Università degli Studi di Salerno

Dipartimento di Informatica

Tesi di Laurea Magistrale in

Informatica

Summarization of multilingual documents based on Deep Learning

Relatore
Prof. Vincenzo Deufemia

Candidato
Salimzhanov Abylay
Matr. 0522500618

Anno Accademico 2018-2020

Content

Abstract.

NLP has proved to be useful for information to be used in a more intelligently way due to the accessibility of data and the selection of tools created by the different communities of researchers. For natural language processing, there are sets of obstacles for building a model every time we want to run an experiment. Automatic text summarization, the computer-based production of condensed versions of documents, is an important technology for the information society. Without summaries it would be practically impossible for human beings to get access to the ever-growing mass of information available online.

We consider an open issue in the Natural Language Processing area for a set of different problems considered and to be solved for specific tasks. For our purpose of summarization, we offered not only one solution, but the set of interrelated approaches, to have results based on of Rouge score which showed value greater than most methods proposed in scientific papers of previous years. Regarding to Rouge-1, we achieved a 0.51 score without taking into account the reordering of sentences in the final summary. In autonomous prototype was considered several steps usage of methods on the regard of problems which are the most critical in the way of getting result. Usage of Bert extractive summarization approach allowed applying semantic analysis to pre-summary, which gives a pretty well defined and comprehensive summary. Applying Universal Sentence Encoder, we obtained a summary, which resolves most of the issues of summarization task.

Introduction.

Humans can consolidate textual information from multiple sources and organize the content into a coherent summary. Automatic text summarization is an important tool for enhancing users' ability to make decisions in the face of overwhelming amounts of data. The key to making this technology useful and practical is to have high-performing systems that work on a variety of texts and settings.

Automatic text summarization is a complex research area in which a considerable amount of work has been done by researchers. Summarization requires both understanding and generation. Just as machine translation led to many advances in Natural Language Processing (e.g., alignment algorithms, attention), summarization can lead to new advances in NLP overall. The major difference is that it must care about analyzing content in summarization. A good summary must be easy to read and give a good overview of the content of the source text.

Since summaries tend to be more and more oriented towards specific needs, it is necessary to tune existing evaluation methods accordingly. Unfortunately, these needs do not give a clear basis for evaluation, and the definition of what is a good summary remains to a large extent an open question. Therefore, the evaluation of human or automatic summaries is known to be a difficult task. It is difficult for humans, which means the automation of the task is even more challenging and hard to assess. However, because of the importance of the research effort in automatic summarization, a series of proposals have been made to partially or fully automate the evaluation. It is also useful to note that in most cases automatic evaluations already correlate positively with human evaluations.

The goal of this thesis is to investigate the problem of implementing multi-document and multilingual summarization systems. The most important obstacles facing multi-document summarization include excessive redundancy in source content, ambiguous sentence construction, and the shortage of training data. Taking in advance the state of the art of the research area we pay attention to an extractive approach to solve an NLP task.

The proposed solution consists of three main steps: the document body is first run through the BertSum Extractive summarizer model. The purpose of using this approach is its productivity and advanced features that are described in the work of Yang Liu and Mirella Lapata "Text Summarization with Pretrained Encoders". At the first stage, the main goal was to reduce the size of the original document body and at the same time maintain the context and essence of the document. A detailed description and structure of the technology is described in the second chapter of this work. The next step of the autonomous summarizer is to best understand the semantics of the input document in turn, and thereby use the embedding approach using the latest technology from Google Universal Sentence Encoder, the reason for the choice, architecture and subtleties of this approach, we described in the second chapter of the dissertation. Also at this stage we use the traditional Embeddings Ranking method, but in this case not previously used at the proposal level, which expands the possibilities and gives a significant result based on the data obtained.

We described the Embeddings Ranking ++ method in Chapter 2, where we visually provide an opportunity to see the difference between the k-means method and the traditional Embeddings Ranking method at the word level, thus we achieve a solution to the problem of repeatability or similarity of sentences in the context of the

final summary. The final point in the developed prototype is the Sentence Reordering method, which makes it possible to achieve the correct order of submitting offers, and thus we get logically connected and consistent sentences in the summary.

The structure of the thesis consists of three parts, in the first part we describe the evolution of the development of technology and our considerations on their account, as well as describes the methods for which we have considered this or that approach and are actually practical and referring to scientific papers, we came to this solution provided by us. The second chapter directly describes the steps of the approach, problems, causes, and solutions. In each section, we divided a separate technology and gave it a detailed description, accompanied by a visual representation, as well as structured it in a more meaningful context and collected all the steps in the developed prototype. In the third part, we describe the results and give them an assessment, comparing with previous works. And also describe the future desire to complement the functionality of our work.

Chapter 1. Considerations.

Summarization.

The technique of automatically changing the authentic textual content of an electronic record right into a shorter textual content (precis) containing the main ideas of the original textual content is referred to as the summarization of the text. Summarization of the text can be divided into the following types: extractive, if the abstract of the record is composed of sentences of the original record; and abstractive, if the sentences of the unique file are rephrased to compose a summary. Due to the requirement to rephrase the textual content, the challenge of the summary is considered more difficult than extractive.

There are two forms entry to the text summarization machine: numerous documents and a single record. In the case of several input documents, their subjects may additionally coincide with every other, as within the case of news documents of the same subject, or their topics may additionally fluctuate from every other (on this case, the summarization system should highlight the most vital statistics about the topic of each enter file). In the case of a single enter document, essential records may additionally appear within the file most effective once.

The problem here is that often there may be no additional records approximately which information is more essential within the document. Therefore, the summation of a record is taken into consideration a more tough mission.

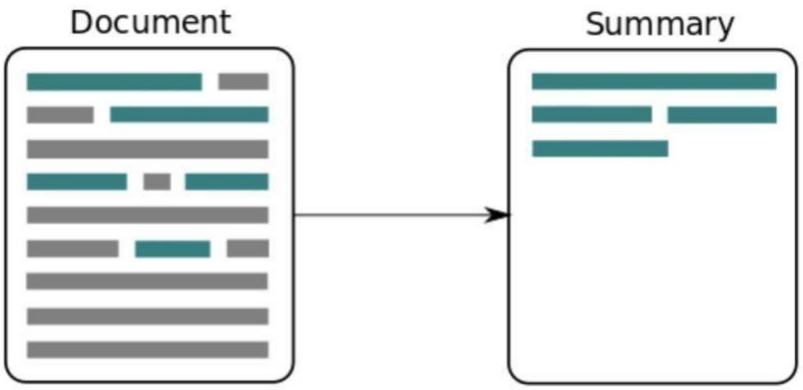


Figure - 1. Text summarization visualization

Extractive textual content summarization entails the selection of terms and sentences from the source report to make up the new precis. Techniques involve ranking the relevance of terms so one can choose those most relevant to the means of the supply.

Abstractive text summarization involves generating new terms and sentences to capture the means of the source report. This is an extra challenging approach but is also the method at the end used by humans. Classical methods function by using selecting and compressing content material from the source record.

Most summarization these days is primarily based on a sentence-extraction paradigm where a list of functions speculate to signify the relevance of a sentence in a report or set of documents is used as a text interpretation mechanism. The tactics to sentence selection can be driven by statistical facts or based on a prominent summarization theory, which takes into consideration linguistic and semantic information.

