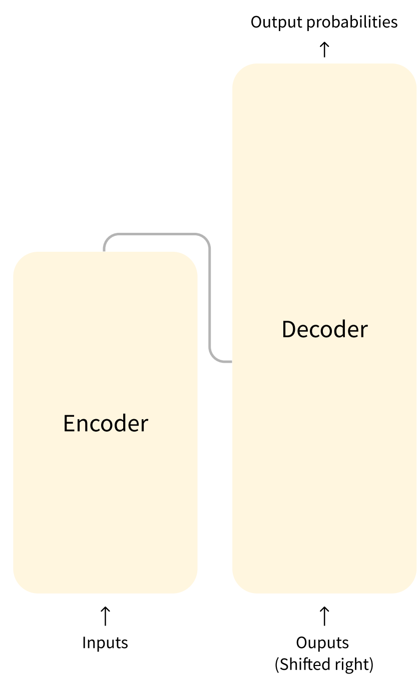


Figure 3.1: Basic Seq2Seq Model Architecture [20]

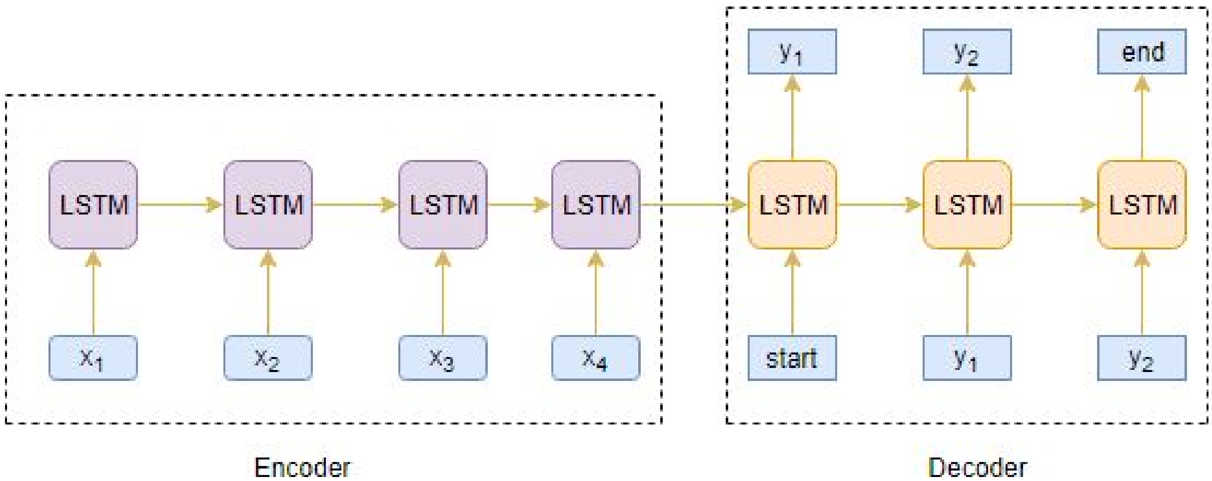


Figure 3.2: Encoder-Decoder Details [20]

3.3 Transformers

Transformers are built with "attention" in mind. Attention is a method wherein encoders and decoders are fed only the relevant inputs. To denote these inputs, inputs are assigned weightings where higher weights denote higher importance. These weightings are adjusted over time using feed-forward neural networks. Overall, transformers try to predict an output using only the important parts of the sentence, therefore giving a higher degree of "attention" to important input words over the others. The transformer process is very involved and thus more easily visualized as in Figure 3.3.

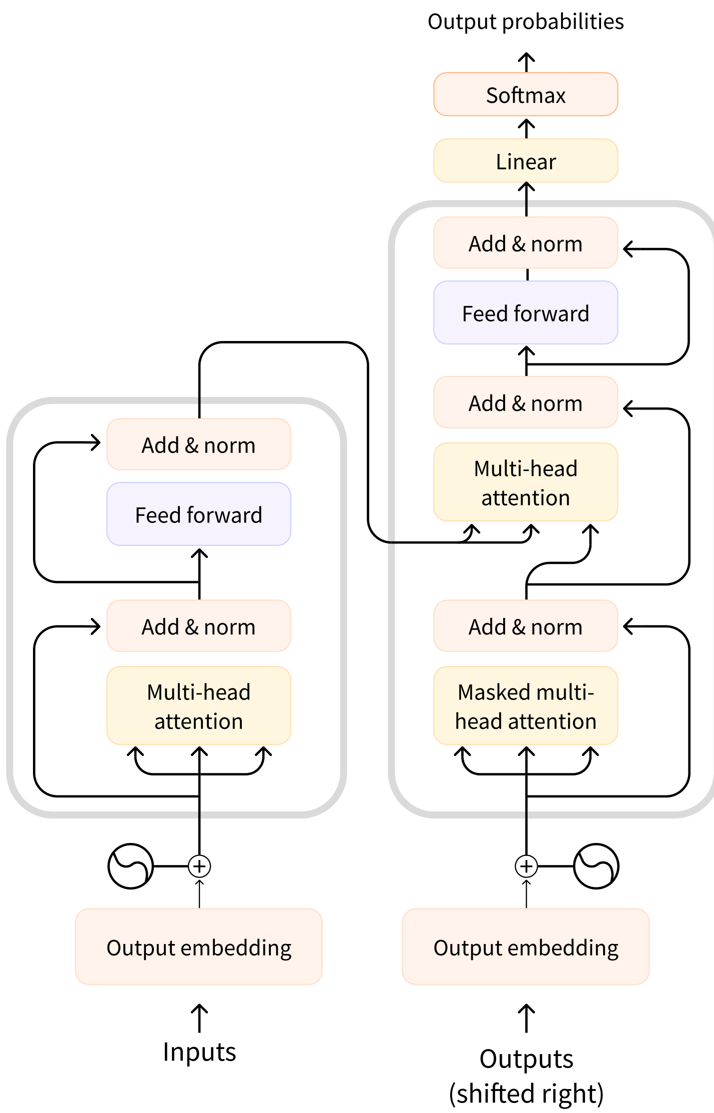


Figure 3.3: Transformer Encoder-Decoder Architecture [20]

3.4 Model Hyper-Parameter Choices

HuggingFace provides a strong API with a large array of customization choices [18] for users to control. Part of these customization choices are the numerous parameters that can be tuned to impact a model's generation. This section will discuss the main parameters that are utilized to tweak generated outputs, as well as discuss parameters that can reduce repetition or increase entity inclusion rates. It should be noted that though there are many parameters to adjust, they often only have a small impact on the generated output.

Common Adjustments:

- **Minimum & Maximum Length:** Sets the minimum and maximum lengths that the generated output can be. While the maximum length is a hard cut-off, the minimum length is not always reached if there are no more suggested tokens for generation. Furthermore, setting a minimum length can also force the model to make longer generations, which is ideal for our goals.

- **Temperature:** A numerical input that increases or decreases the confidence a model has in what the most likely response is. A higher value makes the model less confident. For example, a model that has a low temperature examining the following sentence "The mouse ate some ____" might consider "Cheese" to be the correct word with 95% confidence and 5% confidence for "Pizza". If the temperature were set higher, it might begin to equalize the confidence wherein "Cheese" would drop to 75% and "Pizza" rise to 25%. As temperature rises higher, the rates would equalize further.

- **Greedy Searching:** This method selects the word with the highest probability, conditional on all prior words, as the next entry. This method is quite simple but can lead to poor and unvaried results as words with very high conditional probabilities can be skipped over if they're masked by earlier words with low probabilities. In Figure 3.4, the starting word prior to generation is "The". As we follow along the lines or beams, we find that the next words could be "dog", "nice", or "car" with 40%, 50%, and 10% probability respectively. And so when utilizing a greedy search, the highest probability is selected and the beams not selected are discarded.

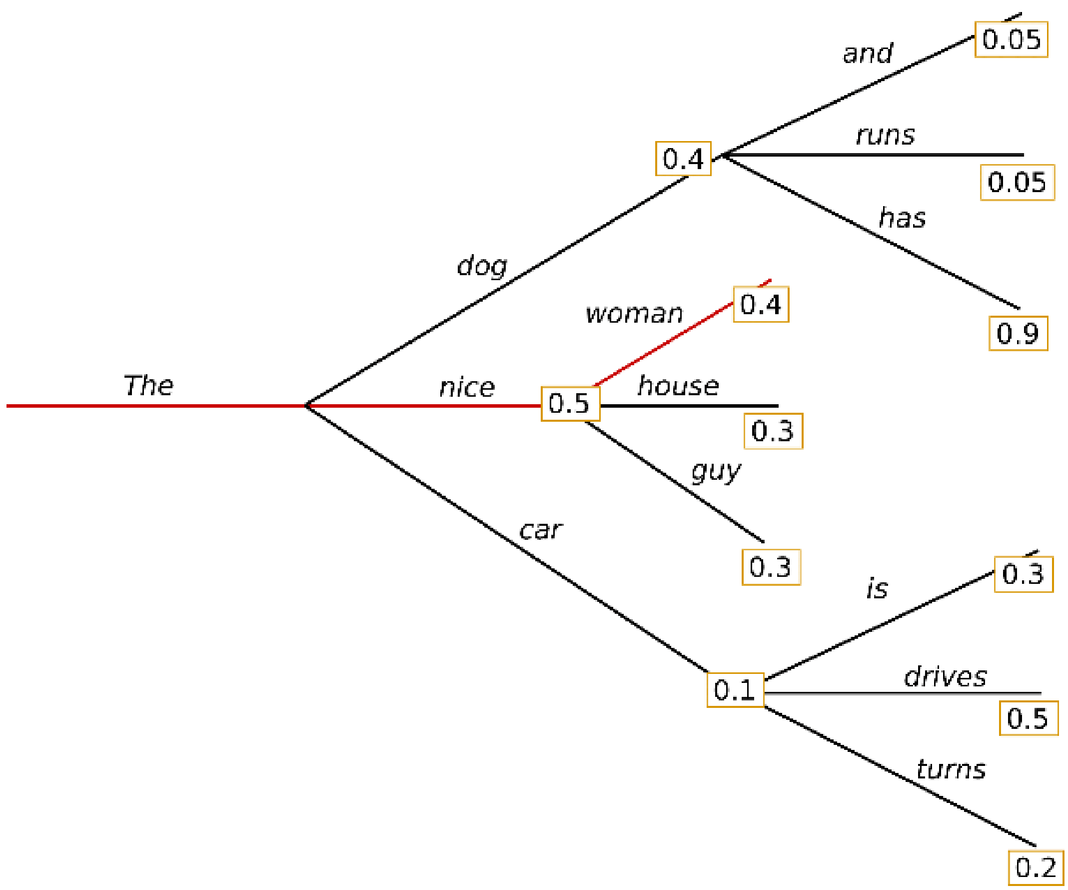


Figure 3.4: Greedy Search Generation [18]

- **Beam Searching:** Beam searching provides a solution to the masking that occurs when utilizing a greedy search. Though it is more costly, this method keeps track of the probabilities for *n*-beams and then selects the words giving the highest probability. Figure 3.5 is a simple repeat of the one from the greedy searching section, but now has an additional line (going upwards instead of straight ahead after "The"). If the number of beams in a beam search were set to 2, the generative model would explore both the highest and second highest probability beams, in this case "Nice" and "Dog". At the next time step, it would then find the sequence "The dog has" to have an overall higher probability and thus would select this beam instead of the original, "The nice woman".

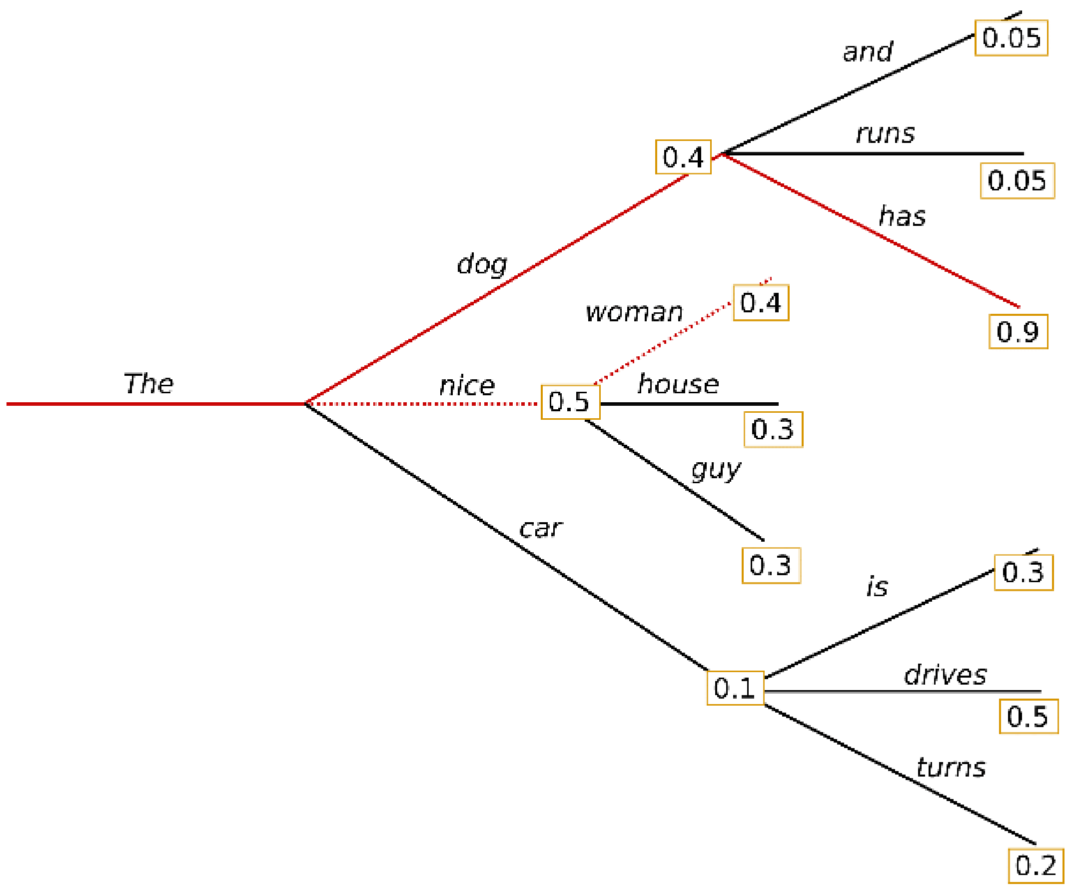
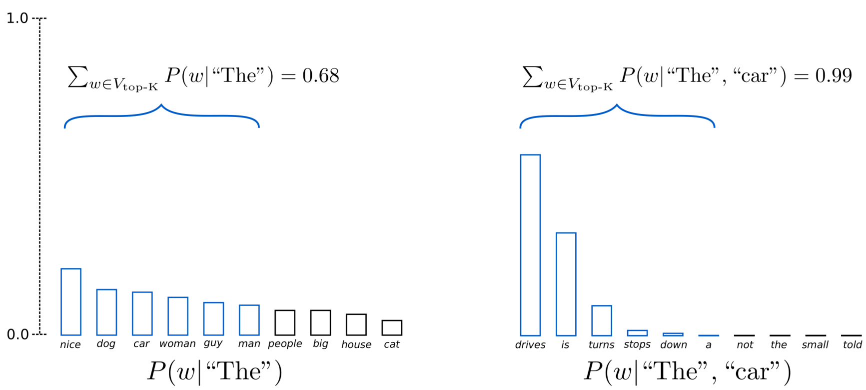
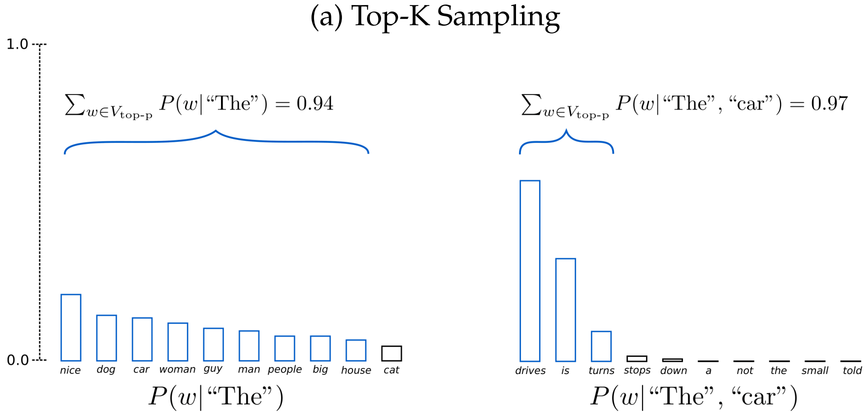


Figure 3.5: Beam Search Generation [18]

- **Sampling:** This method randomly selects the next word based on the conditional probability distribution of the prior words.
  - **Top-K Sampling:** By utilizing Top-K sampling, the k most likely words are selected, while the rest of the potential words are removed, thus changing the probability distribution of the remaining k words.
  - **Top-P (Nucleus) Sampling:** In a very similar process to Top-K sampling, Top-P or "Nucleus" sampling selects the smallest group of words whose total probability exceeds some rate p. Thus it provides a more dynamic approach as compared to Top-K.



