August 2020

Dictionary-based Data Generation for Fine-Tuning Bert for Adverbial Paraphrasing Tasks

Mark Anthony Carthon
University of Wisconsin-Milwaukee

DICTIONARY-BASED DATA GENERATION FOR FINE-TUNING BERT FOR ADVERBIAL PARAPHRASING TASKS

by

Mark Carthon III

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Mathematics

at

The University of Wisconsin-Milwaukee

August 2020

ABSTRACT

DICTIONARY-BASED DATA GENERATION FOR FINE-TUNING BERT FOR ADVERBIAL PARAPHRASING TASKS

by

Mark Carthon III

The University of Wisconsin-Milwaukee, 2020
Under the supervision of Professor Istvan Lauko

Recent advances in natural language processing technology have led to the emergence of large and deep pre-trained neural networks. The use and focus of these networks are on transfer learning. More specifically, retraining or fine-tuning such pre-trained networks to achieve state of the art performance in a variety of challenging natural language processing/understanding (NLP/NLU) tasks. In this thesis, we focus on identifying paraphrases at the sentence level using the network Bidirectional Encoder Representations from Transformers (BERT). It is well understood that in deep learning the volume and quality of training data is a determining factor of performance. The objective of this thesis is to develop a methodology for algorithmic generation of high-quality training data for paraphrasing task, an important NLU task, as well as the evaluation of the resulting training data on fine-tuning BERT to identify paraphrases. Here we will focus on elementary adverbial paraphrases, but the methodology extends to the general case. In this work, training data for adverbial paraphrasing was generated utilizing an Oxford

synonym dictionary, and we used the generated data to re-train BERT for the paraphrasing task with strong results, achieving a validation accuracy of 96.875%.

To

Yahweh,

Sam,

my loved ones,

and all those who supported me

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI Artificial Intelligence

UWM University of Wisconsin - Milwaukee

MT Machine Translation

NLP Natural Language Processing

NLU Natural Language Understanding

BERT Bidirectional Encoder Representations from Transformers

IBM International Business Machines

ALPAC Automatic Language Processing Advisory Committee

ETAOINSHRDLU Frequency order of letters in the English Language

ATN Augmented Transition Network

LSTM Long short-term memory

SYSTRAN System Analysis Translator

RNN Recurrent Neural Network

FFNN Feed Forward Neural Network

MARGIE Memory Analysis Response Generation in English

APE(X)C All Purpose Electric (X) Computer

OTAZ Oxford Thesaurus A to Z

OT Oxford Thesaurus

RE Regular Expressions

GPU Graphics Processing Unit

TPU Tensor Processing Unit

CoLab Collaboratory

NER Named Entity Recognition

NEL Named Entity Labeling

History of Natural Language Processing (NLP)

The field of NLP comes from a field called Machine Translation (MT). Machine Translation is the study of how to algorithmically translate one language to another. One of the first people to work in Machine Translation was a Persian intellectual named Al-Kindi (circa 801 AD - 873 AD). Al-Kindi was born in a city named Kufa but was educated in Bagdad. He was a prominent figure in the Grand Library of Bagdad. This was a public academy and the intellectual center of Bagdad during the Islamic Golden Age. Several Abbasid caliphs appointed Al-Kindi to oversee the translation of Greek scientific and philosophical texts into the Arabic language. Al-Kindi developed various methods to do this. These methods relied on the frequency of the characters that were used in the texts. He developed techniques for systematic language translation including, but not limited to: cryptanalysis, frequency analysis, probability theory, and statistics, all of which are used in modern machine translation.

In 1623, Rene Descartes (and others) proposed an artificial universal language; in this language, equivalent ideas in different languages would share the same symbol. Both Leibniz and Descartes put forward proposals for codes which would relate words between languages. Of course, all these proposals were purely theoretical, and none of them resulted in the development of an actual algorithm for language translation. During the mid-1930s when the first patents for ”translating machines” were applied for, one proposal by Georges Antsrouni was simply an

automatic bilingual dictionary using paper tape. Paper tape was a form of data storage that consisted of a long strip of paper in which holes were punched. Paper tape was used mostly during the 19th and 20th centuries. They were effective with teleprinter communication as input for computers of the 1950s and 1960s, and later as a storage medium for minicomputers and CNC machine tools. Another proposal was made by Peter Troyanski, a Russian who included a bilingual dictionary, and a method for dealing with grammatical roles between languages based on Esperanto.

Esperanto was the most widely spoken constructed international auxiliary language. It was created by a Polish ophthalmologist called Ludwik Lejzer Zamenhof in 1887. The purpose of this universal language Esperanto was to create a flexible language that would serve as a universal second language to foster world peace, international understanding, and to build a ”community of speakers”. Peter Troyanski’s proposal was a system that was separated into three stages. The first stage consisted of a native-speaking editor in the source language to organize the words into their logical forms and to exercise the syntactic functions. A logical form of a syntactic expression is a precisely specified semantic version of that expression in some formal system. Think of this first stage as tokenization for BERT. The second stage of Troyanski’s proposal was for the machine (such a machine did not exist when this proposal was initially made) to translate these logical forms into the target language. Stage three required a native-speaking editor to normalize this output. This system did not become relevant on a big scale until the 1950s when computers became widely utilized.

In 1946, Andrew Donald Booth, a British electrical engineer, physicist, and computer scientist, proposed the idea of using digital computers for translation of natural languages. Warren Weaver was a researcher at the Rockefeller Foundation, and he presented one of the first set of computer-based machine translation proposals in 1949; it was called ”Translation Memorandum”. Warren Weaver proposals came about thanks to advancements in information theory, successes in cryptography during World War II, and theories about invariants at the foundation of natural language. In 1950, a breakthrough was made by Alan Turing when he published his famous paper ”Computing Machinery and Intelligence” which proposed the ”Turing Test” as a criterion of intelligence. This criterion depends on the ability of a computer program to impersonate a human in a real-time written conversation with a human judge, sufficiently well that the judge is unable to reliably distinguish between the computer program and a real human on the basis of conversational content alone.

In 1951, Yehosha Bar-Hillel began research in MT and later claimed that without a ”universal encyclopedia” a machine would never be able to deal with the problem of one word having multiple definitions because during this time, semantic ambiguity could only be solved by writing source texts for machine translation in a controlled language that uses a vocabulary in which each word has exactly one meaning. Also, in 1954, Georgetown University created an MT research team. On January 7, 1954 Georgetown and IBM teamed up to give a public demonstration of its Georgetown-IBM experiment system which involved fully automatic translation of more than 60 Russian sentences into English. This demonstration was widely reported in the newspapers and gained public interest. However, this feat was limited because it had only 250 words and translated 49 carefully selected Russian sentences into English. Most of

the words were related to the field of Chemistry. Despite its shortcomings, this demonstration encouraged others to put more resources into furthering MT on a worldwide scale.

Andrew Donald Booth designed the All Purpose Electronic (X) Computer, APE(X)C, at Birkbeck College (now the University of London). In 1954 a demonstration was made on the APE(X)C of a rudimentary translation of English into French. In 1955, Japan and Russia began to develop their own MT research programs and in 1956 the first ever MT conference was held in London. In 1957, Noam Chomsky, an American linguist developed the idea of syntactic structures which was a work that sought to prove that semantics and syntax are two separate concepts. He did this by forming the sentence ”Colorless green ideas sleep furiously”, which shows that semantics and syntax really are different.

In the 1960s, some NLP processing systems were developed, such as: SHRDLU which was a natural language system that worked in restricted ”block worlds” with restricted vocabularies. This block world was a breakthrough in Artificial Intelligence (AI) which made it easier for systems to model and reason about abstract symbols. Also, ELIZA was another NLP processing system which was a simulation of a psychotherapist. This was written by Joseph Weizenbaum who was a German American computer scientist from MIT. ELIZA was able to provide human-like interaction whenever the patient was saying things within ELIZA’s small database. This was quite remarkable considering that ELIZA was given no information about human thought or emotion.

During the 1960s, The United States and the Soviet Union did research in Russian - English translation of scientific journals and technical articles. This endeavor was successful in that the rough translations helped to give a basic understanding of the material. In 1962, the Association for Machine Translation and Computational Linguistics was formed in the U.S.A. and many researchers joined it. Research in MT continued until the US Government commissioned the Automatic Language Processing Advisory Committee (ALPAC) report. This committee was made up of seven scientists who decided that there was a lack of progress in the last ten years of MT research despite significant funds being allocated toward this research area.
This ALPAC report concluded that MT was more expensive, less accurate, and slower than a human translator. This report also concluded that MT was unlikely to make very much progress towards human-level translation. Consequently, MT funding was greatly reduced.