Basic statistical tactics to content choice have relied on using frequency computation to identify applicable document key phrases. The primary mechanism used is to measure the frequency of every phrase in a document and to adjust the frequency of words with additional corpus evidence which includes the inverse document frequency of the word in a modern file collection. This helps boost the rating of phrases that are not too common in a modern corpus and moderate the score of otherwise too common contemporary phrases. These methods expect that the relevance of a given idea in a text is proportional to the number of instances, the concept is "mentioned" inside the report, assuming that every distinct phrase corresponds to a different idea.

However, counting concept occurrences in the textual content is not a trivial project given the presence of synonymy ("dog" and "puppy" could seek advice from the concept dog) and coreferential expressions ("Obama" and "the President" could seek advice from the identical individual) which contributes to textual content cohesion. Once keywords are identified in the report, sentences containing those key phrases may be selected using various sentence scoring and ranking mechanisms (the relevance of a sentence may be proportional to the range of key phrases it contains). More sophisticated strategies than the usage of simple word counts also exist, for example subject matter signatures have been proposed as a way to version applicable keywords mainly domain names and as interpretation and sentence relevance determination mechanism.[1]

The function of sentences in text is likewise deemed indicative of sentence relevance.

For instance in information stories, the primary or leading paragraph typically contains the main information approximately the event reported inside the information, even as the rest of the textual content gives details as well as background information approximately the event, therefore choosing sentences from the start of the text could be a reasonable strategy.

However, in scientific texts the introduction typically offers background information, at the same time as the main trends are reported within the conclusions, so the positioning strategy wishes to be tailored to the textual content kind to be summarized. Another superficial, easy-to-implement technique consists of measuring the relevance of sentences through evaluating them with the identity of the file to be summarized, a sentence containing title words might be regarded as applicable and the extra identify phrases a sentence has the more applicable the sentence would be.

Two essential questions in text summarization are; i) a way to pick out the critical content of a file, and ii) a way to express the selected content material in a condensed manner. Text summarization research has normally concentrated greater on the product: the summary, and much less at the cognitive basis of textual content understanding and manufacturing that underlie human summarization. Some of the limitations of present-day structures would benefit from a higher knowledge of the cognitive basis of the project. However, formalizing the content material of open domain documents continues to be a research issue, so most systems are most effective primarily based on a selection of sentences from the set of original files. Where the transformation of the unique

textual content material into a summary is concerned, there are two main kinds of summaries: an extractive summary, a fixed of sentences from the enter file and an abstractive precis (an abstract), a precis in which a number of its fabric isn't always present within the input document. Summaries can also be labeled into indicative or informative depending on their intended reason to alert or to inform respectively.

Early research in summarization targeting summarization of single documents (single document summarization), however inside the present-day web context, many processes focus on the summarization of more than one, associated documents (multi-report summarization). Most summarization algorithms these days aim at the technology of extracts given the difficulties associated with the automatic generation of nicely fashioned texts in arbitrary domain names.

It is generally generic that there is a number of things that decide the content to choose from the supply file and the kind of output to produce, as an instance element inclusive of the audience (professional vs non-expert reader) truly influences the fabric to pick out. This quick advent overviews some conventional works on content selection and realization, summarization gear and resources, and summarization evaluation. For the fascinated reader, there are several papers and books that provide enormous overviews of text summarization structures and strategies.[2]

A multi-record summary is a quick illustration of the crucial contents of a hard and fast of related files. The relation between the documents may be of various types, an example document may be associated due to the fact they're

approximately the same entity, or due to the fact they discuss the equal topic, or due to the fact they may be approximately the same occasion or the same event kind. Fundamental issues when dealing with multi-source enter in summarization are the detection and discount of redundancy as well as the identification of contradictory facts.[3]

The multi-document summarization trouble is studied within the context of multi-source data extraction in particular domain names. Here templates instantiated from various documents are merged with the usage of precise operators aiming at detecting equal or contradictory records. In a statistics retrieval context, in which multi-report summaries are required for a hard and fast of documents retrieved from a seek engine in reaction to the query, the Maximal Marginal Relevance (MMR) technique may be applied. The approach ratings text passages (e.g., paragraphs) iteratively contemplating the relevance of each passage to a user query and the redundancy of the passage with admiration to summary content already selected. In the case of normal summarization, computing similarity among sentences and the centroid of the files to summarize has resulted in aggressive summarization solutions.

Sentence ordering is also a problem for multi-report summarization. In single report summarization it's assumed that providing the records within the order this information appears inside the input record would normally produce a suitable summary. By contrast, in multi-document summarization precise attention needs to be paid to how sentences extracted from a couple of assets are going to be presented. Various strategies exist for dealing with sentence ordering,

for instance if sentences are time-stamped via guide date, then they could be presented in chronological order. However, this isn't constantly viable due to the fact recognizing the date of a reported occasion is not trivial and not all documents contain an ebook date.[4]

Sentence order also can be conceived to symbolize the different subjects to be addressed within the summary. For instance, a clustering algorithm may be used to identify topics within the set of input files and find out in what order the subjects are supplied within the input files, this in turn might be used to give sentences in order much like that observed within the input set. A probabilistic technique to sentence ordering seeks to estimate the probability of a series of sentences. It tries to discover a domestically most beneficial order by studying ordering constraints for pairs of sentences. An entity-grid version tries to symbolize coherent texts by way of modeling entity roles (e.g., subject, object) in consecutive sentences, the model can discriminate coherent and incoherent texts.

In the past determined its usage a hard and fast of different techniques and strategies, some of them will be met nowadays or in a few ways changed, and complexed to each other:

Positional method: the concept of which is to take the first and last sentence of a paragraph are topic sentences (85% vs 7%).

Luhn's method: concept of which is the frequency of content terms (words appearing most and least). There were taken in advance stemming and stop words removal, and then select sentences the highest concentrations of salient content terms. Luhn proposed that the significance of each word in a document signifies how important it is. The idea is that any sentence with maximum occurrences of the highest frequency words (stop words) and least occurrences are not more important to the meaning of the document than others. Although it is not considered a very accurate approach. It selects only the words of higher importance as per their frequency. Higher weights are assigned to the words present at the beginning of the document.

Edmundson's method: concept of which are the main four features, (P) is the position, (F) is word frequency, (C) is the cue words, (D) is the document structure. And in the end, taking linear combinations of 4 features.

FRUMP: concept of which is finding specific parts of the article, summarizer tends to make logical through whole chronology. Edmundson proposed the use of a subjectively weighted combination of features as opposed to traditionally used feature weights generated using a corpus. He considered the already known features (used in Luhn's method) but added a few other features.

Latent Semantic Analysis is an algebraic-statistical method that extracts hidden semantic structures of words and sentences. It is an unsupervised approach that does not need any training or external knowledge. LSA uses the context of

the input document and extracts information such as which words are used together, and which common words are seen in different sentences. A high number of common words among sentences indicates that the sentences are semantically related. The meaning of a sentence is decided using the words it contains, and meanings of words are decided using the sentences that contain the words. Singular Value Decomposition, an algebraic method, is used to find out the interrelations between sentences and words. Besides having the capability of modeling relationships among words and sentences, SVD has the capability of noise reduction, which helps to improve accuracy. To see how LSA can represent the meanings of words and sentences the following example is given.