(a) Top-K Sampling



(b) Top-P Sampling

Figure 3.6: Sampling Methods [18]

Repetition Solutions:

- **Repetition Penalty:** One common issue with generated outputs is that tokens can tend to repeat themselves. As a quick overview, setting a penalty value reduces the value given to tokens that have already appeared. There is also a setting that can prevent *n*-gram repetition by preventing any outputs that would have an *n*-gram appear twice.

3.5 Known Abstractive Summarization Issues

This section will provide some background on known issues with abstractive summarization generation, with examples from the dataset used presented. There are two issues that are well-known and well-researched in the NLP field, and various solutions have been proposed to mitigate them. One of the common issues in text generation is repetitive outputs, while the other is that generated text can create new entities that were not present in the original text and can even go on tangents related to these entities. Due to the use of pre-trained models for this project which prevents the altering of the underlying Seq2Seq encoder-decoder structure, some of the researched solutions could not be implemented, but are still discussed.

As an example used in this research, Table 3.1 displays one of the original articles from the WikiGold dataset [19] and two of the subsequent summarizations provided by the selected summarization model. Though the cases with repetition and entity hallucination are not necessarily frequent, different paragraph structures and hyper-parameter configurations can influence rates of occurrence.

Type	Output
Original	010 is the tenth album from Japanese Punk Techno band The Mad Capsule Markets . This album proved to be more commercial and more techno-based than Osc-Dis , with heavily synthesized songs like Introduction 010 and Come . Founding member Kojima Minoru played guitar on Good Day , and Wardanceis cover of a song by UK post punk industrial band Killing Joke . XXX can of This had a different meaning , and most people did n't understand what the song was about . it was later explained that the song was about Cannabis (' can of this ' sounding like Cannabis when said faster) it is uncertain if they were told to change the lyric like they did on P.O.P and HUMANITY . UK Edition came with the OSC-DIS video , and most of the tracks were re-engineered .
Repetition Loop	Japanese Punk Techno band The Mad Capsule Markets' second album Osc-Dis was met with a lot of controversy after the band changed the lyrics of their song XXX can of This to make it sound more like XXX can of This, make it sound more like XXX can of This, make it sound more like XXX can of This, make it sound more like XXX can of This
Entity Hallucination	Japanese Punk Techno band The Mad Capsule Markets have released their second album.010 in the UK, following the success of their first album Osc-Dis, which was released in the UK and Ireland in May of this year and became one of the best-selling albums of all time in Ireland.

Table 3.1: Example of Summary Degeneration

In the paper "The Curious Case of Neural Text *De*Generation" [14], the authors explored hyper-parameter tuning options and their impact on both repetition and entity hallucination rates. While this paper was for sequential text generation as opposed to abstractive summarization, it still provides valuable insights.

Repetition Loops

In a further expansion on the paper by Holtzman et al. exploring text degeneration, the likelihood objective function was examined to determine if it had an impact on degeneration [15]. After examination, it was found that the likelihood function was giving too much weight to generated outputs that contained repetition. To solve this issue, researchers adjusted the likelihood objective function to penalize unlikely tokens further than it otherwise might have at both a sequence and token level.

Furthermore, though the above techniques were tested on sequential text generation, this method of "unlikelihood" training was also extended for abstractive summarization [16], with some modifications made. As a baseline without any tweaks, the unlikelihood method boosted ROUGE scores and reduced repetition. The key modification applied to further boost scores and reduce repetition used a variation on a coverage mechanism that penalized the model if the decoder's cross attention mechanic examines the same token multiple times. This variation acted on a token level, wherein the model is penalized for giving high probabilities to the same tokens multiple times.

Entity Hallucination/Factual Consistency

One such approach to handling entity hallucination hypothesized that the issue of entity hallucination is embedded within the training dataset. The researchers [17] proposed three new metrics to quantify if entities are consistent across both source and generated texts.

- Precision-Source or Prec(S) = $N(G \cap S) / N(G)$
  Where N refers to the entity set for the source document (S) and generated summary (G). This metric examines how many entities from the generated summary are a part of the source document.

- Precision-Target or Prec(T) = N(G ∩ T) / N(G)
  Where T refers to the human-made summary. This metric examines how many entities from the generated summary are a part of the human-made summary.

- Recall-Target or Recall(T) = N(G ∩ T) / N(T)
  This metric examines how many entries from the human-made summary are *not* present in the generated summary.

To mitigate entity hallucination, these metrics were utilized to subsequently filter out the dataset based on specific precision-recall thresholds. Specifically, if entities within the generated summary were not present in the source doc, the sentence in the summary is removed. If the summary is only one sentence, then the source document-human summary pair is entirely removed from the training data.

Along with this data filtration method was a proposed change to encoder structure, wherein the encoder is trained to classify summary-"worthy" entities that are in the source document and human summary. The filtering of data combined with summary-"worthy" entity classification resulted in significant improvements in curbing entity hallucination.

As the goal of this research is creating augmented data for named entity recognition, hallucination in summary outputs is un-ideal unless a method can be found to properly tag these newly added entities. Though this method was successful in reducing hallucination, it requires a training set with both source document and human-made summary. Unfortunately, no dataset exists that has (a) entity tags (b) source articles, and (c) human-made summaries of the articles. As a result, this method could not be employed. In an attempt to still mitigate the issue of entity hallucination, ROUGE score filtering is utilized and discussed in more detail in the Methodology chapter.

Chapter 4: Methodology

The methodology section aims to highlight the approach that will be taken stemming from the aforementioned methods and information discussed in the literature review and background sections. Some of these reviewed methods will serve as a baseline for comparison to the new approach examined. While rules-based data augmentation approaches have been fairly well studied, even within the NER field where the task is somewhat more challenging, generative data augmentation approaches have significantly less exposure and testing.

4.1 Proposed Summary Generation Approach

There are two types of summarization methods, these being extractive and abstractive. Extractive summarization is a method wherein each sentence within the overall text is evaluated for importance. These sentences are then compared amongst one another, at which point only the subset of sentences classified as important are used to construct a summary. Meanwhile, abstractive summarization evaluates the text and then generates a brand new summary, both taking portions from sentences as well as reorganizing, changing phrases, and adding/removing words.

Since the goal of DA is to artificially construct *new* sentences, extractive summarization provides no benefit. Any summary constructed would be a combination of sentences taken verbatim from the original data. Thus, we consider abstractive summarization for this project.

4.2 Overview of Process Pipeline

This section will quickly outline the steps to the process of generating sufficient abstractive summaries for data augmentation and subsequent testing. Section 4.3 through to Chapter 5 will go into more detail.

1. Pre-Trained Model
   (a) Abstractive Summary Models
   (b) Named Entity Recognition Models
2. Source Data Adjustments
   (a) Tag Linearization
   (b) Tag Replacement Variants
   (c) Named Entity Weighted Order
   (d) Article Stemming
   (e) Shuffling
   (f) ROUGE Scoring
3. Train-Test Splitting
4. Model Training
5. Model Evaluation and Comparison

4.3 Models for Abstractive Summarization

For summarization, there are currently 4 well-known options that can provide abstractive outputs (as opposed to extractive). These are:

- BART [7]
- T5 [9]
- GPT-2 [10]
- PEGASUS (X-Sum Variant) [11]

Of the above models, BART and T5 are Seq-2-Seq structured transformers, while GPT-2 is a decoder and PEGASUS has an underlying T5 model but was trained on data specifically for abstractive summarization. BART, T5, and GPT-2 can be utilized for text generation, translation, Q&A, and abstractive summarization, among other things, while PEGASUS is geared more towards only abstractive summarization. The following sections briefly highlight the unique differences in models.

BART

The BART transformer structure was created by Facebook. The structure is an expansion on BERT, which at its core is a bi-directional encoder which was trained via masked language modelling. BART furthers this by adding in an autoregressive decoder to provide further functionality over BERT and access to standard encoder-decoder tasks. Training of the model first corrupted source text via noising function and then learned to reconstruct the corrupted text as close to source-level as possible.

T5

The T5 transformer structure is very similar to BART and was created by Google and released within the same week that BART was. It was also an expansion from BERT and thus has the same bi-directional encoder as BART, and further adds its own auto-regressive decoder. The primary differences in T5 and BART are the structure of layers within the decoder. One other slight difference is that the training method used by T5 is no longer corrupt/fill-in-the-blank but instead has a mix of variations used to train for different tasks.

GPT-2

GPT-2 was created by Open-AI and is a transformer structure comprised of only an autoregressive decoder. Training for GPT-2 occurred on web-texts, wherein it attempted to predict new tokens with only information of the prior ones given to it. The decoder method is similar to T5 and BART, but the actual structure of the model is different. Due to lack of support via HuggingFace for GPT-2 summarization, this model was not examined in the paper.

PEGASUS

PEGASUS was created by Google and based off the original T5 Seq2Seq transformer structure. Its goal was to fix other models' weaknesses in abstractive summarization generation. PEGASUS pre-trains the Seq2Seq model on a large text corpus wherein important sentences get removed/ masked from the input source and then generated utilizing the remaining sentences. As a result, this pre-trained model is quite strong at summarization specifically, but cannot be utilized as well for other Seq2Seq tasks.

Final Decision

Based on the training information for the above models, PEGASUS seems an ideal choice. To quantitatively verify this, a small sample of 25 articles from the WikiGold dataset were passed into all three models (BART, T5, PEGASUS) simultaneously, with each model using the same configuration of hyper-parameters. The output of each model was then examined by human eye for sentence structure and fidelity. The ideal (or "sufficient", these two words may be used interchangeably) output was one that was coherent, stayed mostly faithful to the original article, and also provided a new structure that was not simply re-utilizing a sentence directly from the article. Sufficient length was also a factor. Multiple hyper-parameter combinations were examined, after which point it became clear that the PEGASUS X-Sum model (herein referred to simply as PEGASUS) was able to provide the best results consistently (as seen in Table 4.1. The other models tended to do one of three things most frequently:

1. Provide frequent incoherent, or repetitive outputs
2. Provide extractive (or near-extractive) outputs as opposed to abstractive ones
3. Provide excessively short outputs ($\tilde{3}$-7 words maximum, even for articles that were 300+ words).

On the other hand, the PEGASUS model tended to provide outputs that were often coherent, of significant length (long or multiple sentences), and sufficiently unique from the sentences within articles.

As a result, the PEGASUS model was selected for this research. After this selection, other variants of the PEGASUS model were also examined (such as those tuned specifically on CNN Daily Mail and WikiHow datasets), but the X-Sum variant proved most consistent. Table 4.1 displays parameter settings by model and generated output issue rates on 25 articles.

Parameters	**Model**	**Satisfactory**	**Extractive**	**Repetitive**	**Incoherent**	**Hallucination**
Default	BART	0/25	19/25	0/25	6/25	0/25
	T5	1/25	19/25	0/25	5/25	0/25
	PEGASUS	17/25	7/25	0/25	1/25	0/25
MinLength=Article	BART	0/25	22/25	0/25	3/25	0/25
	T5	1/25	22/25	0/25	2/25	0/25
	PEGASUS	18/25	0/25	5/25	0/25	2/25
MinLength=Article Top P=0.9	BART	0/25	22/25	0/25	3/25	0/25
	T5	0/25	24/25	0/25	1/25	0/25
	PEGASUS	11/25	0/25	6/25	0/25	8/25
MinLength=Article Num Beams=32	BART	0/25	22/25	0/25	3/25	0/25
	T5	0/25	23/25	0/25	2/25	0/25
	PEGASUS	17/25	0/25	6/25	0/25	2/25
MinLength=Article Num Beams=32 RepetitionPenalty=2.0	BART	0/25	22/25	0/25	3/25	0/25
	T5	0/25	23/25	0/25	2/25	0/25
	PEGASUS	17/25	0/25	4/25	0/25	4/25

Table 4.1: Summary Output Success Rates

Clearly, PEGASUS heavily outperforms the T5 and BART models. As for the impact of parameters on the PEGASUS model, the minimum length being set to the article length seems to provide the largest boost to results. Further adding in a fixed number of 32 beams, we see a slight uptick in repetition. Adding in a repetition penalty ends up increasing hallucination. The parameters chosen need to strike a balance between providing sufficiently new sentences while mitigating repetition and especially hallucination. In the end, the parameters selected were:

- Minimum Length = Article Length
- Number of Beams = 32

This pairing did have more repetitions than when the number of beams was set to none, but these repetitions proved to be much smaller tokenwise and only at the end of the sentence for a few words, as where in the case of no beams the repetition was much larger (e.g. the entire sentence would be repeating several times versus a few words repeating at the end of the sentence twice).

4.4 Named Entity Recognition Models

Unlike abstractive summarization, named entity recognition models are consistently strong at performing the task required of them, with little training data required to fine-tune for decent results.

A common choice for NER tasks, the "BERT" [12] model was selected. This model was pre-trained on a large corpus of text data in the English language. The model was unsupervised in that no labels were provided by humans. The model training was performed utilizing a masked language model (MLM) wherein for each sentence $\tilde{1}$5% of the words within were masked or "hidden" from the model, at which point it had to estimate what the actual word was. Feeding the entire sentence at once allowed the model to learn bi-directionally. Similarly, the model was also tasked in predicting if two sentences followed one another or not.

Two key variations on the BERT model are the cased and un-cased versions. The difference is simple, in that the cased variant was trained on un-edited text, while the un-cased variant had all of the text converted to lower-case. Based on the structure of the WikiGold dataset and given that the entities are generally speaking all capitalized (when not numerical), the cased version was selected over the uncased one.

4.5 Data Source Adjustments

Data source adjustments refer to changes made to the source article text in an attempt to improve summarization output.

Linearization

One common approach in NER tasks for preparing generational models is linearization, wherein words with entity tags have said tags added into the sentence itself in some way. Commonly, the tag is moved in front of or behind the word associated with it. Sometimes the tag goes both in front and behind, or has "B"/"E" prefix and suffix to denote the beginning and end of an entity.

Tag Replacement

In some cases, entities can be quite long and constructed of words that are commonly not entities. As a result, models can struggle to identify when certain non-entity words should be grouped together as entities instead. For example, "The 10th Battalion of Slayer's Creek Pontamac Group" is a very long entity that a model might struggle with understanding contextually within an article and thus might result in incoherent generations being output. To help the model understand sentence structure and context more easily, three tweaks were made to the original article structure, resulting in the four following potential input texts for every article. Replacement did not include "O" tags.

1. Original: Unaltered
2. Full Tag Replacement: Every token was replaced with their respective entity tags.
3. One Tag Replacement: Every entity was replaced with a single variant of their entity tag.
4. Unique Tag Replacement: Every entity was replaced with a single entity tag that had a unique number identifier appended to it.

See Table 4.2 as an example that compares the adjusted variants. Note that the unique tag replacement would mark subsequent unique PER/MISC tags as PER2, PER3, and so on.

Variation	**Output**
Original	Oleg Gazmanov is a Russian singer .
Reference Tags	PER PER O O MISC O O
Full Tag Replacement	PER PER is a MISC singer .
One Tag Replacement	PER is a MISC singer .
Unique Tag Replacement	PER1 is a MISC1 singer .

Table 4.2: Example of Article Entity Replacements

Generated Summary Entity Mapping

There are two primary ways of mapping tokens in the generated outputs.