There was a beacon of hope though because this ALPAC report recommended that tools (i.e. automatic dictionaries) be developed to aid human translators and that some research in computational linguistics should continue to receive funding and support. Because of this report, MT research was practically abandoned for over a decade. This had a ripple effect as the Soviet Union and the United Kingdom decided to cut back on their MT research efforts as well. However, Canada, France, and Germany continued to research in MT. In 1969, Roger Schank, an American A.I. theorist, introduced the conceptual dependency theory for NLU; this model was partially inspired by the work of Sydney Lamb, an American linguist. The biggest efforts in MT in the US, during the period after the ALPAC, was given by Systran and Logos; these companies did MT work for the US Department of Defense.

In 1970, the Systran system was installed for the Us Air Force, it was used by the Commission of the European Communities in 1976, and Xerox used Systran to translate technical manuals in 1978. Also, in 1970, William A. Woods developed the augmented transition network (ATN) to represent natural language input. These ATNs were graph-theoretic structures that are used in formal languages. Instead of phrase structure rules, ATNs used an equivalent set of recursively called finite state automata. Eventually, these ATNs were generalized into so-called ’generalized ATNs”.

In 1970, The French Textile Institute used MT to translate abstracts between French, English, German, and Spanish. In 1971, Brigham Young University (BYU) started a project to translate Mormon texts automatically. During the 1970s, it was common for computer programmers to write ’conceptual ontologies’, which structured real-world information into data that was of such a form that a computer would be able to understand what it was. Some examples of this is: MARGIE (Schank, 1975), SAM (Cullingford, 1978), and many more. In the 1960s, MT research concentrated on limited language pairs and input, but in the 1970s, the concentration was on low-cost systems that could translate technical and commercial documents. Up to the 1908s, most NLP systems were based on complex sets of handwritten rules.

By the 1980s, both the diversity and the number of installed systems for machine translation had increased. Several systems that relied on mainframe technology were in use (i.e. Systran and Logos). Systran’s first implementation system was implemented in 1988 by the French Postal Service’s online service called Mintel. There was a market for lower-end machine translation systems because of the much-improved availability of microcomputers. Thus, microcomputers

became more common in Europe, Japan, USA, China, Korea, and the Soviet Union. Japan had a fifth-generation computer and tried to create software for translating between English and many other Asian languages.

During the 1980s, translation relied on intermediary linguistic representation such as: syntactic, morphological, and semantic analysis. In the late 1980s, computational power increased in line with Moore's Law and the fact that researchers were relying on Chomskyan theories less because its theory discouraged the kind of corpus linguistics that is at the foundation of the machine learning approach to language processing. As a result, computational power also became less expensive. Also, more research and attention was allocated towards statistical models for machine translation. Some of the earliest used machine learning algorithms (i.e. decision trees) produced systems with difficult if-then rules to resolve and maintain which were similar to existing hand-written rules, but the statistical models made soft, probabilistic decisions based on attaching real-valued weights to the features of the input data.

In the 1990s, MT began to move away from large mainframe computers and towards personal computers and workstations. Recent research has focused more on unsupervised and semi-supervised learning algorithms. These algorithms learn from data that has not been annotated or a combination of non-annotated and annotated data with the desired answers given and with the creation of the World Wide Web, lots of non-annotated data is available. In the 2010s, representation learning and deep neural network machine learning methods became widespread in NLP. These methods have been shown to produce state-of-the-art results in many NLP tasks such as: language modeling, parsing, etc. Many state-of-the-art techniques use word embeddings

to capture semantic properties of words and an increase in end-to-end learning of higher-level tasks.

Applications of NLP

Natural Language Processing (NLP) is a sub-topic of Artificial Intelligence and its purpose is to train computers to understand, interpret, read, hear, and communicate using human (natural) languages. Neural networks have long been associated with NLP research, but their use has become increasingly practical, widespread, and dominant in applications in both image and language processing from the early 2000s. In 1997, a variety of neural network architectures such as Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) models were introduced and found great success with NLP tasks, voice and text recognition, and text and speech generation. RNN are iterative models used in sequence to sequence translation, use previous outputs as inputs for future iterations of the algorithm. In 2001, Yoshio Bengio and his team proposed the first neural language model using a feed-forward neural network (FFNN). The big difference between RNNs and FFNNs are that RNNs use previous outputs in an iterative fashion as future input, but the simpler architecture FFNNs do not, the data is feed forward, an output is produced, then the new input is fed forward and so on.

In 2011, Apple released Siri which became known as the world's first successful NLP assistant to be used by the public. The Automated Speech Recognition module in Siri can translate the user's words into digitally interpreted tokens/concepts. Siri has predefined commands within its software and uses the tokens it receives from the user's speech and matches it up with one or more of its predefined commands. Then it carries out the said commands. Machine Learning techniques are necessary because NLP engines need to recognize a command even if the

command is not expressed using the exact sequence of tokens that the engine has been programmed with; it has to allow for some freedom of expression while still capturing the main idea of the user's words and matching it to the predefined set of commands.

In November 2014, Amazon announced Alexa alongside Echo. The inspiration for Alexa stems came from the computer voice and conversational system on the Starship Enterprise in Sci-fi works such as Star Trek. One of the reasons the Amazon developers chose the name Alexa is because x is a hard consonant; hard consonants are recognized with higher precision than other types of letters. The Amazon developers also liked the name Alexa because it was reminiscent of the Library of Alexandria, which is where the field of MT began (the precursor to NLP).

Google Assistant was unveiled during Google's developer conference on May 18, 2016. It was part of the unveiling of the Google Home smart speaker and a new messaging app called Allo. Google's CEO claimed that Google Assistant was designed to be a conversational experience, a two-way experience, and "an ambient experience that extends across devices".

Motivation for Data Generation

The goal of this thesis is to investigate the facilitation of the training of large neural networks to learn and understand (i.e.: develop the capacity to identify and differentiate) grammatical structure and extended vocabulary, including a wide range of idiomatic language use in the English language. More specifically, we would aim to re-train a Bidirectional Encoder Representations from Transformers (BERT) neural network to enhance its capacity to understand the language well enough to be able to identify paraphrases for a large variety of sentences. The task of computational paraphrase identification, a high-level artificial intelligence task in the language understanding domain (NLU), is to evaluate whether the two sentences in a sentence pair are semantically equivalent.

BERT is constructed as a large and deep neural network with novel network structures (attention mechanism) to be able to encode longer range morphological and semantic relations within a text. It has several variants in terms of network size as well as training language data used, but in all variants it has over 100 million “learnable “ network parameters to be adjusted in an optimization process. BERT has been trained in a supervised training fashion on two tasks for which generations of annotated data is easy to produce: Masked Language Modeling and Next Sentence Prediction, and it has been demonstrated that the network has a strong capacity to be trained of fine-tuned to produce state of the art performance in a wide range of NLP tasks (transfer learning).

To either train from scratch or to retrain/fine tune for close to human performance on a complex task such large pre-trained networks in a supervised training regime, extensive amount of labeled training data is needed. For strong performance of the targeted trained network in a general NLP setting the training data deployed need to represent a rich variety of the targeted language use. Training such large systems is expected to require millions of annotated training samples that show a representative distribution of the targeted language use. Such richness in the training data would be key to improve the performance of state-of-the-art systems in automated language processing and language understanding, but currently for most tasks it is not available in the necessary volume and quality.

Development of the needed training data sets with the current methods (i.e.: based on hand-labeling by humans) are prohibitively expensive: it would require hundreds of thousands of expert human workhours or possibly substantially more. Thus, current industry standard data sets for training and evaluation for several NLP tasks are relatively small. For instance the General Language Understanding Evaluation (GLUE) benchmark's MRPC component to evaluate a network's capacity to train/perform for the task of paraphrasing is relatively small (on the order of 5000 sentences) and it is also restricted to a limited a (journalistic) language domain. One of the objectives in this thesis is to explore a method to generate a large annotated training dataset for fine tuning a trained network in a supervised learning setting for the task of paraphrase identification, i.e. to decide if two sentences are semantically identical, or correspond to each other as paraphrases.