Example: Three sentences are given as an input to LSA.

d0: ‘The man walked the dog’.

d1: ‘The man took the dog to the park’.

d2: ‘The dog went to the park’.

We can see that d1 is more related to d2 than d0; and the word ‘walked’ is related to the word ‘man’ but not so much related to the word ‘park’. These kinds of analysis can be made by using LSA and input data, without any external

knowledge. The summarization algorithms that are based on the LSA method usually contain three main steps.

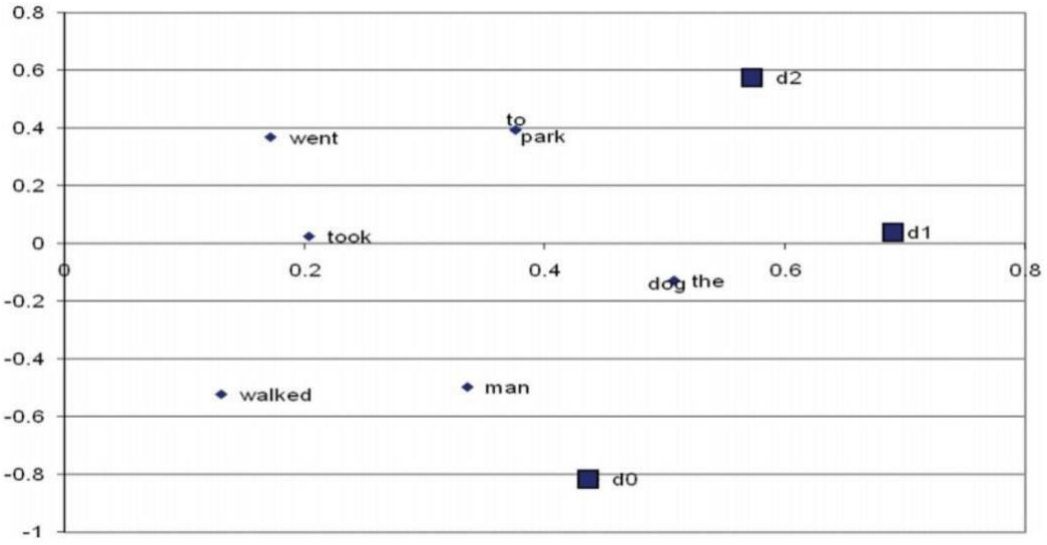


Figure-2. LSA represents the meaning of words and sentences.

LSA has several limitations. The first one is that it does not now use the records about phrase order, syntactic relations, and morphology. This type of data can be necessary for finding out the meanings of words and texts. The 2nd predicament is that it makes use of no world knowledge, however simply the facts that exist within the input document.

The 1/3 difficulty is related to the performance of the algorithm. With large and greater inhomogeneous information, the overall performance decreases sharply. The decrease in overall performance is resulting from SVD (Singular Value Decomposition), which is a very complicated algorithm.

After the first step of pre-processing comes to the step of finding out the principal topics. For this step, a concept × idea matrix is created by using finding

out the ideas that have common sentences. The not unusual sentences are the ones that have mobile values aside from 0 in each idea which is considered. Then the new cellular values of the concept × idea matrix are set to the entire commonplace sentence scores.

The idea × concept matrix-based totally on the V_T is given. After the introduction of the idea × concept matrix, the power of every idea is calculated. For every idea, the energy value is computed through getting the cumulative cellular values for every row of the concept × idea matrix.

The concept with the highest power cost is chosen as the principal topic of the entire document. A higher energy price indicates that the idea is much extra associated with the alternative standards, and its miles one of the main subjects of the entered text.

TF-IDF is a statistical measure used to evaluate the significance of a phrase or sentence in any context-based totally on its occurrence within the document, the higher its TF-IDF weight. To aim at the task, we first choose the significant words into the document. For simplicity and a fair contrast with the original method, we choose the one's phrases having tf * idf weight extra than a subject threshold. There are several approaches to calculate TF-IDF.

A non-trivial ranking is to recall ranking paragraphs as documents in a query retrieval problem, wherein the question is the name of the article. Let's compute TF-IDF for the query, with recognize to the files, and then simply to sum for each word in the query.

There are considered the count of the phrase within the document, the general quantity of files, and the total number of documents containing the phrase, respectively. Model training is composed of counting the weight of each unique word in every document. One of the beneficial homes of TF-IDF vectors: the cosine distance between vectors characterizes the similarity of the article, which may be used for clustering.

Text Analysis.

Normalization.

In literature sources, it has been again and again pointed out that during natural language there are opposite operating tendencies – the tendency to shop attempt and the tendency to provide redundancy. The effect of these two tendencies is directly associated with the intentions of fundamental speech act participants – the speaker and the listener. From the speaker's factor of view, it is simply better to choose a more economical manner for expressing thoughts, i.e. The speaker saves "generative" funds. The listener saves "perceptual" cognitive resources, i.e. He/she is interested in the handiest viable transitions from text to thought (which means), which contributes to the greatest degree of redundancy. Nowadays there is, though, an increasing tendency in the direction of the principle of least attempt, i.e. Language financial system about society's self-destructive dependency on faster living. [5]

Furthermore, in the Web where user-generated content has already hit its big mass, the need for realistic computation and records aggregation is increasing exponentially, as proved via headhunting within the industry for

"huge information experts" and the advancements in a new "Data Science" discipline. The democratization of online content advent has caused the increase of Web debris, which is unavoidably and negatively affecting information retrieval and extraction.

Natural language text consists of many redundant factors, without which means it does not now change / whose presence or absence does not affect on the meaning. More frequently than now not, they act as noise seeing that they occur

uniformly throughout the body of the tongue. Similar factors are nearly conjunctions, prepositions, components of phrases that are responsible for the form always appear. Besides, there are distinct ways to jot down equal objects, for example, numerals can be written in numbers.

Here, textual content mining comes in useful. Text mining is the method of extracting first-rate facts from the textual content on apps and all through the Web. It is a constituent detail of the larger umbrella of superior analytics. In a nutshell, textual content mining works with the aid of uploading textual facts from a whole lot of sources. Then, herbal language processing (NLP) is used to pull insight from the text.

Text mining is used in business to gauge the sentiment of customers or summarize survey results. It's used in politics to measure preference for certain candidates. Mining is even used by intelligence organizations to identify areas of cyber-crime. Text mining is complicated; however, you can now apprehend why it has grown to be such an important part of how we examine textual content today.

There are many additives of mining and studying text information, however, it all begins with records retrieval. One can't analyze the text without retrieving it within the first place; hence, records retrieval is the vital preliminary step to textual content mining. After the text is retrieved, it's time to begin structuring or normalizing it. When fixing any hassle in NLP text normalize, that is, bring about a general, more informative form, without "noise". Normalization involves several steps. The normalization method allows you to get rid of grammatical statistics from the supply text (cases, numbers, verb bureaucracy and

tenses, participles, gender, etc.), maintaining just the semantic component. Text normalization is an automated manner of the synthesis processor, which converts the written form (orthographic form) of the textual content into the spoken form.