1. **Sliding n-gram Approach:** The sliding n-gram approach is required for generated summaries based off of the original un-altered article. This approach maps all n-gram segments of entities from the source article. Then, the generated summary is initialized as all "outside" tags. Lastly, each n-gram within the summary is looped through and the tag overwritten if a mapping is found. Though this approach is not perfect, it is fairly accurate. Table 4.3 illustrates how the mapping function iterates on a sample text.

Iteration	**Output**
Original	Oleg Gazmanov is a Russian singer .
Reference Tags	PER PER O O MISC O O
Initialization	O O O O O O O
1-Gram Iteration	O O O O MISC O O
2-Gram Iteration	PER PER O O MISC O O

Table 4.3: Example of Sliding n-gram Mapping

2. **Substitution:** As an attempt to improve entity mapping and prevent any errors, the tag replacement article variations were utilized. In the case of the One-Tag replacement scheme, it was sufficient to simply substitute one of the entities associated with the tag as this would not cause any incoherence within sentences. In the case of the Unique-Tag replacement, the exact entity was substituted back in.

Named Entity Weighting

Weighted ordering is a method of shuffling the sentences within the original article. Sentences are given a weighting based on the proportion of entity words within the entire sentence. Once each sentence is weighted, the article has its sentences re-ordered in either ascending or descending order of entity weighting. This technique was devised from the fact that, generally speaking, the first few sentences of an article tend to be the most important in summarizations done by humans. By proxy, one can assume that training data for summarizations would also follow this rule and as a result the PEGASUS model might inherently prioritize the first few sentences in its generation. In an examination of twenty-five articles, twenty-one of the article summaries ($\tilde{8}$4%) referred to content from the first three sentences, often starting the summary the same as the first few words or phrases from the article.

Article Stemming

Also relying on the entity weighting at a sentence level, this method simply removes any sentences that had a weighting below some defined threshold (e.g. 5%, 10%, etc.). While this approach might help increase the number of entities in the generated summary, it also has a large downside. Often, the generated outputs will combine two or three sentences into one fluent sentence that pulls details from all of the other sentences. By removing some sentences, we also remove filler information that the model could use to transition contextually from one entity to another. As a result, the summaries could become more extractive in nature, which is not necessarily ideal.

Shuffling

One of the issues with abstractive summarization generation is that results are near identical for repeated runs on the same article. This provides a unique issue wherein it becomes challenging to get a sufficient number of generated samples from which to augment with. A simple approach of shuffling the sentences within the articles is taken. For each article, the sentences are split into a list. This list is then randomly shuffled, recorded, and recombined into a full article string. For subsequent shuffles, they are checked against the previously recorded shuffles to ensure that said shuffle is not mimicking the one already utilized. This list of newly shuffled articles is then fed into the model to generate summaries. Though shuffling does not always result in new summary generations, there are enough permutations existing that this is no longer an issue.

ROUGE Scoring & Summary Filtering

The ROUGE metric stands for "Recall-Oriented Understudy for Gisting Evaluation". It provides a quantifiable output that scores the degree of similarity between one text and another. There are a few variations on ROUGE that adjust the level of granularity. For example, ROUGE-N measures how many N-grams overlap between two texts, while ROUGE-L compares via the longest sequence match and ROUGE-S compares via ordered pair similarities.

This project focuses on the ROUGE-N metric, for which recall refers to the percentage of n-grams from the reference text that are present in the generated text, and precision refers to the percentage of n-grams in the generated text are also present in the reference text. An F1 score also exists as a balancing metric.

In the context of this project, ROUGE scores provide a metric that can assess the adequacy of the generated summaries as compared to the original article. However, ROUGE scores do not *necessarily* provide insight into if a summary is fluent and void of repetition. As a result, there is some degree of post-processing that is required by a human. Given that the goal is to generate numerous summaries for data augmentation, human post-processing would require extensive examination of generated outputs, which is less than ideal.

To get around this issue, we can utilize the knowledge that *most* summaries generated using the PEGASUS model are sufficiently fluent and that *most* lack repetition. In previous human examination of a subset of article summaries (Table 4.1), less than 8% were incoherent and though 25% were repetitive, this only occurred at the very end of summaries. Furthermore, though higher ROUGE scores do not directly indicate the degree of fluency or repetition, higher scores tend to be more suitable.

As a result, we can filter out samples with lower scores (those with F1-scores below 20%) and simply take the top samples. For the number of samples selected, we simply take the article sentences and multiply this by 3 to generate 3 samples per sentence in the article. Though this approach does not guarantee that all the generated summaries utilized are sufficient, it provides more confidence than simply selecting randomly.

4.6 Model Evaluation & Comparison

The baseline approaches test other well-researched in an attempt to provide valid comparison for the new summarization approach. The two baseline approaches (both discussed prior in Section 2) being utilized are:

1. **Rule-Based Approach:** As a rule-based baseline, we examine three of the four data augmentation techniques; Label-Wise Token Replacement, Shuffle in Segments, and Synonym Replacement. Mention Replacement is not included due to the use of IO tag scheme over the required BIO scheme.

2. **Paraphrase Approach:** As a generational model baseline, we examine the method of paraphrasing on a sentence level to provide newly augmented sentences.

Chapter 5: Experimental Results and Analysis

5.1 The WikiGold Dataset

5.1.1 Dataset Requirements & Selection

To carry out this research focused on utilizing abstractive summarization for Named Entity Recognition data augmentation, the dataset utilized had two key requirements.

1. The dataset needed to contain named entity tags for every token.
2. The dataset needed to contain sentences that were organized and coherent, with numerous relating to a single topic to form a paragraph/article. Individual unrelated sentences would not suffice for summarization.

The only dataset that adhered to the above two conditions was the "WikiGold" dataset [19]. The WikiGold dataset is a manually annotated corpus for named entity recognition and made up of a small sample of varied Wikipedia articles. Each word within the dataset (tokenized) was labelled with one of five tags which were taken from the CONLL-03 dataset. These tags were:

- O - The standard tag indicating a non-entity or "Outside" entity.
- LOC - An indication that the associated word is a Location.
- PER - An indication that the associated word is a Person.

- ORG - An indication that the associated word is an Organization.
- MISC - An indication that the associated word is an entity, but one that does not fall into the above categories.

Similarly to the CONLL03 dataset, a simple IO tagging format was utilized. Though this could easily be converted into an IOB system, for the purposes of abstractive summarization this change was not necessary and would hinder sentence structure for some techniques utilized.

5.1.2 Low-Resource Mimicry Selection

To mimic the varying degrees of data resources available and to properly examine the augmented data's impact on NER model performance, different groupings of 50, 100, 250, and 500 sentences are selected for testing purposes to perform augments upon.

To select these batches, the sentences within each article were counted. Next, articles with less than 2 and more than 40 sentences were filtered out to prevent both low sentence counts that cannot generate a sufficient number of shuffles and high sentences that would fill the entire batch size with just one or two articles and thus cause training data to be concentrated on one context/topic. With this filtering done, articles were then randomly selected until the total sentences combined from them were within 10% of the batch size (e.g. for batch size 50 between 45 and 55 sentences). Articles that were not selected for the training set were left for testing. This was then repeated 10 times with different random selections for each batch size to provide sufficient replication.

5.1.3 Descriptive Statistics

As a high-level overview of the WikiGold dataset, some statistics are provided to the reader. The dataset can be described with the following metrics:

- # of Articles: 145
- # of Sentences: 1,768
- # of Tokens: 39,007

- # of Non-Alphanumeric Tokens: 4,893
- # of Entities: 6,431
- Entity Categories: O, ORG, LOC, PER, MISC

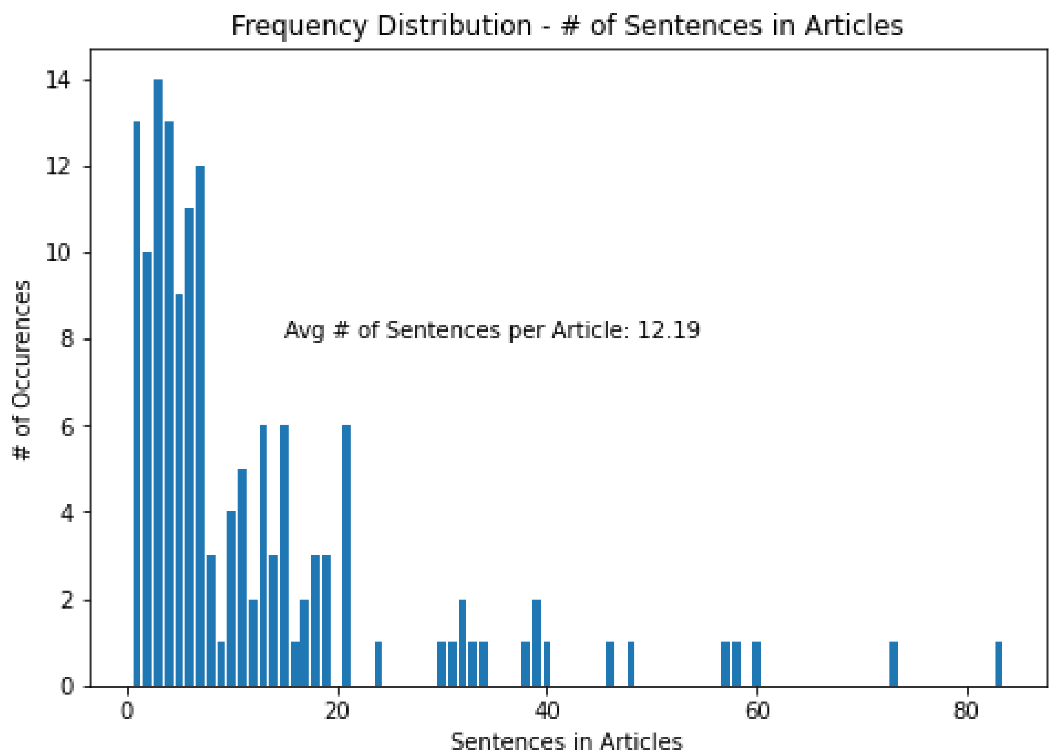


Figure 5.1: Distribution of Sentence Count per Article

Batch Size	**X-Small**		**Small**		**Medium**		**Large**	
Set	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**	**Train**	**Test**
# of Articles	5	140	9	136	23	122	45	100
# of Sentences	49.6	1,718.4	100.6	1,667.4	246.3	1,521.7	505.8	1,262.2
# of Non-Entities	841.2	31,734.8	1,753.1	30,822.9	4,380.9	28,195.1	9,321.7	23,254.3
# of Entities	181.1	6,249.9	406.6	6,024.4	972.2	5,458.8	2,018.7	4,412.3

Table 5.1: Metrics by Training Test Split
(Averaged over 10 Randomized Replications)

5.2 Data Augmentation Method Results

Linearization

Attempts to linearize entity tags into the sentence prior to summarization resulted in a high degree of incoherency (wherein the model did not know how to interpret the linearized tags in a summary). As a result, linearization was unsuccessful.

Named Entity Weighting & Stemming

While in theory weighting named entities to the front of the articles should produce a generation output with a higher number of entities due to how models are trained, this only resulted in one viable generated output. As for stemming, nearly all articles contained a sufficient number of entities within each sentence and as a result would have rarely removed sentences. Furthermore, the generated output from a shuffle post-stem often did not vary from the original un-stemmed version if the sentence removed were to be replaced back in the proper location. As a result, this approach was passed on in favour of the simpler shuffling mechanic.

5.3 NER Model Settings

Training settings for the BERT-Base Cased model iterated through 3 epochs with a learning rate of 2e-5 (0.00002) and a weight decay rate of 0.01. The model was evaluated with Precision, Recall, and F1 metrics.

5.4 Model Results

Table 5.2 below illustrates precision, recall, and F1 metrics on the testing set when the model was trained via their respective training sets. Training was randomized and was replicated 10 times on different train-test variations of the same batch-size to provide an accurate confidence interval on the accuracy metrics.

To make results clearer, the dataset with the highest F1 score for each batch size is bolded. The original dataset pertains to a batch of un-augmented data. In the case of the baseline datasets (Rules and Paraphrased), they contain the original un-augmented data and an additional three augmented sentences per original sentence. In the case of the summarization methods (Article, One Tag, and Uni Tag, as seen in Table 4.2), they contain the original un-augmented data and then take a number of generated summaries that is equal to three times the number of sentences in the original article to approximately match the three additions from the baseline (as each generated summary is often one sentence).

The metrics for the Rules dataset are for the highest-performing method/binomial rate combination (LWTR at a rate of 10%), though results across methods within rates were quite similar. As the rate increased, performance dropped slightly across all rule-based methods.

Overall, we see that the Rules based data augmentation method performed the best by a sizeable amount, with paraphrased and article augmentation methods falling slightly behind. One key difference is that the rules-based approach tends to have a significantly higher recall score at low batch sizes than the other two methods, which contributes to the higher F1 score overall as in comparison precision is quite similar albeit also somewhat higher.

Batch Size	Dataset	Precision	Recall	F1-Score
X-Small (S=50)	Original	3.33%±6.53%	0.00%±0.00%	0.01%±0.01%
	Rules*	**26.28%± 7.18%**	**16.48%± 4.79%**	**20.01%± 5.16%**
	Paraphrased	22.38%± 6.27%	8.37%± 3.08	11.68%± 3.56%
	Article	23.46%± 5.98%	8.69%± 3.65%	12.05%± 4.41%
	One Tag	15.97%± 9.63%	0.91%± 0.88%	1.64%± 1.56%
	Uni Tag	16.38%± 9.10%	1.83%± 1.47%	3.06%± 2.33%
Small (S=100)	Original	8.45%±5.56%	0.59%±0.77%	0.99%±1.23%
	Rules*	**53.77%± 3.1%**	**59.14%± 7.34%**	**56.18%± 4.84%**
	Paraphrased	47.27%± 2.58%	47.64%± 5.06	47.20%± 3.75%
	Article	43.49%± 3.83%	40.2%± 5.62%	41.61%± 4.71%
	One Tag	23.18%± 3.07%	12.56%± 3.37%	16.03%± 3.47%
	Uni Tag	25.32%± 2.35%	15.52%± 3.33%	18.91%± 3.12%
Medium (S=250)	Original	41.52%±2.97%	42.10%±3.90%	41.76%±3.32%
	Rules*	**70.07%± 1.7%**	**75.77%± 1.9%**	**72.8%± 1.54%**
	Paraphrased	67.34%± 1.03%	69.51%± 2.02	68.38%± 1.32%
	Article	66.50%± 1.20%	66.34%± 1.66%	66.41%± 1.34%
	One Tag	56.21%± 1.53%	47.97%± 2.70%	51.72%± 2.18%
	Uni Tag	56.70%± 2.34%	49.23%± 3.24%	52.68%± 2.84%
Large (S=500)	Original	65.93%±1.05%	70.49%±1.45%	68.11%±0.83%
	Rules*	**75.91%± 1.76%**	**82.19%± 1.21%**	**78.92%± 1.34%**
	Paraphrased	74.41%± 1.20%	79.14%± 0.85	76.70%± 0.99%
	Article	72.75%± 0.84%	75.92%± 0.47%	74.30%± 0.63%
	One Tag	68.58%± 1.02%	67.45%± 1.25%	68.00%± 0.96%
	Uni Tag	69.96%± 0.88%	69.91%± 1.40%	69.91%± 0.94%

Table 5.2: Model Results for Tested Methods (α = 0.05, n=10)
**Label-Wise Token Replacement, 10% Rate*

Chapter 6: Future Work and Conclusions

6.1 Future Work

Theoretical Optimal Tuning

With all the aforementioned uncertainty in summary generation, the process of fine-tuning hyper-parameters would be incredibly costly and time-consuming. The approach that could be taken is discussed in this section, but was not done in actuality.

There are three main hurdles that would require various configurations to be tested to determine the optimal hyper-parameters for ideal summary generation.

1. **Model Hyper-Parameters:** As discussed in the previous sections, there are a large number of model parameters that can be adjusted to tweak generated outputs. In total, there are seven parameters of interest that would need to be tested, these being: search method (greedy or beam), sampling type (top-k or top-p), temperature, repetition penalty, and minimum length.

2. **Article Variations:** Also discussed prior, the four variations on the original articles that intend to make generation easier for the model would all need to be evaluated.

3. **Shuffle Variations:** Lastly, different shuffling of article sentences would need to be tested for further consistency.