We base our paraphrase sample generation method on content extracted from an extensive synonym dictionary: The Oxford Thesaurus, An A-Z Dictionary of Synonyms, Oxford University Press (1994) (OTAZ). This dictionary puts a focus on compiling complete synonymic word and expression sets with an emphasis on representing rarer word and expressions uses, associated with semantic content of synonym sets. That information is very hard for networks to learn from corpus (even from billion-word corpora) due to very low frequency use even in very large text collections (corpus). However, some common synonyms may have been omitted from the collection to limit the volume of the original publication. We expect that the most common synonymic uses are well-represented in the training data that was originally used for training BERT; thus, they are already represented in the models we aim to fine-tune.

Another attractive option as a source for the adverbial, adjective, noun and verbal synonym sets could be WordNet database, however WordNet does not contain rarer part of speech sets (such as Interjections, Conjunctions,...) of words and expressions and is more limited in the number of example sentences listed to illustrate semantic function. Additionally, to its polysemous synonymic organization of its headwords, OTAZ provides additional stylistic and usage information on the synonym sets and importantly a rich set of example sentences (1-4 example sentences per synonym sets) using the headword/expression of each synonym set entry. The idea of sentence pair generation using lexicographical resources for enhanced training of neural language networks for classification of sentence pairs relies on substitution of synonymic words and expressions into example sentences from thesauruses, synonym dictionaries and similar lexicographical resources.

In OTAZ fundamentally such a substitution task poses challenges of varying complexities depending on the part of speech categories the synonym sets belong to. The difficulties are ranging from the simple, (e.g.: for the case of adverbial synonym sets with very little to no morphological variation induced with substitution), to the sophisticated in the case of synonymic verbal expressions that are required to be adjusted in number, tense and attracted noun cases during a substitution step to provide correct sentence pairs. In this work, for simplicity, we limit our investigation for generation and training use of adverbial synonym sets only. There was not a substantial amount of data online for download, so we had to create our own training data. We used text data from the Oxford synonym dictionary cited above.

In the format of this dictionary assigned to a given word or expression heading was its part-of-speech, a list of synonyms or synonymic expressions, and one or more (up to four) example sentences. Using regular expression as our editing tool, the data for adverbial synonyms was converted to a text file with each example sentence on a separate line paired with its head expression's synonymic set -. Then we used regular expressions to organize the text, and to generate alternate matching sentence paraphrases by substitution of the marked expression heading with its synonymic alternatives. This provided on average about 10 paraphrase alternatives for each example sentence. As network training would require a balanced set of negative as well as positive examples, we also produced non-matching paraphrases from each example sentence, by associating it with a synonym set that were not matching the marked (head) expression in the example and performing the substitutions indicated above.

The substitution steps were performed with an iterative application of a complex regular expressions; the matching and replacement steps are as outlined below. These test processing methods are computationally highly efficient. After substitution we formed all possible sentence pairing in both paraphrase and non-paraphrase sentence matchings and attached an appropriate label to each resulting sentence pair (1 for paraphrase and 0 for non-paraphrase pairs).

This procedure generated approximately 45000 paraphrase pairs and with a similar amount of non-paraphrase pairs we obtained a set of close to 90000 tagged training sentence pairs. These pairs were generated from the approximately 1000 adverbial synonym set and example sentence combinations that are found in OTAZ. We emphasize that these samples are produced varying adverbial expressions only. This paraphrase dataset is rich enough to finetune large pretrained neural networks (e.g. BERT) for the recognition of adverbial paraphrasing.

In the generation process, we used a set of special characters to delineate the structure of the text database that we formulated from OTAZ. After completing the substitution, pairing steps, and eliminating the auxiliary characters from the results, we obtained our training data. This dataset is structured as a pair of sentences with a binary label to indicate whether the sentences are paraphrases of each other; this dataset was in a tab-separated format. This text document was loaded with the base BERT network model to perform fine-tuning.

Regular Expressions

Regular expressions are powerful text pattern recognition codes/software tools that can be very efficiently implemented by programs (so-called Regex engines) based on the theory of finite automata. A regular expression is an algebraic formula whose value is a pattern identifying of a set of strings that are matched by the expression, called the language of the expression. Regular expression engines are a built-in part of most programming languages as well as more advanced text editors and are essential components of software that performs text mining and high volumes of sophisticated text processing.

A regular expression defines a character sequence search pattern. Usually such patterns are used by string searching algorithms for ”find” or ”find and replace” operations on strings, or for input validation. It originated in the 1950s and a mathematician from the U.S., Stephen Cole Keene, is given credit for creating it. Here are some basic components that are use in regular expressions:

Normal text or character sequences:

abc matches the string abc; *Note: regex (language) special characters ()[]*^${}.?\- need to be escaped, i.e. preceded by the character “\”, in a regular expression to be matched within a normal text*

Anchors : ^ and $

^ matches the beginning of a line (a zero-length string at the beginning of a line)

$ matches the end of a line (a zero-length string at the end of a line)

Repetition Quantifiers : * + ? and {}

*	zero or more times:	abc*	matches a string that has ab followed by zero or more c
+	one or more times:	abc+	matches a string that has ab followed by one or more c
?	zero or one times:	abc?	matches a string that has ab followed by zero or one c
{n}	n times:	abc{2}	matches a string that has ab followed by 2
{n}	n or more times:	cabc{2,}	matches a string that has ab followed by 2 or more c
{n,m}	n to m times:	abc{2,5}	matches a string that has ab followed by 2 up to 5 c
(text)	grouping/capturing:	a(bc)*	matches a string that has a followed by zero or more

copies of the sequence bc and captures the sequence bc

OR operator: | or []

a(b|c) alternate expressions: matches a string that has a followed by b or c (and captures b or c)

a[bc] alternate characters: same as previous, but without capturing b or c

a[^bc] character exclusion: matches a string that has a followed one character that is not b or c

Character classes: \d \w \s and .

\d matches a single character that is a digit; same as [0-9]

\w matches a word character (alphanumeric character plus underscore); same as [a-z0-9_]

\s matches a whitespace character (includes tabs and line breaks); same as [\t\r\n]

. matches any (one) character

For example, if you have a text document and you want to search for a word character followed by a digit one or more times, you will use the regexp: \w\d+. We used regular expressions to ”clean up” the text documents that we wanted to use for training. We deleted all unnecessary and undesirable characters in the text, we rearranged words in the text, and we created multiple examples sentences from the given example sentence using regular expressions. Regexp made this task easier because it allowed us to make complex edits on very large text documents without having to manually go through single lines. We were able to check that everything worked in the desired way by changing a few lines at a time, first, then once we were confident that we tested and verified the correctness a specific edit, we would change the entire document.

While editing large text files with regular expressions it is imperative that one is very familiar with the document and to follow best practices. Our original data to be edited is well-structured text database organized on a line per record basis and special characters are used to delimit fields within each line, thus it is well-suited for editing using Regex.

A snippet of the reformulated lexicographical database obtained from OT is represented below. The database stored in a text file is line based with Unicode character sets. Here D, @, &, %, # and ~ represent specially selected, special separator Unicode characters not appearing in the text-based data of our database which are used as special markers. Each paragraph in the

representation below, headed by a lead expression and ending with one or more example sentences, shows an adverbial synonym set and its use. To a single lead expression there may be associated several alternative use sense or synonym set. Each paragraph below is in fact a (long) line in the database ending in a newline character. This representation allows us to use regular expressions for complex data generation very efficiently.