Text normalization has a long record in speech technology, dating back to the earliest paintings on complete TTS synthesis. Various processes were taken in the direction of textual content normalization. Normalization is usually the first segment of textual content preprocessing in a textual content-to-speech system. It is claimed that the process of text normalization is very much like that of machine translation from one herbal language to another.

Normalization may additionally be efficiently treated by the equal mechanisms and algorithms as those utilized in system translation with the exception that the target language here is “the spoken shape” of the source language.

When working with herbal facts, it is discretized. A similar manner in NLP is referred to as tokenization, it is composed in dividing the text into parts - tokens, usually a token is one phrase. Unfortunately, dividing the textual content by using spaces isn't completely correct, for the reason that there are many exceptions. If we remember continual expressions as numerous tokens, then the meaning of the textual content might be distorted, which may also have an effect on the result and the excellence of the answer to the trouble. For tokenization it's miles convenient to use normal expressions.

It is important that the tokenization technique ought to distinguish among types of numerical expressions. Numerical expressions together with dates, time, smartphone numbers, ZIP codes, and car license numbers ought to be diagnosed inside the textual content and dealt with uniquely for the reason that their pronunciation very much relies upon what they represent.

An easy department into phrases when processing textual content for TFIDF and Punkt tokenizer for dividing articles utilizing sentences with summarization. The latter makes use of the corps for teaching without a teacher, “learning” the sequences with which sentences start, which allows them to work with non-literary facts, for example, messages in social services. Networks in which sentences often begin with a lowercase letter.

The subsequent normalization step is the elimination of stop words. Stop words similarly distributed all through the language corpus. Stop words are words that do not carry any unbiased semantic load. To reduce the database, systems do not forget to prevent words whilst indexing or changing them with a special marker. These consist of conjunctions and allied words, pronouns, prepositions, particles, interjections, indicative phrases, numbers, punctuation marks, introductory words, some of a few nouns, verbs, adverbs (for example, continually, however, etc.). Due to the constant development and improvement of existing seek algorithms, their classification, clustering, etc., forestall-phrase databases are respectively being updated and changed.

For the unambiguous identification of a word, it's far delivered to its initial shape. This procedure is referred to as lemmatization, this is the transformation of

a word into an initial or, in other words, dictionary shape which is called a lemma. This technique is utilized in algorithms search engines like google whilst indexing net pages.

The system makes it possible to keep page statistics with a fixed number of words inside the index for convenient schematization of files. This permits you to speed up indexing and generate a clearer response to the quest query, because the seek engine analyzes the shortened form of the word faster.

As known, a lemma is the original, basic form of a phrase. For nouns and adjectives, it is the singular shape of the nominative case. For verbs, the lemma is an infinitive, an indefinite shape of the phrase that answers the question inside the infinitive.

For lemmatization, however, it isn't sufficient to use the simplest dictionary, due to the fact there are a huge number of neologisms that obey the same morphological regulations in the formation of paperwork. Good lemmatizers conduct complete morphological parsing, wherein phrases are divided into morphemes: themes (the most significant parts) and affixes (give additional meaning to the word). Besides lemmatization, one extra useful device used in complete-textual content search engines like google and yahoo is stemming - an easy model of morphological analysis that makes use of positive guidelines to extract the idea of a phrase.

Stemming is typically known as an approximate heuristic process throughout which endings are discarded from phrases inside the expectation that in maximum cases this could justify itself. Stemming is based on the regulations

of the morphology of the language and does not require the garage of the dictionary of all phrases. Thus, stemming cuts off finishing phrases and suffixes from the phrase so that the relaxation is the same for all grammatical types of the phrase.

A more complex approach to fixing the hassle of determining the idea of a phrase is lemmatization. The intention of stemming and lemmatization is one - to bring phrase paperwork and derivative types of a word to a commonplace basic form.

The difference between stemming and lemmatization is that lemmatization takes into account the context and converts the word to its meaningful basic form, whilst stemming certainly removes the previous couple of characters, which frequently ends in incorrect meaning and spelling errors.

For example, lemmatization would efficiently outline the preliminary shape of “caring”, whilst stemming would reduce off the -ing finishing and convert it to “car”.

“Caring” -> Lemmatization -> “Care”

“Caring” -> Stemming -> “Car”

Also, once in a while the identical word can have numerous extraordinary lemmas. Based on the context one is using, one has to define the “part-of-speech” (POS) tag for the word in that specific context and extract the corresponding lemma. Another part of breaking down textual content records is the processes concerned in building syntactic structures in the course of language

comprehension commonly referred to as parsing or syntactic processing. Higher stage NLP also includes coreference resolution, that is tasked with finding all expressions in a hard and fast textual content that refers to the identical character or thing.

When processing data before vectorization, the following operations are sequentially performed:

1) bringing the text to lowercase, removing accents and performing Unicode normalization (Café -> cafe)
2) omitting punctuation characters;
3) deleting stop words;
4) lemmatizing each word.

Embeddings

Vectorization is a method in NLP to map phrases or phrases from vocabulary to a corresponding vector of real numbers that is used to find word

predictions, phrase similarities/semantics. Count the commonplace phrases or Euclidean distance is the popular approach used to suit similar documents which can be based totally on counting the number of not unusual phrases among the documents.

Cosine Distance technique will not work even supposing the variety of common phrases increases but the document talks about different topics. To triumph over this flaw, the “Cosine Similarity” method is used to discover the similarity between the documents.

A very fundamental definition of a word embedding is an actual number, vector representation of a phrase. Typically, these days, words with comparable meaning may have vector representations that are close together within the embedding space (even though this hasn’t continually been the case). When building a word embedding space, normally the aim is to seize a few kinds of courting in that space, be it meaning, morphology, context, or some other type of dating. By encoding phrase embeddings in a densely populated space, we can represent phrases numerically in a way that captures them in vectors that have tens or masses of dimensions as opposed to millions (like one-hot encoded vectors). The splendor is that one-of-a-kind word embeddings are created both in special ways and using specific textual content corpora to map this distributional courting, so the results are word embeddings that help us on unique tasks inside the global of NLP.[6]

Words aren’t things that computer systems clearly understand. By encoding them in a numeric form, we will observe mathematical guidelines and do matrix

operations to them. This makes them super within the world of machine studying, especially.

Take deep learning for example. By encoding words in a numerical form, we are able to take many deep learning architectures and follow them to phrases. Convolutional neural networks have been carried out to NLP tasks the use of word embeddings and have set the contemporary overall performance for plenty tasks.

Even better, what we have located is that we will truly pre-educate word embeddings which apply to many tasks. That’s the point of interest of the various kinds we will cope within this article. So, one doesn't research a brandnew set of embeddings in keeping with tasks, in keeping with corpora. Instead, we will analyze general representation that can then be used throughout tasks. One of the maximum fundamental approaches we can numerically represent words is through the one-hot encoding method (also sometimes known as to be counted vectorizing). The concept is extraordinarily simple. Create a vector that has as many dimensions as your corpora has particular words. Each particular word has a unique size and might be represented by using a 1 in that dimension with 0s anywhere else.