These variations would then have to be examined by human eye, though ROUGE filtering could help cut down manual labour required. Overall, combining the three core factors that would need to be tested rigorously results in a theoretical number of combinations too costly to examine. For instance, examining 30 shuffled variants across the 4 article variations takes upwards of 12 hours on one set of conservative model hyperparameters. As such, configuring all 7 and trying various combinations in a grid-search method would take many days.

Higher Augment Multiples

While the paraphrasing and abstractive summarization results are fairly similar, there is a limit to how many paraphrased variations models can generate. While the sentence "His name was Bill Gates and he founded Microsoft" could viably be paraphrased in three other ways (e.g. "Named Bill Gates, he is the founder of Microsoft, etc.), results are often still quite similar and not varied. With this in mind, it would be worth exploring the impact of (a) a higher augment multiple (i.e. augment 7 new sentences instead of 3 from one original) and (b) how paraphrasing holds up against summarization in these scenarios.

6.2 Project Conclusion

The Aim

The goal of this project was to explore a new method of data augmentation specifically for the natural language processing task of tagging named entities. Current methods of data augmentation for NER tasks include simple algorithmic rules-based approaches, as well as more complicated generational approaches such as back-translation, paraphrasing, and sequential generation. As a new approach, abstractive summarization was explored and tested.

Methods Explored

With the new approach via abstractive summarization, some unique challenges presented themselves. Firstly, generating a summary requires sentences that are related to one another and pertain to the same context. Furthermore, a single sentence cluster (article) will only generate one to two sentences at most, meaning that an article that is 40 sentences in length will only have 2 new augmented sentences. However, human generated summaries tend to be constructed from the first few sentences of an article; as a result, models that are focused on generating summaries (PEGASUS) tend to have inherently learned a higher weighting for the initial sentences in longer articles. With this in mind, we can shuffle the sentences within an article to significantly alter the generated summary output. Through this technique, the article with 40 sentences can now be re-shuffled many times to provide as many augmented sentences as necessary.

Summary generation is not always consistent; in an attempt to increase consistency, methods of replacing entities with their associated tags (somewhat similar to sentence linearization) prior to generation were also explored.

Conclusion

The results of summary generation with and without tag replacement were significantly lower for all batch sizes, with the tag replaced variants performing much worse on lower sample sizes. As for the original article summary method without tags replaced, it performed only slightly worse to the paraphrasing method performed in other research. Though it did outperform on the smallest batch size, this could simply be due to only 10 replications being performed. When compared to the rules-based approach baseline however, both the paraphrasing method and abstractive summarization techniques end up being outperformed by a fairly significant margin at the X-Small, Small, and Medium batch sizes. At the large batch size, all methods provide a slight boost but are much closer in performance.

Bibliography

[1] Xiang Dai and Heike Adel. (2020) *An Analysis of Simple Data Augmentation for Named Entity Recognition*. https://aclanthology.org/2020.coling-main.343.pdf

[2] Bosheng Ding, Linlin Liu, Lidong Bing, Canasai Kruengkrai, Thien Hai Nguyen, Shafiq Joty, Luo Si, and Chunyan Miao. (2020) *DAGA: Data Augmentation with a Generation Approach for Low-resource Tagging Tasks*. https://aclanthology.org/2020.emnlp-main.488.pdf

[3] Shuguang Chen, Gustavo Aguilar, Leonardo Neves, and Thamar Solorio. (2021) *Data Augmentation for Cross-Domain Named Entity Recognition*. https://arxiv.org/pdf/2109.01758.pdf

[4] Usama Yaseen and Stefan Langer. (2021) *Data Augmentation for Low-Resource Named Entity Recognition Using Backtranslation*. https://arxiv.org/pdf/2108.11703v1.pdf

[5] Petr Marek, Štěpán Müller, Jakub Konrád, Petr Lorenc, Jan Pichl, and Jan Šedivý. (2021) *Text Summarization of Czech News Articles Using Named Entities*. https://arxiv.org/pdf/2104.10454.pdf

[6] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. (2019) *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. https://arxiv.org/abs/1907.11692

[7] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. (2019) *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. https://arxiv.org/abs/1910.13461

[8] Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. (2018) *Marian: Fast Neural Machine Translation in C++*. https://arxiv.org/abs/1804.00344

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. (2019) *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. https://arxiv.org/abs/1910.10683

[10] OpenAI (2019) *GPT-2: 1.5B Release* https://openai.com/blog/tags/gpt-2/

[11] Jingqing Zhang, Yao Zhao, Mohammad Saleh and Peter J. Liu (2020) *PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization* https://arxiv.org/pdf/1912.08777.pdf

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova (2019) *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* https://arxiv.org/pdf/1810.04805.pdf

[13] Rui Wang and Ricardo Henao (2021) *Unsupervised Paraphrasing Consistency Training for Low Resource Named Entity Recognition*. https://aclanthology.org/2021.emnlp-main.430.pdf

[14] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi (2020) *THE CURIOUS CASE OF NEURAL TEXT DeGENERATION* https://arxiv.org/pdf/1904.09751.pdf

[15] Anonymous Authors (2020) *NEURAL TEXT DEGENERATION WITH UNLIKELIHOOD TRAINING* https://openreview.net/attachment?id=SJeYe0NtvH&name=original_pdf

[16] Pranav Ajit Nair and Anil Kumar Singh (2021) *On Reducing Repetition in Abstractive Summarization* https://aclanthology.org/2021.ranlp-srw.18.pdf

[17] Feng Nan, Ramesh Nallapati, Zhiguo Wang, Cicero Nogueira dos Santos, Henghui Zhu, Dejiao Zhang, Kathleen McKeown and Bing

Xiang (2021)*Entity-level Factual Consistency of Abstractive Text Summarization* https://arxiv.org/pdf/2102.09130.pdf

[18] Patrick Von Platen (2020, March 18). *How to generate text: using different decoding methods for language generation with Transformers* HuggingFace. https://huggingface.co/blog/how-to-generate

[19] Pritish Uplavikar (2016). *Named Entity Recognition, Wikigold CONLL* https://github.com/pritishuplavikar/Named-Entity-Recognition/blob/master/wikigold.conll.txt

[20] HuggingFace. *How do Transformers Work?* https://huggingface.co/course/chapter1/4?fw=pt

Chapter 8: Appendices

8.1 Code Files

Note: Files are not fully integrate-able in an automated manner. Functions were run in an IDE and thus utilized pre-declared variables that would otherwise be unknown due to not being fed as a function input (e.g. "df").

8.1.1 Main Code Files

Summarization/Paraphrase Model

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Aug  4 17:46:46 2022

@author: Doug
"""

################
# LIBRARIES
################

import pandas as pd
import csv
import numpy as np
import tensorflow as tf
import time

#For Copies
import copy

#For Synonyms
```

```python
import requests
from bs4 import BeautifulSoup

#For Shuffle
import random

#Hugging Face
from transformers import PegasusTokenizer, PegasusForConditionalGeneration
    , T5Tokenizer, T5ForConditionalGeneration, MT5Tokenizer,
    MT5ForConditionalGeneration, AutoModel, AutoTokenizer

#OS
import os.path
from os import path as os_path

#ITERTOOLS
import itertools

#ROUGE METRIC
from rouge_score import rouge_scorer

#FACTORIAL
import math

#NLTK
import nltk
nltk.download("punkt")

#MEMORY CLEARING
from GPUtil import showUtilization as gpu_usage
import torch
from numba import cuda

#################
##### CACHE #####
#################

#Change Cache
import os
os.environ['TRANSFORMERS_CACHE'] = 'H:/TempHF_Cache/cache/transformers/'
os.environ['HF_HOME'] = 'H:/TempHF_Cache/cache/'
os.environ['XDG_CACHE_HOME'] = 'H:/TempHF_Cache/cache/'

################
# CODE
```

```
################
#https://huggingface.co/spaces/Wootang01/Paraphraser_two/blob/main/app.py

#INITIAL SETUP
#Set Device
torch_device = "cuda" #If throwing CUDA error, restart Python

#Set Models
tokenizer1 = PegasusTokenizer.from_pretrained("google/pegasus-xsum")
model1 = PegasusForConditionalGeneration.from_pretrained("google/pegasus-xsum").to(torch_device)

tokenizer2 = PegasusTokenizer.from_pretrained("tuner007/pegasus_paraphrase")
model2 = PegasusForConditionalGeneration.from_pretrained("tuner007/pegasus_paraphrase").to(torch_device)


##############
#### DATA ####
##############
def gold_dataframe():
    #Create List of Articles, Tokens
    df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/wikigold.txt",
                     sep=' ', header=None, doublequote = True, quotechar='"',
                     skipinitialspace = False, quoting=csv.QUOTE_NONE)

    df.columns = ["Token","Entity"]

    current_article, current_token, article_list, token_list = [], [], [],[]

    for index, row in df.iterrows():
      if df["Token"][index] != "-DOCSTART-":
        current_article.append(df["Token"][index])
        current_token.append(df["Entity"][index])
      else:
        article_list.append(current_article), token_list.append(current_token)
        current_article, current_token = [], []

    for index in range(len(article_list)):
      article_list[index] = " ".join(article_list[index])
```

```
        token_list[index] = " ".join(token_list[index])

    #Back to DF
    temp_dict = {"Article":article_list,"Entity":token_list}
    df = pd.DataFrame(temp_dict)
    return df

df=gold_dataframe()

def tagged_dataframe():
    df=gold_dataframe()
    df["Tagged_All"]=df["Article"]
    df["Tagged_One"]=df["Article"]
    df["Tagged_Uni"]=df["Article"]

    #List of Entities
    Entity_List = ["ORG","LOC","PER","MISC"]

    for index, row in df.iterrows():

        list_of_words = df["Tagged_All"][index].split(" ")
        list_of_tags = df["Entity"][index].split(" ")

        ###TAGGING ALL
        word_sentences = []
        tag_sentences = []

        word_segment = []
        tag_segment = []

        for i,word in enumerate(list_of_words):
            if word != ".":
                word_segment.append(word)
                tag_segment.append(list_of_tags[i])
            else:
                word_segment.append(word)
                word_sentences.append(word_segment)
                word_segment = []

                tag_segment.append(list_of_tags[i])
                tag_sentences.append(tag_segment)
                tag_segment = []

        for i,sentence in enumerate(word_sentences):
            for j,word in enumerate(sentence):
```

```
                tag = tag_sentences[i][j]
                if tag != "O":
                    word_sentences[i][j] = tag[2:]

        for i,sentence in enumerate(word_sentences):
            word_sentences[i] = " ".join(sentence)

        df["Tagged_All"][index] = " ".join(word_sentences)





        #TAGGING SEGMENTS AS ONE
        previous = ''
        segment_list = []
        tag_list = []
        temp1 = []
        temp2 = []

        for index2, tag in enumerate(list_of_tags):
            if tag == previous or previous == '':
                temp1.append(list_of_words[index2])
                temp2.append(tag)
            elif tag != previous:
                segment_list.append(temp1.copy())
                tag_list.append(temp2.copy())

                temp1.clear()
                temp2.clear()

                temp1.append(list_of_words[index2])
                temp2.append(tag)

            previous = tag

        segment_list.append(temp1)
        tag_list.append(temp2)

        for index3, group in enumerate(segment_list):
            segment_list[index3] = " ".join(group)
            tag_list[index3] = tag_list[index3][0]

        #print(segment_list)
        #print(tag_list)
```

```
        for index4, thingy in enumerate(segment_list):
            new_tag = tag_list[index4]
            if new_tag != "O":
                segment_list[index4] = new_tag[2:]


        df["Tagged_One"][index] = " ".join(segment_list)







        #TAGGING UNIQUE
        Entity_List_New = Entity_List.copy()
        list_of_words_tagged = df["Tagged_One"][index].split(" ")
        #list_of_words = df["Article"][index].split(" ")
        #If time, make numbering unique i.e. if band shows up twice give same # for it

        for i, word in enumerate(list_of_words_tagged):
            if word in Entity_List:
                Tag_Index = Entity_List.index(word)
                Original_Tag = Entity_List_New[Tag_Index]

                if Original_Tag[-1].isdigit():
                    New_Tag = Original_Tag[:-1]+str(int(Original_Tag[-1])+1)
                else:
                    New_Tag = Original_Tag+"1"

                Entity_List_New[Tag_Index] = New_Tag
                list_of_words_tagged[i] = New_Tag

        df["Tagged_Uni"][index] = " ".join(list_of_words_tagged)

    return df

df = tagged_dataframe()

###########################
###### SUMMARIZATION ######
###########################
```

```
def abs_summary(
    input_text, num_return_sequences, num_beams, min_length, temperature=1.5
):
    #PEGASUS XSUM

    batch1 = tokenizer1(input_text, truncation=True, padding="longest", return_tensors="pt").to(torch_device)
    translated1 = model1.generate(**batch1, temperature=temperature, min_length=min_length,
                                  num_beams=num_beams, num_return_sequences=num_return_sequences)#,do_sample=False,top_k=None)
    Pegasus = tokenizer1.batch_decode(translated1, skip_special_tokens=True)

    return Pegasus

def article_iter(shuffled, num_samples, num_beams, temperature, df_name):
    for i in range(num_samples):
        header = "Sample" + str(i+1)
        df[header] = " "

    for index, row in df.iterrows():
        print(index)

        if shuffled==False:
            article_txt = df["Article"][index]
        else:
            current = df['Article'][index].split(" .")
            current = current[:-1] #removes final empty portion
            random.shuffle(current)
            article_txt = " .".join(current)

        prop_length = article_txt.count(" ")#/6 #arbitrarily Picked

        results = abs_summary(
            input_text = article_txt,
            num_return_sequences=num_samples,
            num_beams=num_beams,
            min_length=int(prop_length),
            temperature=temperature
            )

```

```
        for i in range(num_samples):
            header = "Sample" + str(i+1)
            df[header][index] = results[i]

    df.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/
    "+df_name+".csv")

    return df

def weighted_entity_order(high_first):

    for index, row in df.iterrows(): #Each Article
        print(index)

        tokens = df['Article'][index].split(" ")
        entities = df['Entity'][index].split(" ")
        sentences = df['Article'][index].split(" .")[:-1]

        for i,sentence in enumerate(sentences):
            sentences[i] = sentence+" ."

        len_sentence = 0
        num_entities = 0
        weights = []

        for i, token in enumerate(tokens):
            if token == ".":
                wt_sentence = num_entities/len_sentence
                weights.append(wt_sentence)
                len_sentence = 0
                num_entities = 0
            elif entities[i]!="O":
                len_sentence += 1
                num_entities += 1
            else:
                len_sentence += 1

        order = sorted(range(len(weights)), key=lambda k: weights[k],
    reverse=high_first)
        new_order = [""]*len(weights)

        for i,val in enumerate(order):
            new_order[i] = sentences[val]

        new_order = "".join(new_order)
```

```
        df["Article"][index] = new_order

    return df




















def run_samples_overnight(method_choice):
    tic = time.perf_counter()
    df = tagged_dataframe()
    shuffled = True
    num_shuffle=50
    method = method_choice

    for i in range(num_shuffle):
        header = "Sample" + str(i+1)
        df[header] = " "

    for index, row in df.iterrows():
        i=0
        cnt=0
        timeout_cnt = 0

        if shuffled==False:
            article_text = df[method][index]

            results = abs_summary(
                input_text = article_text,
                num_return_sequences=1,
                num_beams=32,
```

```
                min_length=int(article_text.count(" ")),
                temperature=4
                )

        if shuffled==True:
            while i < num_shuffle:
                current = df[method][index].split(" .")
                current = current[:-1] #removes final empty portion

                total_permutations = math.factorial(len(current))/1
                print("Article: "+str(index)+" and method: "+method)
                print("Permutations: "+str(total_permutations))
                print("Shuffle "+str(i+1))

                random.shuffle(current)
                article_text = " .".join(current)

                if i==0:
                    shuffle_list = []
                    shuffle_list.append(article_text)
                    i=i+1
                    cnt+=1

                elif article_text not in shuffle_list:
                    shuffle_list.append(article_text)
                    i=i+1
                    cnt+=1

                timeout_cnt +=1

                print("Count: "+str(cnt)+"\n")

                if cnt==total_permutations or timeout_cnt == num_shuffle
*3:
                    print("Permutation Limit Reached\n")
                    break