Below are several lines of the reformulated data from the OTAZ with special characters used as markers:

Ðabove+♥adv.♦1*╕above, overhead, on_high, aloft, in_the_sky, in_the_heavens, <Tab>: Far aboveΣ, the clouds scudded swiftly by.♪

Ðabroad+♥adv.♦1*╕abroad, overseas, in_foreign_lands, in_foreign_parts, <Tab>: We were abroadΣ on assignment for a few years.♪

Ðabsolutely+♥adv.♦1*╕absolutely, unqualifiedly, unconditionally, unreservedly, unexceptionally, unequivocally, unquestionably, positively, definitely, really, genuinely, decidedly, surely, truly, certainly, categorically, <Tab>: She is absolutelyΣ the best dancer I have ever seen.♪

Ðabsolutely+♥adv.♦1*╕absolutely, unqualifiedly, unconditionally, unreservedly, unexceptionally, unequivocally, unquestionably, positively, definitely, really, genuinely, decidedly, surely, truly, certainly, categorically, <Tab>:: I absolutelyΣ refuse to go.♪

Ðabsolutely+♥adv.♦2*╕absolutely, totally, utterly, completely, entirely, fully, quite, altogether, wholly, <Tab>: It is absolutelyΣ necessary that you undergo surgery.♪

Ðaccordingly+♥adv.♦1*╕accordingly, hence, therefore, consequently, thus, in_consequence_of, in_consequence_whereof, so, and_so, <Tab>: Smoking was forbidden; accordinglyΣ, we put out our cigars.♪

Ðaccordingly+♥adv.♦2*╕accordingly, suitably, in_conformity, in_compliance,÷ conformably, appropriately, compliantly, <Tab>: Dinner-jackets were required, and the men dressed accordinglyΣ.♪

Ðactually+♥adv.♦1*╕actually, really, in_reality, in_fact, in_actuality, in_point_of_fact, in_truth, absolutely, as_a_matter_of_fact, indeed, truly, literally, <Tab>: The interest rates actuallyΣ charged by banks may vary from those quoted publicly.♪

In later discussion, we will refer to the replacement procedure based on the following entry as an example; the delineating characters are shown in blue:

```
Ðabout+♥adv.♦4*╕about, here_and_there, far_and_wide, hither_and_yon, hither_and_thither, helter-skelter, <Tab>: My papers were scattered aboutΣ as if a tornado had struck.♪
```

Structure of Transformer Networks

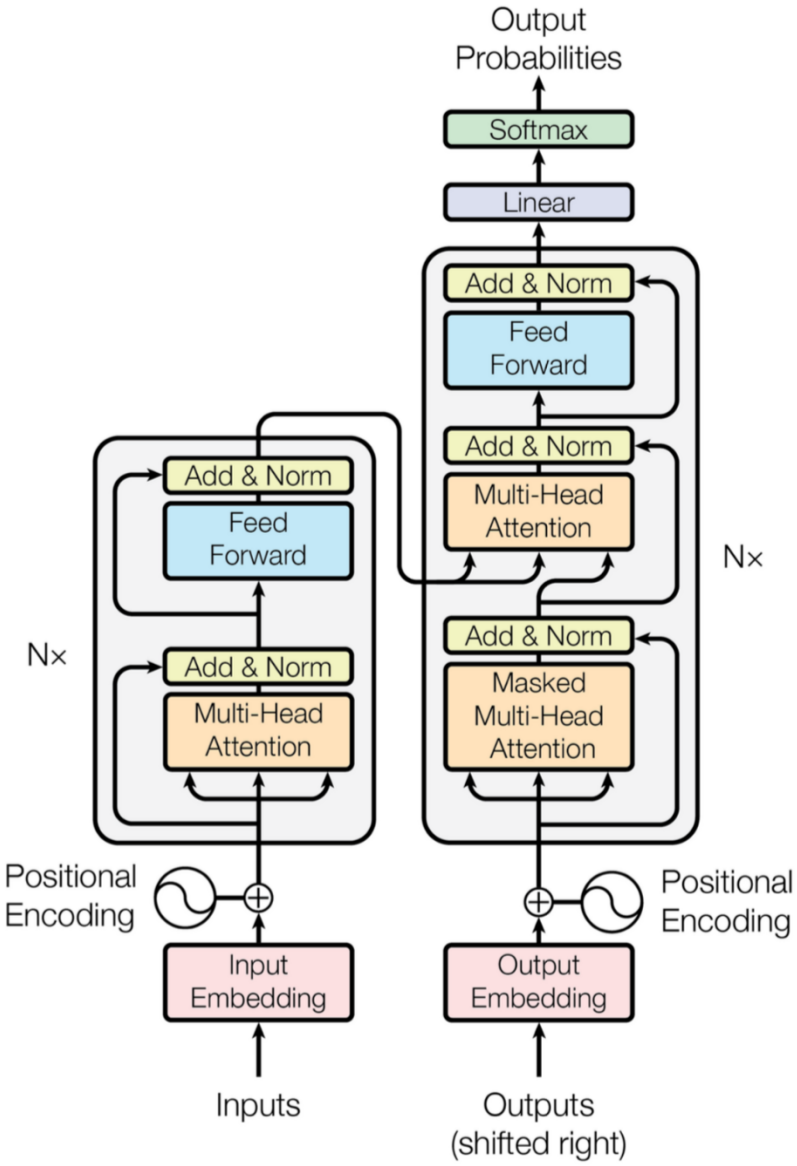

Figure 1: The Transformer - model architecture.

The BERT network is a type of Transformer network. A Transformer network is a large neural network that has both encoders and decoders (BERT only has encoders) and was introduced in 2017 by Vaswani et al. in the well-known paper titled ”Attention is All You Need”. Transformer networks are designed to efficiently identify association patterns between distant terms in sequences and are designed to be trained for sentence translation and other NLP tasks. The input could be a French sentence, for example, and the Transformer based on a large set of training examples that is usually referred as a corpus, would be trained to translate the French sentence into English, and then output the English translation. The architecture of the Transformers is composed of a stack of encoders and a stack of decoder network components. The encoders basically take the French sentence and transform it into the language of the network , and then the last encoder feeds this information into the stack of decoders. The decoders take the information from the last encoder and they turn the information from the language internal to the network into the desired language output for the user (which is the corresponding English language sentence in this example). The underlying network architecture in a distributed fashion (i.e. assigned to each artificial neuron) contains a very large set of parameters (numbering in the millions for large and deep state-of-the-art networks) that are identified in a large-scale optimization exercise, called training, based on available training examples. This demanding computation exercise requires highly parallelized hardware architectures such as GPUs or TPUs.

Encoders in a Transformer can turn the input into language the model understands via a technique called word embeddings. These word embeddings are the process that the network takes to take a given word and turn it into a vector of a certain size. When using the Transformer

network, the word embeddings all have the same size. These vectors contain the word and some of its meaning. For example, the word ’King’ and the word ’Queen’ each have their own vector from the word embeddings. If you take the vector for King and subtract the vector for Queen, the difference will be a vector that is ”small” in magnitude. These word embeddings are possible after a network has been trained. So, the encoders use word embeddings to gather the necessary information from the input.

Each of the encoders has a structure to it. In each encoder, there is a Self-Attention mechanism and a feed-forward network. The self-attention mechanism was invented in the paper ”Attention is All You Need”. The goal of 11 attention is to help the network to understand the connection that each of the words have with each other. For example, in the sentence: The dog took a bone a buried it in the ground. What does the word ’it’ refer to in this sentence? Someone who is fluent in English would know that the word ’it’ would refer to the bone. The goal of attention is to help the network to understand that the word it refers to the bone. Therefore, attention is so important; for the network to properly understand the sentences, it must understand how each word in a sentence relates to all the other words.

How does the self-attention work? Self-attention requires a few things: word embeddings, queries, keys, and values. For each word embedding, a query, key, and value vector is created for it. Once a network has been trained, the network will have a weight matrix W_Q for the queries, a weight matrix for the keys W_K, and a weight matrix W_V for the values. Each word embedding vector is multiplied by each of the weight matrices. This would produce a query vector q_i, a key vector k_i, and a value vector v_i corresponding to the word embedding x_i. Next

you take $q_i \prod_i k_i$. Then you do $\frac{q_1 k_1}{+\sqrt{dim(k)}}$ and take the "SoftMax" of it. This SoftMax normalizes the constant so that it is between 0 and 1 (a probability). Then you take each value vector times the SoftMax score, then take the sum of all of them. This will give you a weighted sum of all the value vectors. This calculation is very similar for matrices. Here it is:

$$X \times W_Q = Q$$

$$X \times W_K = K$$

$$X \times W_V = V$$

$$softmax\left(\frac{Q \times K^T}{+\sqrt{d_k}}\right) \times V = Z$$

In this calculation, X is the input, Q is the query, K is the key, V is the value, Z is the output/head.