TF-IDF vectors are related to one-hot encoded vectors. However, instead of just featuring a count, they feature numerical representations where words aren't just there or not there. Instead, words are represented by their term frequency multiplied by their inverse document frequency.

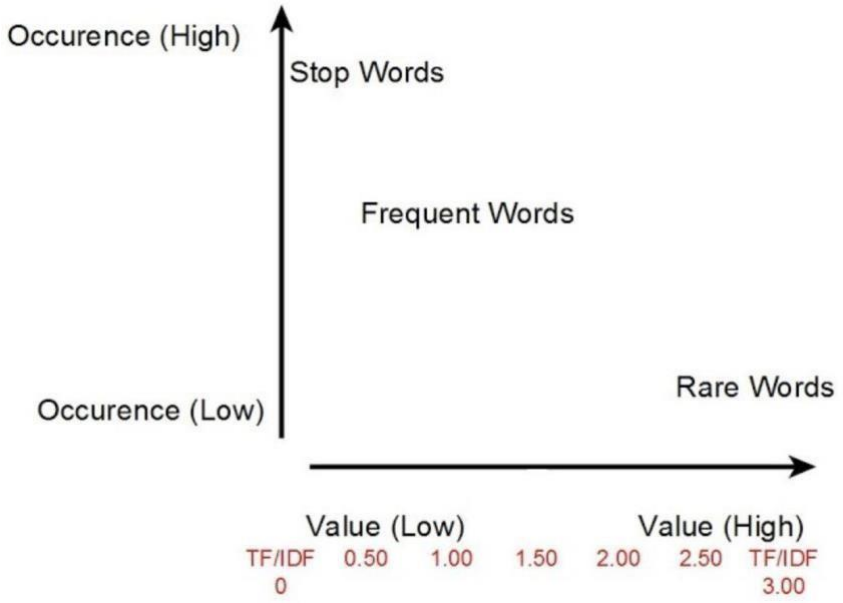


Figure-3. TF-IDF vectorization.

In simpler terms, phrases that occur a lot but anywhere ought to be given little or no weighting or significance. We can consider this as phrases like the or and within the English language. They don't provide a massive amount of value. However, if a word seems little or no or appears frequently, but simplest in one or two places, then these are possibly more vital phrases and must be weighted as such.

A co-occurrence matrix is exactly what it sounds like: a giant matrix that is as long and as wide as the vocabulary size.

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

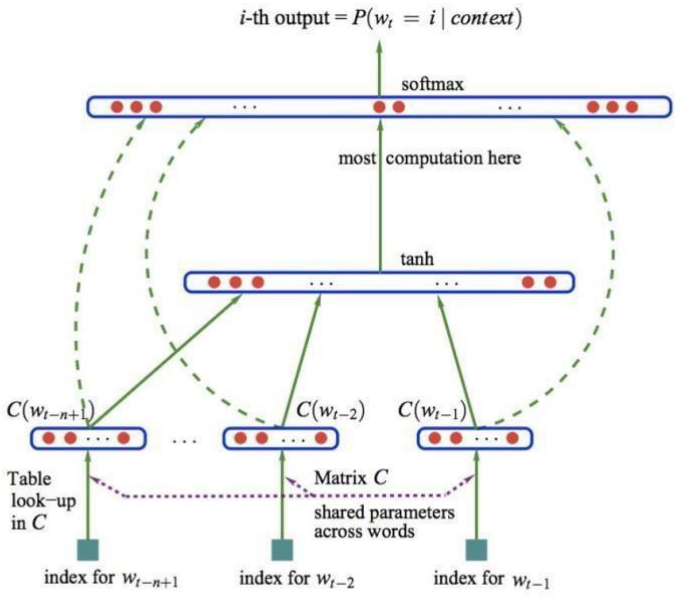
Figure-4. Co-occurrence matrix.

If words occur together, they are marked with a positive entry. Otherwise, they have a 0. It boils down to a numeric representation that simply asks the question of "Do words occur together? If yes, then count this."

A neural probabilistic model learns an embedding by achieving some challenges like modeling or type and is what the rest of these embeddings are greater or less primarily based on.

Typically, we clean textual content and create one-warm encoded vectors. Then, we define our representation size (300 dimensional might be good). From there we initialize the embedding to random values. It's the entry point into the network, and back-propagation is applied to regulate the embedding primarily

based on whatever goal assignment we have. It takes plenty of data and can be very slow. The trade-off right here is that it learns an embedding that is right for the textual content records that the community becomes educated on in addition to the NLP venture that became jointly learned all through training.



Neural architecture: $f(i, w_{t-1}, \cdots, w_{t-n+1}) = g(i, C(w_{t-1}), \cdots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i-th word feature vector.

Figure-5.

Word2Vec is a higher successor to the neural probabilistic model. We nevertheless use a statistical computation approach to analyze from a textual content corpus; however, its method of education is greater efficient than just easy embedding schooling.

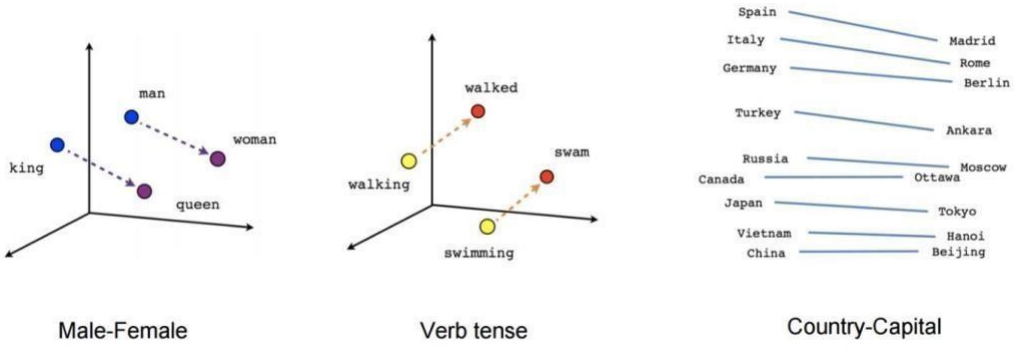


Figure-6. Word to vector model's representations.

It is more or less the standard method for training embeddings the days. It is the first method that demonstrated classic vector arithmetic to create such analogies.

The Continuous Bag-of-Words (CBOW) method learns an embedding by predicting the current words based on the context. The context is determined by the surrounding words.

Continuous Skip-Gram has high best embeddings that can be learned pretty efficiently, particularly while comparing against neural probabilistic models. That manner low area and coffee time complexity to generate a wealthy representation. More than that, the bigger the dimensionality, the more functions we will have in our representation. But still, we can keep the dimensionality loads decreasing than a few other methods. It also allows us to correctly generate something like a billion-word corpora but encompass a group of generalities and keep the dimensionality small.

GloVe is an extension of word2vec, and a far higher one at that. There are several of classical vector models used for herbal language processing which are precise at taking pictures worldwide of a corpus, like LSA (matrix factorization).

They're excellent at worldwide information, but they don't seize meanings so well and sincerely don't have the cool analogy features built in.

GloVe's contribution changed into the addition of international statistics inside the language modeling challenge to generate the embedding. There is not any window characteristic for the local context. Instead, there is a phrase-context/phrase co-prevalence matrix that learns records across the whole corpora.