            #Shuffling Complete
            results=[]
            print("Generating Results!")
            for i in range(cnt):

                summary_text = abs_summary(
                    input_text = shuffle_list[i],
                    num_return_sequences=1,
```

```
                    num_beams=32,
                    min_length=int(shuffle_list[i].count(" ")),
                    temperature=4
                    )

                results.append(summary_text)

        for i in range(cnt):
            header = "Sample" + str(i+1)
            df[header][index] = results[i][0]


    #USE IF NUM SHUFFLED=0 NOT TRUE FALSE
    df.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/
    dataframes/"+method+".csv", encoding="utf-8")
    toc = time.perf_counter()

    seconds_taken = toc-tic
    minutes = seconds_taken/60
    print("Seconds: %0.4f" % seconds_taken)
    print("Minutes: %0.4f" % minutes)

    return minutes


#FOR OVERNIGHT SAVING OF FILES
#9 Hours to Run with: 4 Variations, 50 Samples
minutes_to_run = []
for method_chosen in ["Article","Tagged_All","Tagged_Uni","Tagged_One"]:
    print("Starting Method: "+method_chosen)
    mins = run_samples_overnight(method_choice=method_chosen)
    minutes_to_run.append(mins)

    gpu_usage()
    torch.cuda.empty_cache()
    torch_device="cuda"

print(sum(minutes_to_run))






##########################
```

```python
###### PARAPHRASING ######
##########################

def paraphraser(
    input_text, num_return_sequences, num_beams, max_length=60, temperature=1.5
):
    #PEGASUS XSUM

    batch2 = tokenizer2(input_text, truncation=True, padding="longest", return_tensors="pt").to(torch_device)
    translated2 = model2.generate(**batch2, temperature=temperature, max_length=max_length,
                                  num_beams=num_beams, num_return_sequences=num_return_sequences)#,do_sample=False,top_k=None)
    Pegasus = tokenizer2.batch_decode(translated2, skip_special_tokens=True)

    return Pegasus


def run_para_overnight(method_choice):
    tic = time.perf_counter()
    df = tagged_dataframe()
    num_responses=5
    method = method_choice

    for i in range(num_responses):
        header = "Sample" + str(i+1)
        df[header] = " "

    for index, row in df.iterrows():
        print("Article #"+str(index))
        new_article = []

        current = df[method][index].split(" .")
        current = current[:-1] #removes final empty portion

        for entry in current: #Sentence in Article
            results = paraphraser(
                input_text=entry,
                num_return_sequences=num_responses,
                num_beams=5,
                max_length=60,
```

```
                temperature=1)

            new_article.append(results)

        np_array = np.array(new_article)
        sentence_list = np_array.T.tolist()

        for i in range(num_responses):
            header = "Sample" + str(i+1)
            df[header][index] = " ".join(sentence_list[i])


    df.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Para_"+method+".csv", encoding="utf-8")
    toc = time.perf_counter()

    seconds_taken = toc-tic
    minutes = seconds_taken/60
    print("Seconds: %0.4f" % seconds_taken)
    print("Minutes: %0.4f" % minutes)

    return minutes, df

#FOR OVERNIGHT SAVING OF FILES
#9 Hours to Run with: 4 Variations, 50 Samples
minutes_to_run = []
for method_chosen in ["Article","Tagged_All","Tagged_Uni","Tagged_One"]:
    print("Starting Method: "+method_chosen)
    mins, df = run_para_overnight(method_choice=method_chosen)
    minutes_to_run.append(mins)

    gpu_usage()
    torch.cuda.empty_cache()
    torch_device="cuda"

print(sum(minutes_to_run))
```

NER Model

```
# -*- coding: utf-8 -*-
"""
Created on Wed Aug  3 08:15:08 2022

@author: Doug
"""

##########################
###### NER MODEL #########
##########################

#Guide: https://huggingface.co/course/chapter7/2
#Fix: https://github.com/huggingface/datasets/issues/4099
#Fix 2: https://huggingface.co/datasets/nielsr/XFUN/commit/73
    ba5e026621e05fb756ae0f267eb49971f70ebd

#################
##### CACHE #####
#################
import time
from datasets import load_dataset
from GPUtil import showUtilization as gpu_usage
import torch
#df=tagged_dataframe()
#Change Cache
import os
os.environ['TRANSFORMERS_CACHE'] = 'H:/TempHF_Cache/cache/transformers/'
os.environ['HF_HOME'] = 'H:/TempHF_Cache/cache/'
os.environ['XDG_CACHE_HOME'] = 'H:/TempHF_Cache/cache/'

from transformers import *

##############
#### CODE ####
##############
#For Variants
variants = ["Article","Tagged_One","Tagged_Uni"]
sizes = [50,100,250,500]
repetition = list(range(0,10))
#NEED TO DO ARTICLE--250--v9 LATER (not uploaded)

#For Rule-Based
rates = [0.3,0.5,0.7]
```

```
variants = ["LWTR","SR","SIS"]
sizes = [50,100,250,500]
repetition = list(range(0,10))

#For Paraphrased
#variants=["Paraphrased"]
#sizes = [50,100,250,500]
#repetition = list(range(0,10))

#Other
tic = time.perf_counter()
my_epochs = 3
time_dict = {}

for rate in rates: #Remove this and tab backwards everyhting else to do variants version (summary versions)
    for variant in variants:
        for size in sizes:
            for j in repetition:

                #TEMP TESTING
                #variant="Paraphrased"
                #size=500
                #j=0

                #For VARIANTS Version (Summarization)
                #string_path = "H:\My Files\School\Grad School WLU\MRP\Research\Files\Data\Textfiles\\"+variant+"\\"+str(size)+"\\"+variant+str(size)+"v"+str(j)+"_wikigold_split.py"
                #new_model_path = "H:\\TempHF_Cache\\TrainingArgs\\"+variant+"_"+str(size)+"v"+str(j)+"_NER_Model_"+str(my_epochs)+"Epochs_UNAUGMENTED"

                #For RULES Version (Rules-Based)
                string_path = "H:\My Files\School\Grad School WLU\MRP\Research\Files\Data\Textfiles\Rule_"+str(rate)+"\\"+variant+"\\"+str(size)+"\\0"+str(rate)[2]+variant+str(size)+"v"+str(j)+"_wikigold_split.py"
                new_model_path = "H:\\TempHF_Cache\\TrainingArgs\\0"+str(rate)[2]+"_"+variant+"_"+str(size)+"v"+str(j)+"_NER_Model_"+str(my_epochs)+"Epochs_AUGMENTED"

                #For Paraphrased Version
                #string_path = "H:\My Files\School\Grad School WLU\MRP\Research\Files\Data\Textfiles\\"+variant+"\\"+str(size)+"\\"+variant+
```

```
str(size)+"v"+str(j)+"_wikigold_split.py"
            #new_model_path = "H:\\TempHF_Cache\\TrainingArgs\\"+
variant+"_"+str(size)+"v"+str(j)+"_NER_Model_"+str(my_epochs)+"
Epochs_AUGMENTED"

            #Continue..
            cached_path = "H:\\TempHF_Cache\\cache\\datasets\\"
            raw_datasets=load_dataset(string_path, cache_dir=
cached_path)

            print(raw_datasets)

            from transformers import AutoTokenizer

            model_checkpoint = "bert-base-cased"
            tokenizer = AutoTokenizer.from_pretrained(model_checkpoint
, cache_dir="H:\TempHF_Cache\Base_Tokenizer")

            tokenizer.is_fast

            ner_feature = raw_datasets["train"].features["ner_tags"]
            ner_feature

            label_names = ner_feature.feature.names
            label_names

            #PREPROCESS DATA

            inputs = tokenizer(raw_datasets["train"][0]["tokens"],
is_split_into_words=True)
            inputs.tokens()
            inputs.word_ids()


            def align_labels_with_tokens(labels, word_ids):
                new_labels = []
                current_word = None
                for word_id in word_ids:
                    if word_id != current_word:
                        # Start of a new word!
                        current_word = word_id
                        label = -100 if word_id is None else labels[
word_id]
                        new_labels.append(label)
                    elif word_id is None:
```

```
            # Special token
            new_labels.append(-100)
        else:
            # Same word as previous token
            label = labels[word_id]
            # If the label is B-XXX we change it to I-XXX
            if label % 2 == 1:
                label += 1
            new_labels.append(label)

    return new_labels


labels = raw_datasets["train"][0]["ner_tags"]
word_ids = inputs.word_ids()
print(labels)
print(align_labels_with_tokens(labels, word_ids))


def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(
        examples["tokens"], truncation=True, is_split_into_words=True
    )
    all_labels = examples["ner_tags"]
    new_labels = []
    for i, labels in enumerate(all_labels):
        word_ids = tokenized_inputs.word_ids(i)
        new_labels.append(align_labels_with_tokens(labels, word_ids))

    tokenized_inputs["labels"] = new_labels
    return tokenized_inputs

#Takes a bit...
tokenized_datasets = raw_datasets.map(
    tokenize_and_align_labels,
    batched=True,
    remove_columns=raw_datasets["train"].column_names,
)



from transformers import DataCollatorForTokenClassification
```

```
data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)

batch = data_collator([tokenized_datasets["train"][i] for i in range(2)])
batch["labels"]

for i in range(2):
    print(tokenized_datasets["train"][i]["labels"])


#EVAL -- required pip install seqeval
import evaluate
metric = evaluate.load("seqeval")

labels = raw_datasets["train"][0]["ner_tags"]
labels = [label_names[i] for i in labels]
labels

predictions = labels.copy()
predictions[2] = "O"
metric.compute(predictions=[predictions], references=[labels])





#NOT SURE
import numpy as np

def compute_metrics(eval_preds):
    logits, labels = eval_preds
    predictions = np.argmax(logits, axis=-1)

    # Remove ignored index (special tokens) and convert to labels
    true_labels = [[label_names[l] for l in label if l != -100] for label in labels]
    true_predictions = [
        [label_names[p] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    all_metrics = metric.compute(predictions=
```

```
true_predictions, references=true_labels)
                return {
                    "precision": all_metrics["overall_precision"],
                    "recall": all_metrics["overall_recall"],
                    "f1": all_metrics["overall_f1"],
                    "accuracy": all_metrics["overall_accuracy"],
                }

            #DEFINING THE MODEL
            id2label = {str(i): label for i, label in enumerate(
label_names)}
            label2id = {v: k for k, v in id2label.items()}


            from transformers import AutoModelForTokenClassification

            model = AutoModelForTokenClassification.from_pretrained(
                model_checkpoint,
                id2label=id2label,
                label2id=label2id,
                cache_dir="H:\TempHF_Cache\Base_Model"
            )

            #Check # of Labels is Correct:
            model.config.num_labels

            from transformers import TrainingArguments

            args = TrainingArguments(
                output_dir=new_model_path,
                evaluation_strategy="epoch",
                save_strategy="epoch",
                learning_rate=2e-5,
                num_train_epochs=my_epochs,
                weight_decay=0.01,
                push_to_hub=True,
                hub_token = "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"
            )



            #TUNE
            from transformers import Trainer

            trainer = Trainer(
```

```
                model=model,
                args=args,
                train_dataset=tokenized_datasets["train"],
                eval_dataset=tokenized_datasets["test"],
                data_collator=data_collator,
                compute_metrics=compute_metrics,
                tokenizer=tokenizer,
            )

            #IF ERROR WHEN PUSHING TO HUB USE !git lfs install
            trainer.train()
            #trainer.evaluate()
            fin_results=trainer.evaluate()

            #No Longer Uploading to Hub (Glitchy, Time Waste)
            #trainer.push_to_hub()

            #Save to TXT Files (FOR PARAPHRASED)
            # output_save_path = "H:\My Files\School\Grad School WLU\MRP\Research\Files\Models\Paraphrase_Results\\"
            # with open(output_save_path+variant+str(size)+"v"+str(j)+"_Metrics.txt",'a',encoding='utf-8') as out_file:
            #     out_file.write("Precision/Recall/F1\n")
            #     out_file.write(str(fin_results["eval_precision"])+"\t"+
            #                    str(fin_results["eval_recall"])+"\t"+
            #                    str(fin_results["eval_f1"]))

            #Save to TXT Files (FOR RULES BASED)
            output_save_path = "H:\My Files\School\Grad School WLU\MRP\Research\Files\Models\Rule_Results\\"
            with open(output_save_path+str(rate)[0]+str(rate)[2]+"\\"+variant+"\\"+str(size)+"\\"+"v"+str(j)+"_Metrics.txt",'a',encoding='utf-8') as out_file:
                out_file.write("Precision/Recall/F1\n")
                out_file.write(str(fin_results["eval_precision"])+"\t"+
                               str(fin_results["eval_recall"])+"\t"+
                               str(fin_results["eval_f1"]))



            #RESET
            #gpu_usage()
            torch.cuda.empty_cache()
```

```
                torch_device="cuda"

                toc = time.perf_counter()

                seconds_taken = toc-tic
                minutes = seconds_taken/60
                print("Seconds: %0.4f" % seconds_taken)
                print("Minutes: %0.4f" % minutes)
                time_dict[variant+str(size)+"v"+str(j)]=minutes

#Test
#from transformers import pipeline
#classifier = pipeline("ner", model=model, tokenizer=tokenizer)
#classifier("My name is John Smith")
```

Baseline Augmenting

```
# -*- coding: utf-8 -*-
"""
Created on Wed Aug  3 16:08:28 2022

@author: Doug
"""

import random
import numpy as np
import copy
import pandas as pd
import csv
import nltk
from nltk.corpus import wordnet

##############
#### DF ######
##############

def gold_dataframe():
    #Create List of Articles, Tokens
    df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/wikigold.txt",
                     sep=' ', header=None, doublequote = True, quotechar='"',
                     skipinitialspace = False, quoting=csv.QUOTE_NONE)
    df.columns = ["Token","Entity"]

    current_article, current_token, article_list, token_list = [], [], [],[]

    for index, row in df.iterrows():
      if df["Token"][index] != "-DOCSTART-":
        current_article.append(df["Token"][index])
        current_token.append(df["Entity"][index])
      else:
        article_list.append(current_article), token_list.append(current_token)
        current_article, current_token = [], []

    for index in range(len(article_list)):
      article_list[index] = " ".join(article_list[index])
      token_list[index] = " ".join(token_list[index])
```

```

    #Check it worked
    for index in range(len(article_list)):
      pass
      #print(article_list[index],"\n",token_list[index])

    #Back to DF
    temp_dict = {"Article":article_list,"Entity":token_list}
    df = pd.DataFrame(temp_dict)

    return df

df = gold_dataframe()

#####################
#### ARTICLE VERS ####
#####################
import ast
sizes = [50,100,250,500]

id_dict = {}
for size in sizes:
    with open("H:\My Files\School\Grad School WLU\MRP\Research\Files\Data\Textfiles\\Article\\"+str(size)+"\\000_ID_LIST.txt") as indx_lst:
        for line in indx_lst:
            version = line[1]
            ids = ast.literal_eval(line[4:])
            id_dict[str(size)+"v"+version] = ids



#####################
##### RULE FXNS #####
#####################
def LWTR_Method(augments):
    DA_LWTR_additions = []

    for k in range(0,augments):
        DA_LWTR_word_sentences = copy.deepcopy(word_sentences)
        for i,sentence in enumerate(word_sentences):
            for j,word in enumerate(sentence):
                tag = tag_sentences[i][j]

                if np.random.binomial(1, rate, size=None):
                    substitute = random.choice(tag_groups[tag])
```

```
                    DA_LWTR_word_sentences[i][j] = substitute

            DA_LWTR_additions.append(DA_LWTR_word_sentences[i])

    #print(word_sentences[0])
    #print(DA_LWTR_word_sentences[0])
    #print(DA_LWTR_additions[50])
    #print(tag_sentences[0])
    #print(len(DA_LWTR_word_sentences))
    #print(len(DA_LWTR_additions))

    return word_sentences+DA_LWTR_additions, tag_sentences+tag_sentences*augments

def SR_Method(augments):
    DA_SR_additions = []