There is also the notion of multi-headed attention. This notion helped the network's attention layer to improve its performance. With multi-headed attention, the network produces multiple attention heads or multiple Z matrices. In this process we have multiple $Q/V/K$ matrices. In the Transformer model, we have 8 attention heads. This will give us

$$Z_0, Z_1, \ldots, Z_7.$$

Since the model only expects one attention head, we concatenate all the Z's together to get Z then multiply Z by another weight matrix W which was trained by the model to give us a Z of the desired dimensions. Why would multi-head attention be useful? Consider the sentence, "Dave bought a phone, but didn't know how to operate it." What is the word "it" referring to? Of course, we humans are fluent enough in language to know that the word "it" refers to the phone, but the model is not as fluent in language as we are. The multi-head attention aids the network to

learn that the word ”it” refers to the phone. As we all know, the order of words in a sentence is important. The network has a way of understanding the position of each token in a sentence by using a ”positional encoding” that it learns through training. There are some residuals in the encoders. What happens is that the input X is fed into the encoder, self-attention is performed, the result of the self-attention Z is taken then the network computes X + Z, normalizes it, then feeds it into the feed-forward part of the encoder. The result of the feed-forward part of the encoder Z' is added to Z and normalized. Then this result Z'' is fed into the next encoder and the process repeats. Once our original input has made its way through all the encoders, it proceeds to the decoder.

The structure of the decoder is as follows: The output of the encoder is fed into the decoder. The decoder takes this through self-attention, adds and normalizes, then does an encoder-decoder attention, then adds and normalizes, then goes through the feed-forward, then adds and normalizes, then repeats this process as it gets pushed into the next decoder. Once the original output of the encoder layers gets pushed through all the decoders, it goes through a linear layer, then gets its SoftMax score. This linear layer is what takes the vector that the decoders spit out and turns it into one of the tokens in its dictionary. It does this by projecting the output vector from the decoders into a logits vector whose length is the number of words in the given language that it knows. The SoftMax is used to help the network decide which of the words is the most probable.

BERT, a transformer neural network is the product of several NLP advancements, such as, Semi-Supervised Sequence Learning, ELMo, ULMFiT, the OpenAI transformer, and the Transformer. It is a transformer that only has encoders and no decoders.

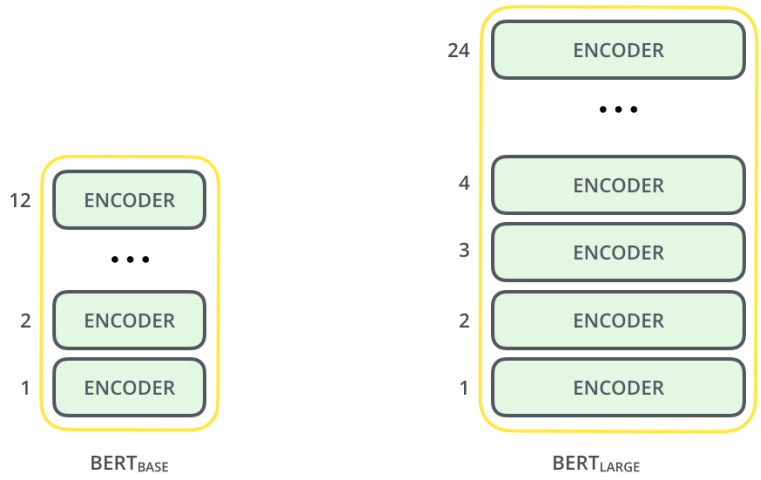

Figure 2: BERT architecture

BERT has been pre-trained on massive datasets, such as, all of Wikipedia and many books. BERT is mostly used for sentence classification and sentiment classification. As an example, we can see BERT used for spam classification outlined in Figure 3.

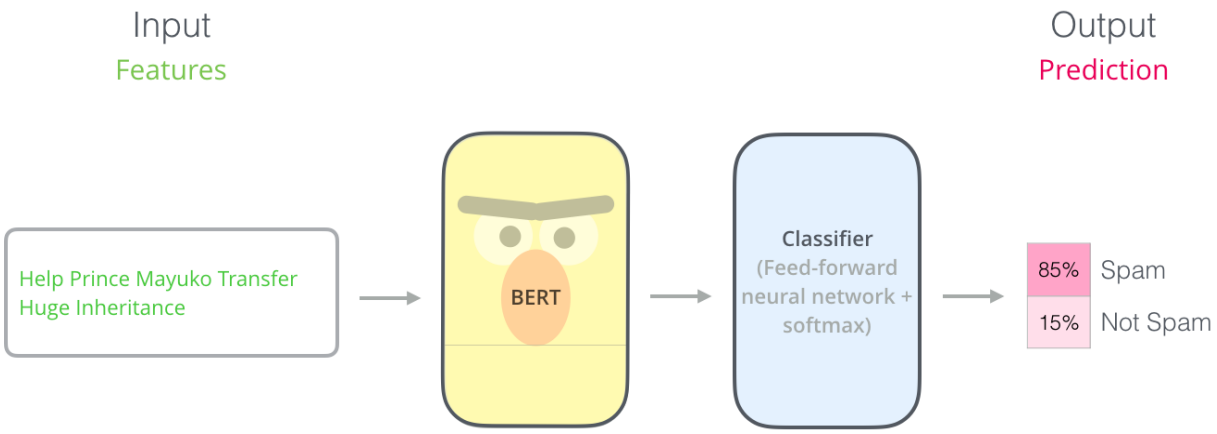

Figure 3: BERT performing spam classification

BERT takes a sentence as input, performs a word-embedding technique on all of the tokens, feeds these tokens through its encoders (see Figure 2), then sends it through a classifier to output a classification based on the probability of the input being spam or not spam.

In addition to spam classification, BERT can also do fact-checking at a high level. The tokenization process for BERT means that it takes words or parts of words that it recognizes from its training and turns them into word-embeddings. A word embedding is a vectorized text encoding, i.e. when a token is turned into a relatively high-dimensional vector representation. Usually, a word embedding is a vector of size 300 for a given token; a token is usually a word, but a token could also be one part of a word, or a hyphenated set of words, etc. These word embedding vectors have been developed internally by BERT itself for the purpose of identifying words (or pairs of words) that can be assembled to represent words from its dictionary. This word-embedding process allows it to perform vector/matrix calculations for the test data mentioned previously in this chapter. BERT's input, along with word/token embedding also represents the token's position within the text as well as representations of sentences; this concept is relevant for multi-sentence tasks. Our paraphrase classification problem relies on this multi-sentence tokenization.

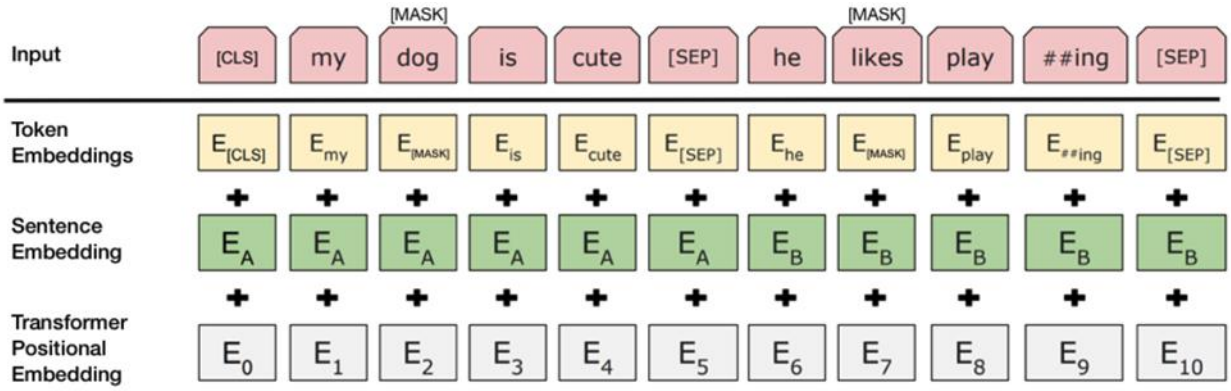

Figure 4: BERT input representation

The input is compressed into a sequence of vectors. These vectors enter the network at its first parametrized layer of encoders. The encoders process these vectors in a series of highly parallelizable numerical linear algebra steps within the layer. Then this result is fed into a second layer of an identically structured parametrized layer. This process is repeated several times (12 times in the base BERT network). This leads to progressive information processing by the network. The output of the last layer is typically processed by a FFNN component that is trained to assign probabilities to the token positions. These probabilities measure the likeliness that a given token in the input belongs to a defined ranged of output classes. This process is well-suited for sequence translation, Named Entity labeling, and Named Entity Recognition (NER).