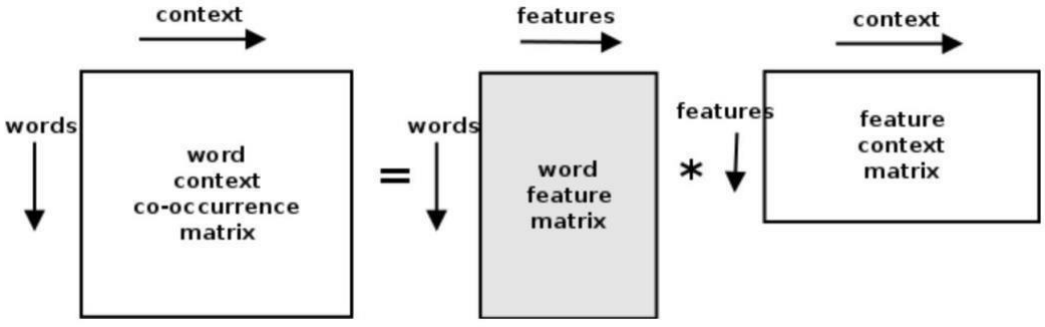


Figure-7. Global Vector model.

Now, with FastText we enter into the world of actually cool recent phrase embeddings. What FastText did was decide to compose sub-phrase data. It did so by splitting all phrases into a bag of n-gram characters (typically of length 3-6).

It might add these sub-words together to create an entire phrase as a final feature. The component that makes this powerful, it lets in FastText to naturally guide out-of-vocabulary words!

This is massive due to the fact in different approaches if the gadget encounters a phrase that it doesn't recognize, it simply has to set it to the unknown phrase. With FastText, we can supply meaning to words like circumnavigating if we simplest understand the phrase navigate because our semantic expertise of the word navigate can help users at the least provide a piece greater semantic statistics to circumnavigate, although it isn't always a phrase our device learned throughout the training.

Beyond that, FastText uses the skip-gram goal with bad sampling. All sub phases are fantastic examples, and then random samples from a dictionary of phrases inside the corpora are used as bad examples. These are the predominant things that FastText covered in its training.

Another simply cool issue is that Facebook, in developing FastText, has published pre-trained FastText vectors in 294 exceptional languages. This is something extraordinarily awesome, in my opinion, because it lets in builders to leap into making tasks in languages that normally don't have pre-educated word vectors at a totally low cost (since education their phrase embeddings takes numerous computational resources).

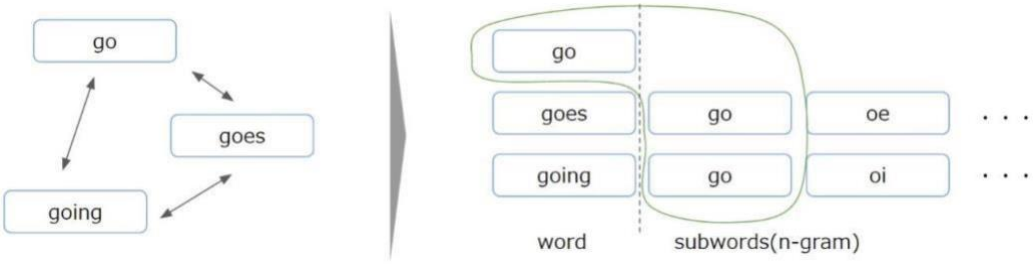


Figure-8. FastText model.

ELMo is a deep contextualized phrase representation model. ELMo is skilled as a bi-directional, layer LSTM language model. In real thrilling side-effect is that its final output is without a doubt an aggregate of its internal layer outputs. What has been located is that the bottom layer is ideal for things like POS tagging and other extra syntactic and useful tasks, while the higher layer is good for things like word-feel disambiguation and other higher-level, extra abstract responsibilities. When we combine these layers, we find that we honestly get incredibly excessive performance on downstream responsibilities out of the box.

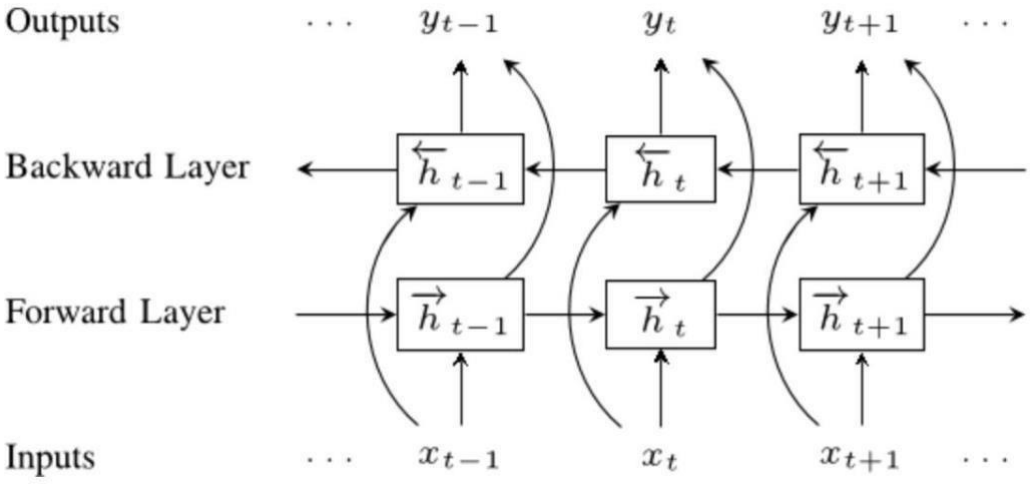


Figure-9. ELMo' bidirectional LSTM language model.

Clustering.

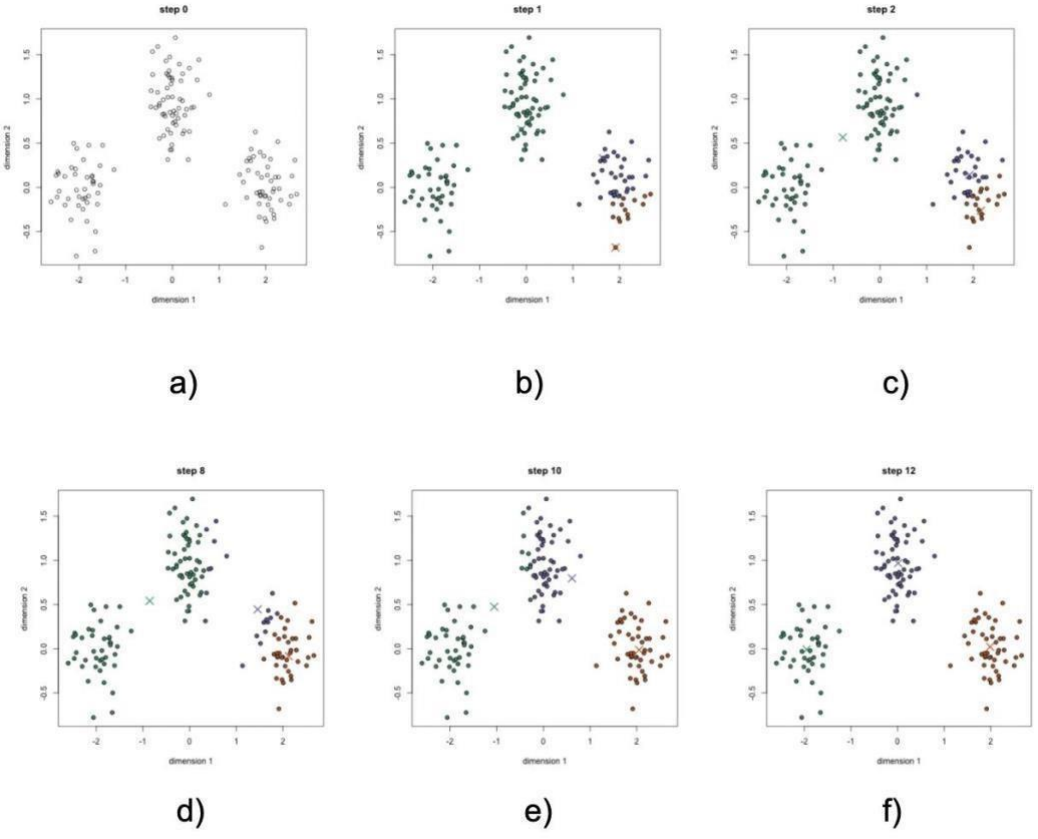
The most basic clustering algorithm that in this case can be applied is the algorithm for searching for connected components in a graph.