    SR_Issue_List = ["in","In","It","it","does","Does","IAEA","have","Have","be","Be","less","Less","He","he","Pesos","Inc","inc","acts","Acts","an","An","units"]

    for k in range(0,augments):
        DA_SR_word_sentences = copy.deepcopy(word_sentences)

        for i,sentence in enumerate(word_sentences):
            for j,word in enumerate(sentence):
                tag = tag_sentences[i][j]

                if np.random.binomial(1, rate, size=None):
                    try:
                        substitute = wordnet.synsets(word)[0].lemmas()[0].name()
                        if word not in SR_Issue_List:
                            DA_SR_word_sentences[i][j] = substitute
                    except:
                        DA_SR_word_sentences[i][j] = word

            DA_SR_additions.append(DA_SR_word_sentences[i])

        #print(word_sentences[0])
        #print(DA_SR_word_sentences[0])
        #print(tag_sentences[0])

    return word_sentences+DA_SR_additions, tag_sentences+tag_sentences*augments
```

```
def SIS_Method(augments):
    DA_SIS_additions = []

    for k in range(0,augments):
        DA_SIS_word_sentences = copy.deepcopy(word_sentences)

        for i,sentence in enumerate(word_sentences):

            previous = ''
            segment_list = []
            temp1 = []


            for j,word in enumerate(sentence):
                tag = tag_sentences[i][j]

                if tag == previous or previous == '':
                    temp1.append(word)
                elif tag != previous:
                    segment_list.append(temp1.copy())

                    temp1.clear()

                    temp1.append(word)

                previous = tag

            segment_list.append(temp1)

            #print(segment_list)

            for j,entry in enumerate(segment_list):
                if np.random.binomial(1, rate, size=None):
                    random.shuffle(entry)

            DA_SIS_word_sentences[i] = np.hstack(segment_list).tolist()
            DA_SIS_additions.append(DA_SIS_word_sentences[i])

        #print(word_sentences[0])
        #print(DA_SIS_word_sentences[0])
        #print(tag_sentences[0])

    return word_sentences+DA_SIS_additions, tag_sentences+tag_sentences*
    augments
```

```



############
###SETUP####
############
rates=[0.1,0.3,0.5,0.7]
for rate in rates:
    print(rate)
    for key in id_dict.keys():
        print(key)
        df=gold_dataframe()

        breakdown=key.split("v")
        size_choice=breakdown[0]
        version=breakdown[1]

        df = df.iloc[id_dict[key]]

        word_sentences = []
        tag_sentences = []
        for index, row in df.iterrows():

            words = df["Article"][index].split(" ")
            tags = df["Entity"][index].split(" ")

            word_segment = []
            tag_segment = []
            for i,word in enumerate(words):
                if word != ".":
                    word_segment.append(word)
                    tag_segment.append(tags[i])
                else:
                    word_segment.append(word)
                    word_sentences.append(word_segment)
                    word_segment = []

                    tag_segment.append(tags[i])
                    tag_sentences.append(tag_segment)
                    tag_segment = []



        word_sentences_flat = np.hstack(word_sentences)
        tag_sentences_flat = np.hstack(tag_sentences)
```

```

    ### MAP WORDS TO TAGS
    token_map = pd.DataFrame({"Words":word_sentences_flat, "Tags": tag_sentences_flat})
    tag_list = token_map["Tags"].unique().tolist()
    tag_groups = token_map.groupby("Tags")["Words"].apply(list)

    #######################
    #### RUN FUNCTIONS ####
    #######################
    LWTR_Results, LWTR_Tags = LWTR_Method(3)
    SR_Results, SR_Tags = SR_Method(3)
    SIS_Results, SIS_Tags = SIS_Method(3)

    ######################
    #### TO TEXT FILE ####
    ######################
    LWTR_Path = "H:\\My Files\\School\\Grad School WLU\\MRP\\Research\\Files\\Data\\Textfiles\\Rule_"+str(rate)+"\\LWTR\\"+str(size_choice)+"\\v"+str(version)+"_Augmented.txt"
    SR_Path = "H:\\My Files\\School\\Grad School WLU\\MRP\\Research\\Files\\Data\\Textfiles\\Rule_"+str(rate)+"\\SR\\"+str(size_choice)+"\\v"+str(version)+"_Augmented.txt"
    SIS_Path = "H:\\My Files\\School\\Grad School WLU\\MRP\\Research\\Files\\Data\\Textfiles\\Rule_"+str(rate)+"\\SIS\\"+str(size_choice)+"\\v"+str(version)+"_Augmented.txt"

    with open(LWTR_Path, 'w', encoding="utf-8") as LWTR_File:
        for h,sentence in enumerate(LWTR_Results): #sentence from list of sentences
            for g,word in enumerate(sentence): #word from sentence
                LWTR_File.write(LWTR_Results[h][g]+" "+LWTR_Tags[h][g]+"\n")
            LWTR_File.write("\n")

    #SR VARIANT
    with open(SR_Path, 'w', encoding="utf-8") as SR_File:
        for h,sentence in enumerate(SR_Results): #sentence from list of sentences
            for g,word in enumerate(sentence): #word from sentence
                SR_File.write(SR_Results[h][g]+" "+SR_Tags[h][g]+"\n")
            SR_File.write("\n")

    #SIS VARIANT
    with open(SIS_Path, 'w', encoding="utf-8") as SIS_File:
```

```
            for h,sentence in enumerate(SIS_Results): #sentence from list of sentences
                for g,word in enumerate(sentence): #word from sentence
                    SIS_File.write(SIS_Results[h][g]+" "+SIS_Tags[h][g]+"\n")
                SIS_File.write("\n")














#################
###PARAPHRASE####
#################
for key in id_dict.keys():
    df_og = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Original_Paraphrase.csv", encoding="utf-8", index_col=0)
    df_mp = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Mapped_Paraphrase.csv", encoding="utf-8", index_col=0)

    info = key.split("v")
    size = info[0]
    version = info[1]

    df_og = df_og.iloc[id_dict[key]]
    df_mp = df_mp.iloc[id_dict[key]]

    sentence_list = []
    tag_list = []

    #Get Sentences and Tags in Simple Form
    for index,row in df_og.iterrows():
        for header in ["Article","Sample1","Sample2","Sample3"]:
            if header=="Article":
                tokens = df_og[header][index].split(" ")
```

```
                tags = df_mp["Entity"][index].split(" ")
            else:
                tokens = df_og[header][index].split(" ")
                tags = df_mp[header][index].split(" ")

            current_sentence=[]
            current_tag=[]

            for i,token in enumerate(tokens):
                current_sentence.append(token)
                current_tag.append(tags[i])

                if token=="." or i==len(tokens)-1:
                    sentence_list.append(" ".join(current_sentence))
                    tag_list.append(" ".join(current_tag))
                    current_sentence,current_tag = [],[]

    #print(sentence_list[0])

    #Write to Txt File
    txt_file_path = "H:\\My Files\\School\\Grad School WLU\\MRP\\Research\\Files\\Data\\Textfiles\\Paraphrased\\"+str(size)+"\\v"+str(version)+"_Augmented.txt"
    with open(txt_file_path,'w', encoding="utf-8") as aug_file:
        for h,sentence in enumerate(sentence_list): #sentence from list of sentences
            for g,word in enumerate(sentence.split(" ")): #word from sentence
                aug_file.write(word+" "+tag_list[h].split(" ")[g]+"\n")
            aug_file.write("\n")
```

Mapping Functions

```
# -*- coding: utf-8 -*-
"""
Created on Tue Aug  9 08:29:19 2022

@author: Doug
"""

import pandas as pd
import nltk
nltk.download("punkt")
import csv
import random
import copy
from rouge_score import rouge_scorer
import numpy as np


def article_map():
    # load dataframe
    df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/
    Files/Data/dataframes/Article.csv", encoding="utf-8", index_col=0)
    #df_taggedsamples = df.copy(deep=True)
    # 1-gram words to be excluded (i.e. 'and' on its own should never be
    assigned a tag)
    blacklist = ['can','of','A', 'a', 'for', 'and', 'nor', 'but', 'or', '
    yet', 'so', 'both', 'and', 'whether', 'or', 'either', 'neither', 'just
    ', 'the', 'The', 'as', 'if', 'then', 'rather', 'than', 'such', 'that']



    # format samples
    for idx, row in df.iterrows():
        for column in df.iloc[:, 5:]:
            empty_sent_list=[]
            for t in nltk.sent_tokenize(df[column][idx]):
                words = nltk.word_tokenize(t)
                empty_sent_list.append(" ".join(words))

            df[column][idx] = " ".join(empty_sent_list)

    df_taggedsamples = df.copy(deep=True)

    def getNgrams(article, tags, n):
```

```

  if( n < 1 or n > len(tags) -1 ):
    raise Exception("n must be between 1 and total number of tags")

  if( len(article) != len(tags)):
    raise Exception("article length and tag length do not match")

  mapping = {}

  # sliding window of size n
  for i in range(len(tags) - n + 1):
    # collect sliding window of tags
    sequence = tags[i: i + n]
    # if the tag is real (NOT "O") and all the tags match
    if(sequence[0] != 'O' and len(set(sequence)) <= 1):
      # add the full sentence to the dictionary with a tag
      mapping[" ".join(article[i:i+n])] = tags[i]

  return mapping




# loop row by row
for idx, row in df.iterrows():
  ngrams = 5
  # loop for number of n-grams you want (currently 5)
  fullMapping = {}
  for i in range(1,ngrams):
    mapping = getNgrams(df['Article'][idx].split(' '), df['Entity'][idx].split(' '), i)
    fullMapping = {**fullMapping, **mapping}

  # remove blacklisted entries from dictionary
  for word in blacklist:
    try:
      del fullMapping[word]
    except KeyError:
      pass

  # apply the mapping to samples
  for column in df.iloc[:, 5:]:
      if df[column][idx]=='':
          continue
      # get the sample and fill entity array with no-tag ('O')
```

```
          sample = df[column][idx].split(' ')
          entities = ['O'] * len(sample)
          # iterate through each ngram length
          for i in range(1, ngrams):
              # sliding window loop for each ngram length of the full sample
              for j in range(len(sample) - i + 1):
                  sequence = sample[j: j + i]
                  # attempt to find an entity tag for the window
                  try:
                      entity = fullMapping[' '.join(sequence)]
                      # if no tag is found, do nothing
                  except KeyError:
                      pass
                  # if a tag is found
                  #replace the list of entities at the indicies of the sliding window
                  else:
                      for k in range(i):
                          entities[j+k] = entity
          # in the copied dataframe, replace the sample with the entity tags
          df_taggedsamples[column][idx] = ' '.join(entities)

    df.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Original_Article.csv", encoding="utf-8")
    df_taggedsamples.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Mapped_Article.csv", encoding="utf-8")


    return
article_map()



def paraphrase_map():
    # load dataframe
    df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Para_Article.csv", encoding="utf-8", index_col=0)
    df_taggedsamples = df.copy(deep=True)
    # 1-gram words to be excluded (i.e. 'and' on its own should never be assigned a tag)
    blacklist = [',','can','of','A', 'a', 'for', 'and', 'nor', 'but', 'or'
```

```
, 'yet', 'so', 'both', 'and', 'whether', 'or', 'either', 'neither', '
just', 'the', 'The', 'as', 'if', 'then', 'rather', 'than', 'such', '
that']



# format samples
for idx, row in df.iterrows():
    for column in df.iloc[:, 5:]:
        empty_sent_list=[]
        for t in nltk.sent_tokenize(df[column][idx]):
            words = nltk.word_tokenize(t)
            empty_sent_list.append(" ".join(words))

        df[column][idx] = " ".join(empty_sent_list)

df_taggedsamples = df.copy(deep=True)

def getNgrams(article, tags, n):

  if( n < 1 or n > len(tags) -1 ):
    raise Exception("n must be between 1 and total number of tags")

  if( len(article) != len(tags)):
    raise Exception("article length and tag length do not match")

  mapping = {}

  # sliding window of size n
  for i in range(len(tags) - n + 1):
    # collect sliding window of tags
    sequence = tags[i: i + n]
    # if the tag is real (NOT "O") and all the tags match
    if(sequence[0] != 'O' and len(set(sequence)) <= 1):
      # add the full sentence to the dictionary with a tag
      mapping[" ".join(article[i:i+n])] = tags[i]

  return mapping




# loop row by row
for idx, row in df.iterrows():
  ngrams = 5
```

```
  # loop for number of n-grams you want (currently 5)
  fullMapping = {}
  for i in range(1,ngrams):
    mapping = getNgrams(df['Article'][idx].split(' '), df['Entity'][idx].split(' '), i)
    fullMapping = {**fullMapping, **mapping}

  # remove blacklisted entries from dictionary
  for word in blacklist:
    try:
      del fullMapping[word]
    except KeyError:
      pass

  # apply the mapping to samples
  for column in df.iloc[:, 5:]:
      if df[column][idx]=='':
          continue
      # get the sample and fill entity array with no-tag ('O')
      sample = df[column][idx].split(' ')
      entities = ['O'] * len(sample)
      # iterate through each ngram length
      for i in range(1, ngrams):
          # sliding window loop for each ngram length of the full sample
          for j in range(len(sample) - i + 1):
              sequence = sample[j: j + i]
              # attempt to find an entity tag for the window
              try:
                  entity = fullMapping[' '.join(sequence)]
                  # if no tag is found, do nothing
              except KeyError:
                  pass
              # if a tag is found
              #replace the list of entities at the indicies of the sliding window
              else:
                  for k in range(i):
                      entities[j+k] = entity
      # in the copied dataframe, replace the sample with the entity tags
      df_taggedsamples[column][idx] = ' '.join(entities)

df.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Original_Paraphrase.csv", encoding="utf-8")
```

```
    df_taggedsamples.to_csv("H:/My Files/School/Grad School WLU/MRP/
    Research/Files/Data/dataframes/Finished/Mapped_Paraphrase.csv",
    encoding="utf-8")


    return
paraphrase_map()



def tagged_one_map():
    df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/
    Files/Data/dataframes/Tagged_One.csv", encoding="utf-8", index_col=0)

    headers = []
    for i in range(1,51):
        headers.append("Sample"+str(i))

    #SETUP MAP
    mapping={}
    for index,row in df.iterrows():
        list_of_words = df["Article"][index].split(" ")
        list_of_tags = df["Entity"][index].split(" ")

        previous = ''
        segment_list = []
        tag_list = []
        temp1 = []
        temp2 = []
        article_map = {"ORG":[],"PER":[],"LOC":[],"MISC":[]}

        for index2, tag in enumerate(list_of_tags):
            if tag == previous or previous == '':
                temp1.append(list_of_words[index2])
                temp2.append(tag)
            elif tag != previous:
                segment_list.append(temp1.copy())
                tag_list.append(temp2.copy())

                temp1.clear()
                temp2.clear()

                temp1.append(list_of_words[index2])
                temp2.append(tag)

```

```
            previous = tag

        segment_list.append(temp1)
        tag_list.append(temp2)

        for index3, group in enumerate(segment_list):
            segment_list[index3] = " ".join(group)
            tag_list[index3] = tag_list[index3][0]

        for index4, thingy in enumerate(segment_list):
            new_tag = tag_list[index4]
            if new_tag != "O":
                #segment_list[index4] = new_tag[2:]
                article_map[new_tag[2:]].append(thingy)

        mapping[index] = article_map


    #COPY MAP
    #mapping_reset = copy.deepcopy(mapping)
    #APPLY MAP
    # format samples
    for idx, row in df.iterrows():
        for column in df.iloc[:, 5:]:
            empty_sent_list=[]
            if df[column][idx]==' ':
                continue

            for t in nltk.sent_tokenize(df[column][idx]):
                words = nltk.word_tokenize(t)
                empty_sent_list.append(" ".join(words))

            df[column][idx] = " ".join(empty_sent_list)



    df_taggedsamples = df.copy(deep=True)

    for index,row in df.iterrows():
        for i,entry in enumerate(headers):
            if df[entry][index]==' ':
                continue
            #mapping = copy.deepcopy(mapping_reset)
            list_of_words = df[entry][index].split(" ")
            list_of_tags = df_taggedsamples[entry][index].split(" ")
```

```

        if len(list_of_words)!=len(list_of_tags):
            print("Initial Error")