In classification tasks, the last encoder produces a special token called the [CLS] token or classification token. This token is fed into a simple FFNN component that is trained to assign probabilities to the input tokens. These probabilities measure the likeliness of a given input token belonging to a defined range of output classes.

The training of such a network is a (large) optimization exercise. The aim of this exercise is to identify a set of network parameters that maximize the network's performance. The network wants to maximize its performance in classifying the training data while ensuring that it has the capacity to generalize to examples it has not seen before. In other words, the network wants to maximize its classification performance on the data it has seen without overfitting. If this is achieved, then BERT has been shown to perform very well in classification tasks.

Data Generation

The goal of this project was to train a BERT network to be able to recognize when a pair of sentences that contained alternatives of adverbial words or expressions were paraphrases of each other. The first task was to create data for a BERT Transformer, as outlined above. This data needed to be example sentence pairs such that a sentence pair would be labelled with a 1 if the sentences presented were a paraphrase of one another. If one has a sentence pair such that neither sentence is a paraphrase of the other, this pair is labelled with 0. In other words, the goal is to create a large number of sentence pairings of substantial linguistic variety as data to train a neural network to learn to recognize paraphrases.

Such data is not readily available in the required quality and quantity and is very expensive to be produces by humans. The Oxford synonym dictionary (OT) references earlier contains heading words or expressions, synonyms to those headings, and example sentences using those words. An example of a line given in the dictionary is as follows:

about adv.: about, here and there, far and wide, hither and yon, hither and thither, helter-skelter: My papers were scattered about as if a tornado had struck.

The presentation of every word in this synonym dictionary follows this basic structure. This looked like a promising source for data. As such, this entire dictionary was scanned and entered

in a text editor for cleaning. Various special characters were inserted to make the cleaning and generation process easier. For instance, the above example was changed to:

Ðabout+♥adv.♦4*╕about, here_and_there, far_and_wide, hither_and_yon, hither_and_thither, h elter-skelter, : My papers were scattered aboutΣ as if a tornado had struck.♪

As you can see, the Ð marks the beginning of a new word, the + marks the end of the word, the ♥ marks the beginning of the part of speech, the ♦ marks the end of the part of speech, the number four denotes how many times this has been used (because many words have multiple meanings and each word that is in the dictionary has a separate synonym sequence for each different meaning it has), the * marks the end of the number, the ╕ marks the beginning of the first synonym. Multiword expressions have underscores to separate each word (replace spaces within the expression), although some terms are separated by hyphens, for example, helter-skelter. Each synonym is followed by a comma, even if it is the last synonym in the list. Each example sentence follows a colon and ends with a punctuation mark and a character ♪ (musical note). A tab always separates the synonyms and the first example sentence. These characters make it easier to do regular expressions on the text because many of the symbols used are rarely seen in dictionaries and thus, are unlikely to clash with anything in the text originally. If ! was used instead of one of the special characters, there would have been a higher probability of error with the regular expressions because ! is a more commonly used character in dictionaries.

After the dictionary has been cleaned up and organized by the special characters, sample sentences are created. For this, a regular expression search pattern was created:

^([^\n╕ \t]+)╕ ([^\n╕ :÷\t,]+)([,\t]÷* *)([^\n╕ :\t]+)(\t:+)([^\n:]*)([^\n:]+Σ+)([^\n]+)♪([^\n]*)\n

along with the following regular expression replacement pattern:

\1\2\3╕ \4\5\6\7\8♪\9\5\6\2τ\8\n

The following explains the search pattern:

^ means to start at the beginning of the line.

The captured expression ([^\n╕ \t]+) means that the aim is to collect everything that is not an end of line, nor a ╕ , nor a tab if it occurs one or more times. This unit is referenced by \1 that appears in the replacement pattern.

╕ ([^\n╕ :÷\t,]+) means that a ╕ will be matched preceding the regexp in the parenthesis, which is referenced by 2 in the replacement pattern. \2, matches everything after ╕ that is not an end of line, nor a ╕ , nor a :, nor a ÷, nor a tab, nor a comma if it occurs one or more times. This pattern captures the first synonym in the list, while \1 captures everything up until the ╕ .

([,\t]÷* *) is denoted by \3 and it collects everything until the next comma or tab, whichever occurs first. In this instance, \3 captures the next synonym in the list.

([^\n╕ :\t]+) means \4 will matches everything after the second synonym that is not an end of line, nor a ╕ , nor a :, nor a tab if it occurs one or more times. In other words, \4 will match everything after the second synonym up until the tab preceding the example sentence.

(\t:+) means \5 will collect the tab and colon that occurs one or more times, then stop preceding the example sentence.

([^\n:]*) means \6 will collect everything that is not an end of line, nor a colon if it occurs zero or more times. It will stop immediately before the original synonym including the whitespace preceding the first character of the original synonym. In other words, \6 collects everything in the example sentence preceding the first character of the “keyword”.

([^\n:]+Σ+) means \7 will collect the keyword, which conveniently has a sigma at the end of it. Since it is possible for some keywords to have more than one sigma, Σ+ is used to collect sigmas if it occurs one or more times. It is important that Σ marks the keyword only in the original example sentence.

([^\n]+) means \8 will collect everything after the “keyword” but stops before the character ♪. In other words, \8 collects the rest of the example sentence.

♪([^\n]*)\n indicates the search pattern that locates the character ♪, which marks the end of the example sentence. \9 collects everything after the character ♪ that is not an end of line (\9 is most likely empty), and \n finds the end of the line.

The replacement pattern is as follows: \1\2\3╕ \4\5\6\7\8♪\9\5\6\2τ\8\n

This means that everything up to the first synonym is the same, but the ╕ gets moved to the position after the next synonym. Everything then continues as aforesaid until the original end of line is reached. It is here that the new example sentence is placed. The example sentence remains the same, except that the initial keyword is substituted for the next keyword in the list of synonyms that had the ╕ after it in the previous iteration. \n then creates the end of line. One should continue to apply this search and replacement pattern until it can no longer be matched. After it matches no more, each synonym in the list should be in its own example sentence. Then, the number of example sentences on each line is equal to the number of synonyms on each line plus one since the original example sentence is copied.

It is important to note that τ marks the keyword in each example sentence after the original example sentence. Also, for ease of application, the original example sentence is copied by the regexp. Thus, it appears twice on each line where the keyword in the copy is denoted by a τ, while the Σ denotes the same keyword in the original example sentence. Moreover, each example sentence is separated by a tab, followed by a colon, followed by a whitespace.

The first letter in any sentence should be capitalized. In order to assume this is the case, the following search pattern can be achieved by the matching pattern (\t:+)([a-z]), (as the example sentences in each row are preceded by a Tab character followed by a series of semicolons and a space) along with this replacement pattern: \1\U\2\E

Above, (\t:+) means that \1 will collect a tab followed by a colon that occurs one or more times. ([a-z]) means that \2 will collect the first letter of a sentence only if it is lowercase. If the first

letter of a sentence is not lowercase, then this search pattern will not match, so there is no risk of corrupting sentences that are already correct. It is important to note that this is functional only because each example sentence is separated by one or more tabs and one or more colons. As such, this is effectual for each example sentence on every line.

The replacement pattern \1\U\2\E means that the one or more tabs remain in place in front of each example sentence. If ([a-z]) finds a match, then \U\2 converts it to uppercase (hence, the capital U). \E ends the uppercase replacement mode.

At this point, the sentences have been created and rendered grammatically correct (excluding the special characters). It is now time to create the sentence pairs that are required as data for the BERT Transformer. A nested for-loop is used for this task. The algorithm is broken up into four steps.

Step #1. Search pattern: ♪, and the replacement pattern is: ♪♫. The point of step one is to use the character ♫ to separate the two sentences that will be paired.

Step #2. Search pattern is as follows:

(╕ \t:+)([^\n\t]+♪)([^\n]*)(♫)(\t)([^\n\t]+)(\t)([^\n]*\n)

The component parts will now be addressed.

(ㅋ \t:+) means that \1 will search for the ㅋ , which is in between the last synonym and the first example sentence, since all the synonyms have been used for sentence generation in the first regex algorithm. \1 collects the ㅋ , the whitespace, the one or more tabs followed by the one or more colons, followed by a whitespace.