a) $compNum \leftarrow 0$
b) $for\ (u,v) \in E,\ do$
    i) $w(u,v) \leftarrow cos(u,v)$
c) $for\ v \in V,\ do$
    i) $c[v] \leftarrow nil$
d) $for\ v \in V\ do$
    i) $if\ c[v] = nil,\ then$
        1) $c[v] = nil,\ then$
        2) $Q \leftarrow \{v\}$
        3) $S \leftarrow \oslash$
        4) $while\ Q \neq \oslash,\ do$
            (a) $u \leftarrow pop(Q)$
            (b) $c[u] \leftarrow compNum$
            (c) $S \leftarrow S \cup \{v\}$
            (d) $for\ n \notin S,\ (u,n) \in E,\ do$
                (i) $if\ w(u,n) \in E,\ do$
                    (1) $if\ w(u,v) \geq threshold\ and\ c[n] = nil,\ then$
                        a. $Q \leftarrow Q \cup \{n\}$
    ii) $compNum \leftarrow compNum + 1$

Where E, V- the set of edges and vertices of the graph, (u, v), (u, n) - the edges, v, u, n are vertices, w (u, v) is the edge weight (u, v), c[v] is the number of the component (cluster) of the vertex v, Q is the queue of vertices to visit, S many vertices, compNum is the number of the current component.

The algorithm works as follows: a whole graph is constructed; the values of the rims are set as the cosine distance among the vertex vectors. Further, from every vertex not marked with the cluster variety, the BFS set of rules is launched (breadth-first search), ignoring edges less than a selected len value, and putting

any vertex that controls to reach its cluster quantity. This manner is repeated till all nodes are assigned a cluster number. This algorithm is quite simple to enforce and suggests appropriate results, also, it does not now need to understand the number of clusters in advance, the handiest threshold parameter is the threshold underneath which the rims are ignored. Another, extra popular, but not much less simple k-approach clustering algorithm considers vectors as factors in space, selects okay random centroid factors and all article vector points closest to them, forming okay random groups (Fig. 10, b). Next, each group determines the midpoint, which will become a brand-new centroid (Fig. 10, c). The method is repeated until the points prevent moving (Fig. 10, d – f).



Visual representation of k-means.

Figure-10.

The k-means algorithm in empirical comparison shows the results of a slightly good graph approach and has more parameters, which allows you to fine-tune it depending on the size of the data. If there is a lot of data for clustering, as is often the case, then the algorithm may not have enough RAM. To solve this problem, there is a k-means option that accepts data in batches, unfortunately, the operation time of the algorithm increases, depending on the size of the parts.

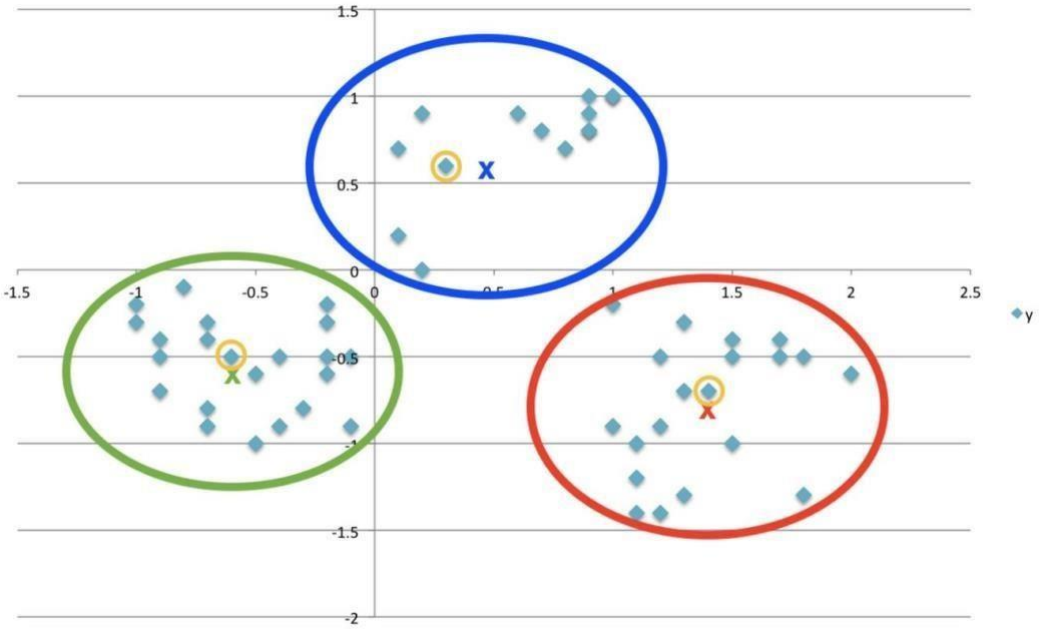


Figure- 11. Each point represents a document which corresponds to a set of sentences, regarding its relevance.

Chapter 2. Proposed Method.

Chain of Steps.

Bidirectional Encoder Representations from Transformers.

Summarization prototype is composed of the main technologies based on each NLP task; we applied several methods for record summarization, thinking about main troubles that face the way of the ideal summarization approach. The first step, we observe BERTSum extractive to decrease initial fee of record and transfer pre-summary to the Universal Sentence Encoder version to get semantics and reduce redundancy on the manner of applying to that technique embeddings, ranking approach for sentence and report level.[7]

There are three types of embeddings applied to our text before to feeding it to the **BERT:**

a. Token Embedding - Words are converted into a fixed dimension vector. [CLS] and [SEP] are added at the beginning and end of sentences respectively.
b. Segment Embeddings - It is used to distinguish, or we can say classify the different inputs using binary coding. For example, input1 = “I love books” and input2 = ”I love sports”. Then after the processing through token embedding, we would have:

[CLS], I, love, books, [SEP], I, love, sp or ts

Segment embedding would result to

Input1 = 0, Input2 = 1

c. Position Embeddings - BERT can help input collection of 512. Thus the resulting vector dimensions will be (512,758). Positional embedding is used because the location of a phrase in a sentence may also modify the contextual meaning of the sentence and therefore should not have equal illustration as a vector.[8]

   For example:

   "We did not play, however, we were spectating"

   In the sentence above "we" must not have the same vector representations.

d. We can have different types of layers within the BERT model each having its specifications:

   Inter Sentence Transformer - In the inter sentence transformer, a simple classifier is not used. Rather various transformer layers are added into the model only on the sentence representation making it more efficient. This helps in recognizing the important points of the document.