        #Go Through Each Word
        for j,word in enumerate(list_of_words):
            if word in ["ORG","PER","LOC","MISC"]:
                replacement = random.choice(mapping[index][word])
                #mapping[index][word].remove(replacement)
                len_replace = len(replacement.split(" "))
                list_of_words[j] = replacement

                #print("Initial Tag: "+word)
                #print("Replacement: "+replacement)

                if len_replace == 1:
                    list_of_tags[j] = "I-"+word
                    #print("Tag Replace: "+list_of_tags[j])
                else:
                    list_of_tags[j] = str(str("I-"+word+" ")*len_replace)[:-1]
                    #print("Tag Replace: "+list_of_tags[j])
            else:
                list_of_tags[j] = "O"

            if len(list_of_words)!=len(list_of_tags):
                print("New ERROR")

            #print(list_of_words)
            #print(list_of_tags)

        df[entry][index] = " ".join(list_of_words)
        df_taggedsamples[entry][index] = " ".join(list_of_tags)


        if len(" ".join(list_of_words).split(" ")) != len(" ".join(list_of_tags).split(" ")):
            print("SECOND ERROR")

df.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Original_Tagged_One.csv", encoding="utf-8")
df_taggedsamples.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Mapped_Tagged_One.csv", encoding="utf-8")

```

```
    return
tagged_one_map()



def tagged_uni_map():
    df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/
    Files/Data/dataframes/Tagged_Uni.csv", encoding="utf-8", index_col=0)

    headers = []
    for i in range(1,51):
        headers.append("Sample"+str(i))

    #SETUP MAP
    mapping={}
    for index,row in df.iterrows():
        Entity_List = ["ORG","LOC","PER","MIS"]
        Entity_List_New = ["ORG","LOC","PER","MIS"]
        list_of_words = df["Article"][index].split(" ")
        list_of_tags = df["Entity"][index].split(" ")

        previous = ''
        segment_list = []
        tag_list = []
        temp1 = []
        temp2 = []
        article_map = {}

        for index2, tag in enumerate(list_of_tags):
            if tag == previous or previous == '':
                temp1.append(list_of_words[index2])
                temp2.append(tag)
            elif tag != previous:
                segment_list.append(temp1.copy())
                tag_list.append(temp2.copy())

                temp1.clear()
                temp2.clear()

                temp1.append(list_of_words[index2])
                temp2.append(tag)

            previous = tag

        segment_list.append(temp1)
```

```
    tag_list.append(temp2)

    for index3, group in enumerate(segment_list):
        segment_list[index3] = " ".join(group)
        tag_list[index3] = tag_list[index3][0]

    #Part That Matters
    for index4, thingy in enumerate(segment_list):
        new_tag = tag_list[index4]
        new_tag = new_tag[2:5]

        if new_tag in Entity_List:
            tag_index = Entity_List.index(new_tag)
            original_tag = Entity_List_New[tag_index]

            if original_tag[-1].isdigit():
                new_tag = original_tag[:-1]+str(int(original_tag[-1])+1)
                article_map[new_tag] = thingy
            else:
                new_tag = original_tag+"1"
                article_map[new_tag] = thingy

            Entity_List_New[tag_index] = new_tag




    mapping[index] = article_map


#COPY MAP
#mapping_reset = copy.deepcopy(mapping)
#APPLY MAP
# format samples
# format samples
for idx, row in df.iterrows():
    for column in df.iloc[:, 5:]:
        empty_sent_list=[]
        if df[column][idx]==' ':
            continue
        for t in nltk.sent_tokenize(df[column][idx]):
            words = nltk.word_tokenize(t)
            empty_sent_list.append(" ".join(words))

```

```
            df[column][idx] = " ".join(empty_sent_list)


    df_taggedsamples = df.copy(deep=True)
    #df=df.head(1)
    #headers=["Sample1"]


    for index,row in df.iterrows():
        for i,entry in enumerate(headers):
            if df[entry][index]==" ":
                continue
            list_of_words = df[entry][index].split(" ")
            list_of_tags = df_taggedsamples[entry][index].split(" ")

            if len(list_of_words)!=len(list_of_tags):
                print("Initial Error")

            #Go Through Each Word
            for j,word in enumerate(list_of_words):
                if word[:3] in ["ORG","PER","LOC","MIS"]:
                    #CATCHING NICHE CASES WHERE GEN CUTS NUMBER OFF
                    if word not in mapping[index]:
                        word = word[:3]+"1"

                    replacement = mapping[index][word]
                    len_replace = len(replacement.split(" "))
                    list_of_words[j] = replacement

                    #print("Initial Tag: "+word)
                    #print("Replacement: "+replacement)

                    if len_replace == 1:
                        list_of_tags[j] = "I-"+word[:3]
                        #print("Tag Replace: "+list_of_tags[j])
                    else:
                        list_of_tags[j] = str(str("I-"+word[:3]+" ")*
    len_replace)[:-1]
                        #print("Tag Replace: "+list_of_tags[j])
                else:
                    list_of_tags[j] = "O"

                if len(list_of_words)!=len(list_of_tags):
                    print("New ERROR")

```

```python
                #print(list_of_words)
                #print(list_of_tags)

            df[entry][index] = " ".join(list_of_words)
            df_taggedsamples[entry][index] = " ".join(list_of_tags)


            if len(" ".join(list_of_words).split(" ")) != len(" ".join(
    list_of_tags).split(" ")):
                print("SECOND ERROR")

    df.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/
    dataframes/Finished/Original_Tagged_Uni.csv", encoding="utf-8")
    df_taggedsamples.to_csv("H:/My Files/School/Grad School WLU/MRP/
    Research/Files/Data/dataframes/Finished/Mapped_Tagged_Uni.csv",
    encoding="utf-8")

    return
tagged_uni_map()










### CONFIRM THAT ALL TOKENS HAVE A MATCHING TAG
uh_oh=0
headers = []
for i in range(1,51):
    headers.append("Sample"+str(i))

for value in ["Article","Tagged_One","Tagged_Uni"]:
    df_og = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/
    Files/Data/dataframes/Finished/Original_"+value+".csv", encoding="utf
    -8", index_col=0)
    df_mp = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/
    Files/Data/dataframes/Finished/Mapped_"+value+".csv", encoding="utf-8"
    , index_col=0)

    for index,row in df_og.iterrows():
        for header in headers:
```

```
            if df_og[header][index]==" " or pd.isna(df_og[header][index]):
                continue
            else:
                og_len = len(df_og[header][index].split(" "))
                mp_len = len(df_mp[header][index].split(" "))

            if og_len!=mp_len:
                print("VARIANT: "+value+", ARTICLE: "+str(index)+", SAMPLE: "+header)
                uh_oh = 1

    if uh_oh!=1:
        print("No Issues!")






#############################
###### SCORING SECTION ######
#############################
def score_dfs():
    variants = ["Article","Tagged_One","Tagged_Uni"]

    for variant in variants:
        print(variant)
        df=pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Original_"+variant+".csv", encoding="utf-8", index_col=0)

        #Setup
        df_scores = df.copy(deep=True)
        f1_threshold = 0.2

        scorer = rouge_scorer.RougeScorer(['rouge1', 'rouge2', 'rougeL'], use_stemmer=True)

        #Set Sample Headers to Iterate
        headers = []
        for i in range(1,51):
            headers.append("Sample"+str(i))

        #Iterate and Apply Scores
        for index,row in df.iterrows():
```

```
            original = df[variant][index]

            for header_title in headers:
                if pd.isna(df[header_title][index]) or df[header_title][
    index]==" ":
                    df[header_title][index]="NA"
                    df_scores[header_title][index]="NA"
                    continue

                new = df[header_title][index]
                scores = scorer.score(original, new)
                f1_score = scores["rouge1"][2]

                if f1_score > f1_threshold:
                    df_scores[header_title][index] = f1_score
                elif f1_score < f1_threshold and new != " ":
                    df_scores[header_title][index] = "Low Score"
                else:
                    df_scores[header_title][index] = "NA"

        #df.replace('N/A',pd.NA)
        df_scores.to_csv("H:/My Files/School/Grad School WLU/MRP/Research/
    Files/Data/dataframes/Finished/Scored_"+variant+".csv", encoding="utf
    -8")

    return
score_dfs()
```

8.1.2 Secondary Code Files

Dataset Statistics

```
# -*- coding: utf-8 -*-
"""
Created on Sat Aug  6 22:12:28 2022

@author: Doug
"""

################
# LIBRARIES
################

import pandas as pd
import csv
import numpy as np
import tensorflow as tf
import time

#For Copies
import copy

#For Synonyms
import requests
from bs4 import BeautifulSoup

#For Shuffle
import random

#Hugging Face
from transformers import PegasusTokenizer, PegasusForConditionalGeneration
    , T5Tokenizer, T5ForConditionalGeneration, MT5Tokenizer,
    MT5ForConditionalGeneration, AutoModel, AutoTokenizer

#OS
import os.path
from os import path as os_path

#ITERTOOLS
import itertools

#ROUGE METRIC
from rouge_score import rouge_scorer
```

```

#FACTORIAL
import math

#NLTK
import nltk
nltk.download("punkt")

#MEMORY CLEARING
from GPUtil import showUtilization as gpu_usage
import torch
from numba import cuda

#################
##### CACHE #####
#################

#Change Cache
import os
os.environ['TRANSFORMERS_CACHE'] = 'H:/TempHF_Cache/cache/transformers/'
os.environ['HF_HOME'] = 'H:/TempHF_Cache/cache/'
os.environ['XDG_CACHE_HOME'] = 'H:/TempHF_Cache/cache/'

################
# CODE
################

def gold_dataframe():
    #Create List of Articles, Tokens
    df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/
    Files/Data/wikigold.txt",
                     sep=' ', header=None, doublequote = True, quotechar='
    "',
                     skipinitialspace = False, quoting=csv.QUOTE_NONE)

    df.columns = ["Token","Entity"]

    current_article, current_token, article_list, token_list = [], [], []
    ,[]

    for index, row in df.iterrows():
      if df["Token"][index] != "-DOCSTART-":
        current_article.append(df["Token"][index])
        current_token.append(df["Entity"][index])
      else:
```

```
        article_list.append(current_article), token_list.append(current_token)
        current_article, current_token = [], []

    for index in range(len(article_list)):
      article_list[index] = " ".join(article_list[index])
      token_list[index] = " ".join(token_list[index])

    #Back to DF
    temp_dict = {"Article":article_list,"Entity":token_list}
    df = pd.DataFrame(temp_dict)
    return df

df=gold_dataframe()

def tagged_dataframe():
    df=gold_dataframe()
    df["Tagged_All"]=df["Article"]
    df["Tagged_One"]=df["Article"]
    df["Tagged_Uni"]=df["Article"]

    #List of Entities
    Entity_List = ["ORG","LOC","PER","MISC"]

    for index, row in df.iterrows():

        list_of_words = df["Tagged_All"][index].split(" ")
        list_of_tags = df["Entity"][index].split(" ")

        ###TAGGING ALL
        word_sentences = []
        tag_sentences = []

        word_segment = []
        tag_segment = []

        for i,word in enumerate(list_of_words):
            if word != ".":
                word_segment.append(word)
                tag_segment.append(list_of_tags[i])
            else:
                word_segment.append(word)
                word_sentences.append(word_segment)
                word_segment = []

```

```python
                tag_segment.append(list_of_tags[i])
                tag_sentences.append(tag_segment)
                tag_segment = []

        for i,sentence in enumerate(word_sentences):
            for j,word in enumerate(sentence):
                tag = tag_sentences[i][j]
                if tag != "O":
                    word_sentences[i][j] = tag[2:]

        for i,sentence in enumerate(word_sentences):
            word_sentences[i] = " ".join(sentence)

        df["Tagged_All"][index] = " ".join(word_sentences)





        #TAGGING SEGMENTS AS ONE
        previous = ''
        segment_list = []
        tag_list = []
        temp1 = []
        temp2 = []

        for index2, tag in enumerate(list_of_tags):
            if tag == previous or previous == '':
                temp1.append(list_of_words[index2])
                temp2.append(tag)
            elif tag != previous:
                segment_list.append(temp1.copy())
                tag_list.append(temp2.copy())

                temp1.clear()
                temp2.clear()

                temp1.append(list_of_words[index2])
                temp2.append(tag)

            previous = tag

        segment_list.append(temp1)
        tag_list.append(temp2)

```

```
        for index3, group in enumerate(segment_list):
            segment_list[index3] = " ".join(group)
            tag_list[index3] = tag_list[index3][0]

        #print(segment_list)
        #print(tag_list)

        for index4, thingy in enumerate(segment_list):
            new_tag = tag_list[index4]
            if new_tag != "O":
                segment_list[index4] = new_tag[2:]


        df["Tagged_One"][index] = " ".join(segment_list)








        #TAGGING UNIQUE
        Entity_List_New = Entity_List.copy()
        list_of_words_tagged = df["Tagged_One"][index].split(" ")
        #list_of_words = df["Article"][index].split(" ")
        #If time, make numbering unique i.e. if band shows up twice give same # for it

        for i, word in enumerate(list_of_words_tagged):
            if word in Entity_List:
                Tag_Index = Entity_List.index(word)
                Original_Tag = Entity_List_New[Tag_Index]

                if Original_Tag[-1].isdigit():
                    New_Tag = Original_Tag[:-1]+str(int(Original_Tag[-1])+1)
                else:
                    New_Tag = Original_Tag+"1"

                Entity_List_New[Tag_Index] = New_Tag
                list_of_words_tagged[i] = New_Tag

        df["Tagged_Uni"][index] = " ".join(list_of_words_tagged)

    return df
```

```






#############
### STATS ###
#############
df = tagged_dataframe()

WPA = []
SPA = []
EPA = {"O":[],"I-ORG":[],"I-PER":[],"I-LOC":[],"I-MISC":[]}
CPA = []

for index, row in df.iterrows():

    list_of_words = df["Article"][index].split(" ")
    list_of_tags = df["Entity"][index].split(" ")

    list_of_sentence_words = df["Article"][index].split(". ")

    words_per_article = len(list_of_words)
    sent_per_article = len(list_of_sentence_words)

    WPA.append(words_per_article)
    SPA.append(sent_per_article)

    for key in EPA.keys():
        EPA[key].append(list_of_tags.count(key))


    cnt = 0
    for word in list_of_words:
        if word in ["(",")",".",",","'",'"',"[","]","-","{","}",":",";"]:
           cnt+=1

    CPA.append(cnt)

sum(CPA)
sum(EPA["I-ORG"]+EPA["I-LOC"]+EPA["I-PER"]+EPA["I-MISC"])

SPA_unique = np.unique(SPA).tolist()
SPA_counts = [SPA.count(num) for num in SPA_unique]
```

```
WPA_unique = np.unique(WPA).tolist()
WPA_counts = [WPA.count(num) for num in WPA_unique]

#####################
#### ARTICLE VERS ####
#####################
import ast
sizes = [50,100,250,500]

id_dict = {}
```

```
for size in sizes:
    with open("H:\My Files\School\Grad School WLU\MRP\Research\Files\Data\
    Textfiles\\Article\\"+str(size)+"\\000_ID_LIST.txt") as indx_lst:
        for line in indx_lst:
            version = line[1]
            ids = ast.literal_eval(line[4:])
            id_dict[str(size)+"v"+version] = ids


set_vers = "Train"
df=gold_dataframe()
sent_length_dict = {}
art_cnt_dict = {}
O_cnt_dict = {}
NO_cnt_dict = {}
for size in [50,100,250,500]:
    batch_avg_sent_length=[]
    batch_tot_sent_length=[]
    vers_art_cnt = []
    vers_O_cnt = []
    vers_NO_cnt = []

    for version in range(0,10):
        index_ids_to_use = id_dict[str(size)+"v"+str(version)]

        if set_vers == "Test":
            index_ids_to_use = list(set(list(range(0,145)))-set(
    index_ids_to_use))

        new_df=df.iloc[index_ids_to_use]

        sentence_lengths = []
        article_cnt = 0
        O_tag_cnt = []
        NO_tag_cnt = []
        for index,row in new_df.iterrows():
            sentence_lengths.append(SPA[index])
            article_cnt+=1

            tag_splitter = new_df["Entity"][index].split(" ")
            O_tag_cnt.append(tag_splitter.count("O"))
            NO_tag_cnt.append(tag_splitter.count("I-PER")+tag_splitter.
    count("I-LOC")+tag_splitter.count("I-ORG")+tag_splitter.count("I-MISC"
    ))

```