([^\n\t]+♪) means that \2 will start after the colon in \1 and collect everything that is not an end of line, nor a tab, one or more times, then it will collect the character ♪. In other words, \2 will collect the first example sentence on the line in addition to the character ♪.

([^\n]*) means that \3 will collect everything in between the character ♪ in \2 and the character ♫ in \4, although it is very likely that \3 will be empty on most lines.

(♫) as \4 is self-explanatory. (\t), which is \5 is also self-explanatory and is the tab in between the first two example sentences.

([^\n\t]+) means that \6 will search for and collect anything after the tab that is not an end of line, nor a tab, in that order, one or more times. This, in practice, is the second example sentence on the line.

(\t) as \7 is the tab between the second and third example sentence.

([^\n]*\n) means that \8 collects everything else after the tab at the end of the second example sentence. This may include other example sentences.

The replacement pattern for step #2 is \1\2\3\5\6\4\7\8\2\6\n

The replacement pattern will take the first example sentence, a tab, then the second example sentence, then the character ♫, then the tab, then all other example sentences, then the first and second example sentence, then the end of the line. In other words, the pairing of the first and second example sentences is pushed to after the last example sentence and our sentence pairing marker, character ♫, is pushed in between the second and third example sentences. The purpose of this is that the second and third example sentences can be paired and pushed to the end in the second iteration of this nested for-loop.

Step #3: search for character ♫ and replace it with whitespace because it needs to be available to separate the second and third sentences in the next iteration.

For step #4, the search pattern is as follows: (ㅋ)(\t)(:+ [^\n\t]+)(♪)(\t)([^\n\t]+)(\t)
The replacement pattern is as follows: \2\3\1\5\6\4\7

(ㅋ) means that \1 searches for the ㅋ followed by a whitespace, which can be found where it remained after the previous synonym.

(\t) means that \2 searches for the tab before the first example sentence.

(:+ [^\n\t]+) means that \3 searches for and collects the one or more colons before the first example sentence, the whitespace, then everything that is not an end of line, nor a tab, one or more times.

\4 is the character ♪ that separates the first and second example sentences.

\5 is the tab before the second example sentence.

([^\n\t]+) means that \6 will take the second example sentence.

(\t) is \7, which takes the tab after the second example sentence.

The replacement pattern is the following: \2\3\1\5\6\4\7

This takes the tab before the first example sentence, then the first example sentence, then the ╕ to mark the beginning of the sentence pair for the next iteration, then the tab, then the second example sentence, then the ta ♪ to separate the second example sentence from the third example sentence. This creates a separation for the next pair of sentences in the next iteration, then the tab is inserted to separate the second example sentence from the third example sentence.

Repeat steps 1-4 until the search pattern finds no matches. This will create sentence pairs at the end of the example sentences that were created. Once this is completed, one can create a new text document with only the sentence pairs: one pair per line. Then, create the labels based on whether both sentences in each pair come from the same line in the original text document that was just created. If both sentences come from the same line, label that pair with a one. Otherwise, label that pair with a zero. Once this is completed, and all auxiliary characters have

been removed, the outcome is annotated training data for training data for fine-tuning the BERT Transformer for the task of paraphrase identification.

As the training data was generated from dictionary entries of exclusively adverbial expressions, the retrained network is expected to be able to identify adverbial paraphrases only, however, in this task, it is expected to perform well, since the training data contains a relatively large set of examples with a very thorough coverage of adverbial phrase usage.

Implementation of Training

We trained BERT in Google CoLab. Google CoLab is a free cloud service that allows one to write and execute Python code in your browser with free access to GPUs and popular libraries such as Keras, TensorFlow, PyTorch, OpenCV. It is great for machine learning, data analysis, and education. CoLab is a hosted Jupyter notebook service that requires no setup to use. The resources for CoLab are not guaranteed and limited, and the usage limits sometimes fluctuate, but the usage limits can be extended and more dependable with a ColabPro subscription.

The code can be found in the appendix, and it has been adopted from code on the GitHub site (https://github.com/naveenjafer/BERT_Amazon_Reviews). It uses the Pytorch package and Hugging Face's popular Transformer Library to provide a solution to fine-tuning the BERT base model applied for a one sentence binary classification problem. We changed the code so that we could use it for solving a two-sentence binary classification problem. We can consider our paraphrasing problem as such a classification exercise. It was run in the Google CoLab environment which allowed us to access GPU compute facilities remotely for free.

We split our training data into 80% used for training and the remaining 20% for validation. This gave us 72210 training samples and 18053 validation samples. We had a max length of 100 units, our Batch Size was 64, and we used 2 epochs to retrain. It is recommended that one uses 2 or 3 epochs for finetuning in Reference number 7. We used the Negative Log Likelihood Loss

Function and the Adam optimization. The training loop did 1129 iterations for each epoch and 10 validation iterations after each epoch. After the first epoch, the training accuracy was over 98% and the validation accuracy was over 96%. After the second and final epoch, the training accuracy was 100% and the validation accuracy was over 97%. The program ran for 22 minutes total.

Since this thesis only addressed sentence pairs with adverbial expressions, in the future, I would love to apply this methodology of generating annotated paraphrasing sentence pairs with other parts of speech, such as, adjectives, nouns, and verbal expressions. Approximately 2.5 million sentences can be generated for other parts of speech by following the methodology outlined above. This data can be used for thorough retraining. Some of the objectives that can be achieved with such data are alternate uses of words and expressions in natural (human) language. This dataset is expected to boost the performance of Transformer-based networks in NLU. In general, dictionary annotations and style can also be exploited; transfer learning methods will make identification more efficient and will increase performance in differentiating stylistic language (formal, colloquial, slang, etc.)

REFERENCES

1. “History of Natural Language Processing.” *Wikipedia*, Wikimedia Foundation, 3 June 2020, en.wikipedia.org/wiki/History_of_natural_language_processing.
2. “Natural Language Processing.” *Wikipedia*, Wikimedia Foundation, 14 July 2020, en.wikipedia.org/wiki/Natural_language_processing.
3. Foote, Keith D. “A Brief History of Natural Language Processing (NLP).” *DATAVERSITY*, 17 June 2019, www.dataversity.net/a-brief-history-of-natural-language-processing-nlp/.
4. “Machine Translation.” *Wikipedia*, Wikimedia Foundation, 14 July 2020, en.wikipedia.org/wiki/Machine_translation#:~:text=Machine%20translation%2C%20sometimes%20referred%20to,speech%20from%20one%20language%20to.
5. “History of Machine Translation.” *Wikipedia*, Wikimedia Foundation, 30 Jan. 2020, en.wikipedia.org/wiki/History_of_machine_translation.
6. Alammar, Jay. “The Illustrated Transformer.” *The Illustrated Transformer – Jay Alammar – Visualizing Machine Learning One Concept at a Time.*, 2018, jalammar.github.io/illustrated-transformer/.
7. Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova, BERT: “Pre-training of Deep Bidirectional Transformers for Language Understanding“, Proceedings of NAACL-HLT 2019, pages 4171–4186
8. Alammar, Jay. “The Illustrated BERT, ELMo, and Co. (How NLP Cracked Transfer Learning).” *The Illustrated BERT, ELMo, and Co. (How NLP Cracked Transfer Learning) – Jay Alammar – Visualizing Machine Learning One Concept at a Time.*, 2018, jalammar.github.io/illustrated-bert/.
9. Naveenjafer. “Naveenjafer/BERT_Amazon_Reviews.” *GitHub*, 2020, github.com/naveenjafer/BERT_Amazon_Reviews.

APPENDIX:

Python Code for Implementation

The code below was implemented in Google CoLab because it allowed us to use a high-power machine remotely for free. Here is the code:

```
! pip install pandas
! pip install torch
! pip install transformers

from google.colab import drive
drive.mount('/content/drive')
projectFolder = "./drive/My Drive/Bert/"

import pandas as pd
import torch
import torch.nn as nn
from transformers import  BertModel, BertTokenizer
from torch.utils.data import DataLoader
import torch.optim as optim
import os
from torch.utils.data import Dataset

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def read_and_shuffle(file):
    df = pd.read_csv(file, delimiter='\t')
    # Random shuffle.
    df.sample(frac=1)
    return df
```

```
def get_train_and_val_split(df, splitRatio=0.8):
    train=df.sample(frac=splitRatio,random_state=200)
    val=df.drop(train.index)
    print("Number of Training Samples: ", len(train))
    print("Number of Validation Samples: ", len(val))
    return(train, val)

def get_max_length(reviews):
    return len(max(reviews, key=len))

def get_accuracy(logits, labels):
    # get the index of the max value in the row.
    predictedClass = logits.max(dim = 1)[1]

    # get accuracy by averaging over entire batch.
    acc = (predictedClass == labels).float().mean()
    return acc

def trainFunc(net, loss_func, opti, train_loader, test_loader, config):
    best_acc = 0
    for ep in range(config["epochs"]):
        for it, (seq, attn_masks, labels) in enumerate(train_loader):
            opti.zero_grad()
            #seq, attn_masks, labels = seq.cuda(args.gpu),
attn_masks.cuda(args.gpu), labels.cuda(args.gpu)
            seq, attn_masks, labels = seq.to(device), attn_masks.to(device),
labels.to(device)

            logits = net(seq, attn_masks)
            loss = loss_func(m(logits), labels)

            loss.backward()
            opti.step()
```

```
            print("Iteration: ", it+1)


            if (it + 1) % config["printEvery"] == 0:
                acc = get_accuracy(m(logits), labels)
                if not os.path.exists(config["outputFolder"]):
                    os.makedirs(config["outputFolder"])


                # Since a single epoch could take well over hours, we
regularly save the model even during evaluation of training accuracy.
                torch.save(net.state_dict(), os.path.join(projectFolder,
config["outputFolder"], config["outputFileName"]))
                print("Iteration {} of epoch {} complete. Loss : {} Accuracy
: {}".format(it+1, ep+1, loss.item(), acc))
                print("Saving at", os.path.join(projectFolder,
config["outputFolder"], config["outputFileName"]))


        # perform validation at the end of an epoch.
        val_acc, val_loss = evaluate(net, loss_func, val_loader, config)
        print(" Validation Accuracy : {}, Validation Loss :
{}".format(val_acc, val_loss))
        if val_acc > best_acc:
            print("Best validation accuracy improved from {} to {}, saving
model...".format(best_acc, val_acc))
            best_acc = val_acc
            torch.save(net.state_dict(), os.path.join(projectFolder,
config["outputFolder"], config["outputFileName"] + "_valTested_" +
str(best_acc)))


def evaluate(net, loss_func, dataloaderq, config):
    net.eval()


    mean_acc, mean_loss = 0, 0
    count = 0


    with torch.no_grad():
```

```
        for seq, attn_masks, labels in dataloaderq:
            #seq, attn_masks, labels = seq.cuda(args.gpu),
attn_masks.cuda(args.gpu), labels.cuda(args.gpu)
            seq, attn_masks, labels = seq.to(device), attn_masks.to(device),
labels.to(device)

            logits = net(seq, attn_masks)
            mean_loss += loss_func(m(logits), labels)
            mean_acc += get_accuracy(m(logits), labels)
            print("Validation iteration", count+1)
            count += 1

            '''
            The entire validation set was around 0.1 million entries,
            the validationFraction param controls what fraction of the
shuffled
            validation set you want to validate the results on.
            '''
            if count > config["validationFraction"] * len(val_set):
                break
    return mean_acc / count, mean_loss / count


config = {
    "splitRatio" : 0.8,
    "maxLength" : 100,
    "printEvery" : 100,
    "outputFolder" : "Models",
    "outputFileName" : "ParaphraseClassifier.dat",
    "threads" : 4,
    "batchSize" : 64,
    "validationFraction" : 0.0005,
    "epochs" : 2,
    "forceCPU" : False
```

```
    }
if config["forceCPU"]:
    device = torch.device("cpu")

config["device"] = device

class SentimentClassifier(nn.Module):
    def __init__(self, num_classes, device, freeze_bert = True):
        super(SentimentClassifier, self).__init__()
        self.bert_layer = BertModel.from_pretrained('bert-base-uncased')
        self.device = device

        if freeze_bert:
            for p in self.bert_layer.parameters():
                p.requires_grad = False

        self.cls_layer = nn.Linear(768, num_classes)

    def forward(self, seq, attn_masks):
        '''
        Inputs:
            -seq : Tensor of shape [B, T] containing token ids of sequences
            -attn_masks : Tensor of shape [B, T] containing attention masks
to be used to avoid contibution of PAD tokens
        '''

        #Feeding the input to BERT model to obtain contextualized
representations
        cont_reps, _ = self.bert_layer(seq, attention_mask = attn_masks)

        #Obtaining the representation of [CLS] head
        cls_rep = cont_reps[:, 0]
```

```
        #Feeding cls_rep to the classifier layer
        logits = self.cls_layer(cls_rep)

        return logits.to(self.device)

from torch.utils.data import Dataset
from transformers import BertTokenizer
import torch

class AmazonReviewsDataset(Dataset):
    def __init__(self, df, maxlen):
        self.df = df
        # A reset reindexes from 1 to len(df), the shuffled df frames are
sparse.
        self.df.reset_index(drop=True, inplace=True)
        self.tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
        self.maxlen = maxlen

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

    def __getitem__(self, index):
        sent1 = self.df.loc[index, 'Sentence1']
        sent2 = self.df.loc[index, 'Sentence2']

        # Classes start from 0.
            label = int(self.df.loc[index, 'Score'])

        # Use BERT tokenizer since it needs to be able to match the tokens to
the pre trained words.
        token1 = self.tokenizer.tokenize(sent1)
        token2 = self.tokenizer.tokenize(sent2)
```

```
        # BERT inputs typically start with a '[CLS]' tag and end with a
'[SEP]' tag. For

        tokens = ['[CLS]'] + token1 + ['[SEP]'] + token2 + ['[SEP]']


        if len(tokens) < self.maxlen:

            # Add the ['PAD'] token

            tokens = tokens + ['[PAD]' for item in range(self.maxlen-
len(tokens))]

        else:

            # Truncate the tokens at maxLen - 1 and add a '[SEP]' tag.

            tokens = tokens[:self.maxlen-1] + ['[SEP]']


        # BERT tokenizer converts the string tokens to their respective IDs.

        token_ids = self.tokenizer.convert_tokens_to_ids(tokens)


        # Converting to pytorch tensors.

        tokens_ids_tensor = torch.tensor(token_ids)


        # Masks place a 1 if token != PAD else a 0.

        attn_mask = (tokens_ids_tensor != 0).long()


        return tokens_ids_tensor, attn_mask, label


print('Loading BERT tokenizer...')

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
do_lower_case=True)


print("Configuration is: ", config)

# Read and shuffle input data.

df = read_and_shuffle(os.path.join(projectFolder, "PARAPHRASE-
DATASET/datap.csv"))


df
```

```
num_classes = df['Score'].nunique()

print("Number of Target Output Classes:", num_classes)

totalDatasetSize = len(df)


# Group by the column Score. This helps you get distribution of the Review
Scores.

symbols = df.groupby('Score')


scores_dist = []

for i in range(num_classes):

    scores_dist.append(len(symbols.groups[i])/totalDatasetSize)


train, val = get_train_and_val_split(df, config["splitRatio"])


val.to_csv(os.path.join(projectFolder, "PARAPHRASE-DATASET/Validations.csv"))

train.to_csv(os.path.join(projectFolder, "PARAPHRASE-DATASET/Train.csv"))


# You can set the length to the true max length from the dataset, I have
reduced it for the sake of memory and quicker training.

#T = get_max_length(reviews)

T = config["maxLength"]


train_set = AmazonReviewsDataset(train, T)

val_set = AmazonReviewsDataset(val, T)


train_loader = DataLoader(train_set, batch_size = config["batchSize"],
num_workers = config["threads"])

val_loader = DataLoader(val_set, batch_size = config["batchSize"],
num_workers = config["threads"])


# We are unfreezing the BERT layers so as to be able to fine tune and save a
new BERT model that is specific to the Sizeable food reviews dataset.


net = SentimentClassifier(num_classes, config["device"], freeze_bert=False)
```

```
net.to(config["device"])
weights = torch.tensor(scores_dist).to(config["device"])

# Setting the Loss function and Optimizer.
loss_func = nn.NLLLoss(weight=weights)
opti = optim.Adam(net.parameters(), lr = 2e-5)
m = nn.LogSoftmax(dim=1)

torch.cuda.set_device(0)
trainFunc(net, loss_func, opti, train_loader, val_loader, config)
```