```python
        vers_art_cnt.append(article_cnt)
        tot_sentences = sum(sentence_lengths)
        avg_sentence_length = sum(sentence_lengths)/len(sentence_lengths)
        batch_avg_sent_length.append(avg_sentence_length)
        batch_tot_sent_length.append(tot_sentences)

        vers_O_cnt.append(sum(O_tag_cnt))
        vers_NO_cnt.append(sum(NO_tag_cnt))

    avg_batch_sent_length = sum(batch_avg_sent_length)/len(batch_avg_sent_length)
    tot_batch_sent_length = sum(batch_tot_sent_length)/len(batch_tot_sent_length)
    tot_batch_art_cnt = sum(vers_art_cnt)/len(vers_art_cnt)
    avg_O_cnt = sum(vers_O_cnt)/len(vers_O_cnt)
    avg_NO_cnt = sum(vers_NO_cnt)/len(vers_NO_cnt)

    art_cnt_dict[size]=tot_batch_art_cnt
    sent_length_dict[size]=tot_batch_sent_length
    O_cnt_dict[size]=avg_O_cnt
    NO_cnt_dict[size]=avg_NO_cnt

print(sent_length_dict)
print(art_cnt_dict)
print(O_cnt_dict)
print(NO_cnt_dict)
```

```
##############
#### PLOT ####
##############
#https://matplotlib.org/stable/gallery/statistics/histogram_multihist.html

import matplotlib.pyplot as plt
```

```
#SENTENCES
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
langs = SPA_unique
students = SPA_counts
ax.bar(langs,students)
plt.xlabel("Sentences in Articles")
plt.ylabel("# of Occurences")
plt.title('Frequency Distribution - # of Sentences in Articles')
plt.text(15, 8, 'Avg # of Sentences per Article: '+str(round(sum(SPA)/len(
    SPA),2)))
plt.show()

#WORDS
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
langs = WPA_unique
students = WPA_counts
ax.bar(langs,students)
plt.xlabel("Sentences in Articles")
plt.ylabel("# of Occurences")
plt.title('Frequency Distribution - # of Sentences in Articles')
plt.text(6, 3, 'Avg # of Sentences per Article: '+str(round(sum(SPA)/len(
    SPA),2)))
plt.show()


#ENTITIES
colors = ["red","blue","green","black","orange"][1]
entity_lab = list(EPA.keys())[1]
entity_cnt = list(EPA.values())[1]
fig = plt.figure()
plt.hist(entity_cnt, density=False, bins=50, stacked=True, color=colors,
    label=entity_lab)
plt.legend(prop={'size': 10})
plt.ylabel('# of Occurences')
plt.xlabel('# of Words per Article');
plt.show()
```

Train-Test Split

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Aug  9 15:21:27 2022

@author: Doug
"""
################
# LIBRARIES
################

import pandas as pd
import csv
import numpy as np
import tensorflow as tf
import time

#For Copies
import copy

#For Synonyms
import requests
from bs4 import BeautifulSoup

#For Shuffle
import random

#Hugging Face
from transformers import PegasusTokenizer, PegasusForConditionalGeneration
    , T5Tokenizer, T5ForConditionalGeneration, MT5Tokenizer,
    MT5ForConditionalGeneration, AutoModel, AutoTokenizer

#OS
import os.path
from os import path as os_path

#ITERTOOLS
import itertools

#ROUGE METRIC
from rouge_score import rouge_scorer

#FACTORIAL
import math
```

```python

#NLTK
import nltk
nltk.download("punkt")

#MEMORY CLEARING
from GPUtil import showUtilization as gpu_usage
import torch
from numba import cuda

#################
##### CACHE #####
#################

#Change Cache
import os
os.environ['TRANSFORMERS_CACHE'] = 'H:/TempHF_Cache/cache/transformers/'
os.environ['HF_HOME'] = 'H:/TempHF_Cache/cache/'
os.environ['XDG_CACHE_HOME'] = 'H:/TempHF_Cache/cache/'

################
# CODE
################

df = pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/
    Data/dataframes/Article.csv", encoding="utf-8", index_col=0)

WPA = []
SPA = []
EPA = {"O":[],"I-ORG":[],"I-PER":[],"I-LOC":[],"I-MISC":[]}
CPA = []

for index, row in df.iterrows():

    list_of_words = df["Article"][index].split(" ")
    list_of_tags = df["Entity"][index].split(" ")

    list_of_sentence_words = df["Article"][index].split(". ")

    words_per_article = len(list_of_words)
    sent_per_article = len(list_of_sentence_words)

    WPA.append(words_per_article)
    SPA.append(sent_per_article)

```

```
    for key in EPA.keys():
        EPA[key].append(list_of_tags.count(key))


    cnt = 0
    for word in list_of_words:
        if word in ["(",")",".",",","'",'"',"[","]","-","{","}",":",";"]:
           cnt+=1

    CPA.append(cnt)

sum(CPA)
sum(EPA["I-ORG"]+EPA["I-LOC"]+EPA["I-PER"]+EPA["I-MISC"])

SPA_unique = np.unique(SPA).tolist()
SPA_counts = [SPA.count(num) for num in SPA_unique]

WPA_unique = np.unique(WPA).tolist()
WPA_counts = [WPA.count(num) for num in WPA_unique]





def rng_train_set(min_exclusion, max_exclusion, sentence_goal, bounds_rate):
    if min_exclusion<0 or max_exclusion<0 or min_exclusion>=max_exclusion or sentence_goal<0 or bounds_rate<0:
        print("Error in Variable Settings! Exited.")
        return

    in_bounds = False

    while in_bounds == False:
        original_article_list = list(range(0,145))

        filter_out_list = []
        filter_cnt = 0
        for i,length in enumerate(SPA):
            if length <= min_exclusion or length >= max_exclusion:
                filter_out_list.append(i)
                filter_cnt += length

        new_article_list = list(set(original_article_list)-set(filter_out_list))
```

```

        new_cnt = 0
        for index in new_article_list:
            new_cnt+=SPA[index]

        #Confirm
        if filter_cnt+new_cnt!=1768 and len(filter_out_list)+len(new_article_list)!=145:
            print("ERROR IN SENTENCE OR ARTICLE COUNT")

        #Get Avg SPA
        avg_spa = new_cnt / len(new_article_list)

        expected_iterations = math.ceil(sentence_goal/avg_spa)
        train_articles = random.sample(new_article_list, expected_iterations)
        test_articles = list(set(original_article_list)-set(train_articles))


        total_sentences = 0
        for article_index in train_articles:
            total_sentences += SPA[article_index]

        avg_sentences = total_sentences/len(train_articles)

        #Check Bounds Acceptance
        if bounds_rate >= 0 and bounds_rate < 1:
            if total_sentences >= sentence_goal*(1-bounds_rate) and total_sentences <= sentence_goal*(1+bounds_rate):
                in_bounds = True
        else:
            if total_sentences >= sentence_goal - bounds_rate and total_sentences >= sentence_goal + bounds_rate:
                in_bounds = True

    return train_articles, test_articles, total_sentences, avg_sentences#, filter_out_list, new_article_list, avg_spa

#Run Replications for Sample Sizes
replications = 10
batch_pools = [50,100,250,500]
train_batch_dict = {50:[],100:[],250:[],500:[]}
test_batch_dict = {50:[],100:[],250:[],500:[]}

```

```
for size in batch_pools:
    print("Batch Size: "+str(size))
    for i in range(replications):
        train_list, test_list, cnt_sentences, avg_sentences =
    rng_train_set(
            min_exclusion=2
            , max_exclusion=40
            , sentence_goal=size
            , bounds_rate=0.1
            )

        train_batch_dict[size].append(train_list)
        test_batch_dict[size].append(test_list)










#Select Scored Samples
def final_selection(augmented_sentence_multiplier):
    variants = ["Article","Tagged_One","Tagged_Uni"]
    augment_multiple=3#augmented_sentence_multiplier

    #Set Sample Headers to Iterate
    headers = []
    for i in range(1,51):
        headers.append("Sample"+str(i))

    nested_dict = {}
    for variant in variants:
        nested_dict[variant] = {}
        for size in train_batch_dict:
            nested_dict[variant][size] = {}
            for attempt in range(replications):
                nested_dict[variant][size][attempt] = {}
                df=pd.read_csv("H:/My Files/School/Grad School WLU/MRP/
    Research/Files/Data/dataframes/Finished/Original_"+variant+".csv",
    encoding="utf-8", index_col=0)
                df_scored = pd.read_csv("H:/My Files/School/Grad School
    WLU/MRP/Research/Files/Data/dataframes/Finished/Scored_"+variant+".csv
    ", encoding="utf-8", index_col=0)
```

```
            df=df.iloc[train_batch_dict[size][attempt]]

            for index,row in df.iterrows():
                possible_shuffles = math.factorial(SPA[index])
                NA_samples = list(df_scored.iloc[index][headers]).count("NA")
                LowScore_samples = list(df_scored.iloc[index][headers]).count("Low Score")

                max_possible = min(50-NA_samples-LowScore_samples, possible_shuffles)
                samples_to_take = min(SPA[index]*augment_multiple, max_possible)

                score_list = list(df_scored.iloc[index][headers])
                for i,score in enumerate(score_list):
                    if score == "NA" or score == "Low Score":
                        score_list[i]=0
                    else:
                        score_list[i]=float(score_list[i])

                index_order = sorted(range(len(score_list)), key=lambda k: score_list[k], reverse=True)
                final_list = index_order[0:samples_to_take]
                final_list = ["Sample"+str(x+1) for x in final_list]

                cleaned_list=[]
                for j,sample in enumerate(final_list):
                    if df[sample][index]!=" " or df[sample][index]!="NA" or not pd.isnull(df[sample][index]):
                        #print(variant+"-"+str(size)+"-"+str(attempt)+"-Article ID "+str(index)+sample)
                        cleaned_list.append(sample)
                        #break

                #nested_dict[variant][size][attempt][index] = final_list
                nested_dict[variant][size][attempt][index] = cleaned_list


            #print(nested_dict[variant][size][attempt][index])
            #print("\n")

```

```
    return nested_dict

fin_results = final_selection(3)










##########################
###### CREATE TXTS ######
##########################
full_list = list(range(0,145))

for variant in list(fin_results.keys()): #Variants (Article, One Tag, Unique Tag)
    #Set DFs
    df_og=pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Original_"+variant+".csv", encoding="utf-8", index_col=0)
    df_mp=pd.read_csv("H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/dataframes/Finished/Mapped_"+variant+".csv", encoding="utf-8", index_col=0)

    for size in list(fin_results[variant].keys()): #Sizes (50/100/250/500)
        #Write Article IDs for Reference per Repetition Version
        with open('H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/Textfiles/'+variant+'/'+str(size)+'/000_ID_LIST.txt', 'w', encoding="utf-8") as f_ids:
            for repetition in list(fin_results[variant][size].keys()): #Repetition (0-->10)
                #Write TOKEN/TAGs to 3 text files, test, original, and original+augmented
                with open('H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/Textfiles/'+variant+'/'+str(size)+'/v'+str(repetition)+'_Augmented.txt', 'w', encoding="utf-8") as f_aug, open('H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/Textfiles/'+variant+'/'+str(size)+'/v'+str(repetition)+'_UnAugmented.txt', 'w', encoding="utf-8") as f_org, open('H:/My Files/School/Grad School WLU/MRP/Research/Files/Data/Textfiles/'+variant+'/'+str(size)+'/v'+str(repetition)+'_Testing.txt', 'w', encoding="utf-8") as f_tst:
                    f_ids.write("v"+str(repetition)+":\t")
```

```python
                f_ids.write(str(list(fin_results[variant][size][repetition].keys())))
                f_ids.write("\n")

                #Iterate through Articles, Then Samples
                for article_id in list(fin_results[variant][size][repetition].keys()): #Article IDs
                    #Get the IDs for Articles
                    train_ids = list(fin_results[variant][size][repetition].keys())
                    test_ids = list(set(full_list)-set(train_ids))

                    #token_list_train = []
                    #tag_list_train = []

                    df_og_train = df_og.iloc[train_ids]
                    df_mp_train = df_mp.iloc[train_ids]
                    df_og_test = df_og.iloc[test_ids]
                    #df_mp_test = df_mp.iloc[test_ids]

                    df_og_train=df_og_train.fillna("NA")
                    df_mp_train=df_mp_train.fillna("NA")
                    df_og_test=df_og_test.fillna("NA")
                    #df_og_train(replace)

                    #Get Augmented Samples
                    sample_list = fin_results[variant][size][repetition][article_id]
                    if sample_list != []:
                        for sample in sample_list:
                            #SKIP NAs
                            if df_og_train[sample][article_id]=="NA":
                                continue

                            tokens_train = df_og_train[sample][article_id].split(" ")
                            tags_train = df_mp_train[sample][article_id].split(" ")

                            if tokens_train[-1]!=".": tokens_train.append("."), tags_train.append("O")#, print("Added a period on article "+str(article_id))

                            #token_list_train.append(tokens_train)
                            #tag_list_train.append(tags_train)
```

```

                            #CONFIRM NO MISMATCH
                            for index,entry in enumerate(tags_train):
                                if len(tags_train)!=len(tokens_train):
                                    print("MISMATCH on ARTICLE "+str(article_id)+" and "+sample)

                            for index,the_token in enumerate(tokens_train):
                                to_append = the_token+" "+tags_train[index]
                                f_aug.write(to_append)
                                f_aug.write("\n")

                                if the_token==".":
                                    f_aug.write('\n')

                            f_aug.write("-DOCSTART- O\n\n") #SWAP BACK TO "-DOCSTART- O\n\n" LATER
                    #f_aug.write("-DOCSTART- O\n\n") #SWAP BACK

                    #GET ORIGINAL UN-AUGMENTED SAMPLES FOR TRAINING
                    #GET ORIGINAL UN-AUGMENTED SAMPLES FOR TRAINING
                    #GET ORIGINAL UN-AUGMENTED SAMPLES FOR TRAINING
                    tokens_train = df_og_train["Article"][article_id].split(" ")
                    tags_train = df_og_train["Entity"][article_id].split(" ")
                    if tokens_train[-1]!=".": tokens_train.append("."), tags_train.append("O")#, print("Added a period on article"+str(article_id))
                    #CONFIRM NO MISMATCH
                    for index,entry in enumerate(tags_train):
                        if len(tags_train)!=len(tokens_train):
                            print("MISMATCH on ARTICLE "+str(article_id)+" and "+sample)
                    for index,the_token in enumerate(tokens_train):
                        to_append = the_token+" "+tags_train[index]
                        f_aug.write(to_append)
                        f_aug.write("\n")
                        f_org.write(to_append)
                        f_org.write("\n")

                        if the_token==".":
                            f_aug.write('\n')
```

```
                            f_org.write('\n')

                    f_aug.write("-DOCSTART- O\n\n") #SWAP BACK TO "-DOCSTART- O\n\n" LATER
                    f_org.write("-DOCSTART- O\n\n") #SWAP BACK TO "-DOCSTART- O\n\n" LATER

                #GET ORIGINAL UN-AUGMENTED SAMPLES FOR TESTING
                #GET ORIGINAL UN-AUGMENTED SAMPLES FOR TESTING
                #GET ORIGINAL UN-AUGMENTED SAMPLES FOR TESTING
                for a_index,row in df_og_test.iterrows():
                    tokens_train = df_og_test["Article"][a_index].split(" ")
                    tags_train = df_og_test["Entity"][a_index].split(" ")
                    if tokens_train[-1]!=".": tokens_train.append("."), tags_train.append("O")#, print("Added a period on article"+str(article_id))
                    #CONFIRM NO MISMATCH
                    for index,entry in enumerate(tags_train):
                        if len(tags_train)!=len(tokens_train):
                            print("MISMATCH on ARTICLE "+str(a_index)+" and "+sample)
                    for index,the_token in enumerate(tokens_train):
                        to_append = the_token+" "+tags_train[index]
                        f_tst.write(to_append)
                        f_tst.write("\n")

                        if the_token==".":
                            f_tst.write('\n')

                    f_tst.write("-DOCSTART- O\n\n") #SWAP BACK TO "-DOCSTART- O\n\n" LATER
```