   $h\tilde{}_I = LN(h_{I-1} + MHAtt(h_{I-1}))$ $h_I = LN(h\tilde{}_I + FFN(h\tilde{}_I))$

Where $h = PosEmb(T)$ and T are the sentence vectors output by BERT, PosEmb is the function of adding positional embeddings (indicating the position of each sentence) to T, *LN* is the layer normalization operation, *MHAtt* is the multi-head attention operation and the superscript i indicate the depth of the stacked layer.[9]

These are followed by the sigmoid output layer

$$\hat{Y}_i = \sigma(W_o h_{Li} + b_o)$$

h_{Li} is the vector for $sent_i$, from the top layer (the L-th layer) of the Transformer

e. We could select the Transformer with L=2, which performs best. That model named BertSumExt could be used directly.

The loss of the model is the binary classification entropy of prediction y_i against gold label y_i Intersentence Transformer layers are jointly fine-tuned with BertSum. Using Adam optimizer with $\beta_1 = 0{,}9$ and $\beta_2 = 0{,}99$. Vary the learning rate over the course of training, according to the formula:

$$lrate = d_{-m0o.d5el} * min(stepnum_{-0.5}, stepnum * warmupsteps_{-1.5})$$

This corresponds to increasing the learning rate linearly for the first *warmupsteps* training steps and decreasing it

thereafter proportionally to the inverse square root of the step number. We used warmupsteps=4000.

e. *Input Document → Token Embeddings →*

→ Interval Segment Embeddings →

Positional Embeddings → BERT →

→ Summarization Layers → Summary

f. Create an object for summarizer function.

g. Store to one variable extracted sentence for future transfer to USE.

h. Call the function to pass our text for summarization.

```
------------------------------------------
1 ROUGE-1 Average_R: 0.53377 (95%-conf.int. 0.53109 - 0.53650)
1 ROUGE-1 Average_P: 0.38181 (95%-conf.int. 0.37940 - 0.38435)
1 ROUGE-1 Average_F: 0.43053 (95%-conf.int. 0.42843 - 0.43265)
------------------------------------------
1 ROUGE-2 Average_R: 0.24965 (95%-conf.int. 0.24688 - 0.25243)
1 ROUGE-2 Average_P: 0.17934 (95%-conf.int. 0.17716 - 0.18164)
1 ROUGE-2 Average_F: 0.20159 (95%-conf.int. 0.19929 - 0.20386)
------------------------------------------
1 ROUGE-L Average_R: 0.48854 (95%-conf.int. 0.48591 - 0.49132)
1 ROUGE-L Average_P: 0.35008 (95%-conf.int. 0.34772 - 0.35261)
1 ROUGE-L Average_F: 0.39447 (95%-conf.int. 0.39236 - 0.39665)

[2020-04-22 00:24:46,673 INFO] Rouges at step 0
>> ROUGE-F(1/2/3/l): 43.05/20.16/39.45
ROUGE-R(1/2/3/l): 53.38/24.96/48.85

[2020-04-22 00:24:46,673 INFO] Validation xent: 5.35811 at step 0
```

Figure - 12. Rouge results in replication from the original paper. On the BertSum we received the same Rouge score to have state of the art results, so then we transmit them to the next step of summarization approach.

Corresponding to the data taken into account, we may assume more data we train the rouge score could be grown not so much as could be expected on the one hand.[10]

Universal Sentence Encoder.

Select Universal Sentence Encoder to reduce similarity between sentences, autonomously the highest score sentences in the correlation matrix. How to achieve:

a) Install / import all necessary libraries
b) Download officially published encoder model
c) Load data coming from BERTSUM
d) Generate embedding with USE (the model will accept a list of strings as input and return 512-d vector as output. If the list contains “N” strings, the output would be of shape (N,512))
e) Giving related title (input of which should contain the theme)
f) Compute inner product, the similarity between normalized vectors (cosine similarity) {correlation matrix will have a shape of (N,1), where N is the number of strings in the text list. Each of the raw in the (N,1) matrix gives its similarity score, i-th row gives the similarity between the i-th string in the text list and related title}
g) Regarding similarity value we select the most valuable i-th strings.
h) Summarization will consist of several steps like: The complete process can be divided into several phases, as follows: Encoding multiple sentences from the input document are encoded to be preprocessed. Each sentence is preceded by a CLS

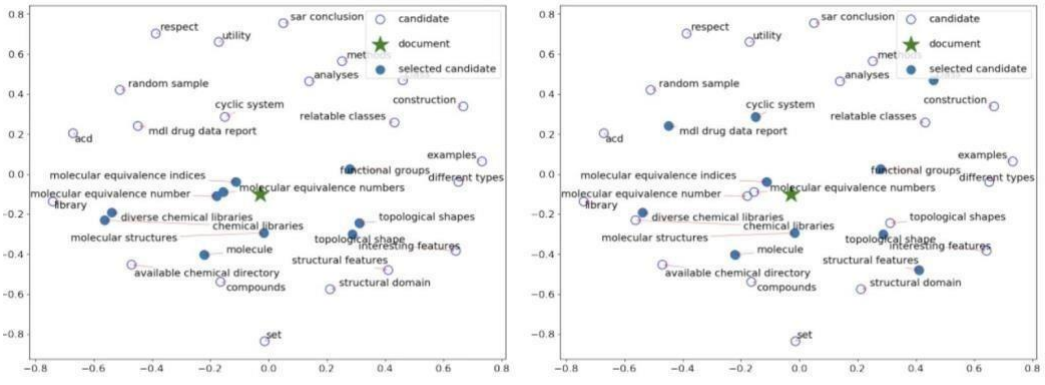
tag and succeeded by a SEP tag. The CLS tag is used to aggregate the features of one or more sentences.

Interval Segment Embeddings is dedicated distinguishing sentences in a document. Sentences are assigned either of the labels discussed above. For example, $\{sent_i\} = E_A$ *or* E_B depending upon i. The criterion is basically as E_A for even i and E_B for odd i.

EmbedRank++

Since EmbedRank extracts the top n phrases that are close to the document vector, similar phrases may be included in the n phrases. For example, in the figure (a) below, "molecular equivalence numbers", "molecular equivalence number", and "molecular equivalence indices" are selected as the key phrases. The amount of information as a phrase set is small, and it cannot be said that it is a preferable key phrase set.

Specifically, we will introduce Maximal Marginal Relevance (MMR). The idea is very simple, the document vector and phrases of similarity a positive score (EmbedRank), already a negative score the degree of similarity between the selected key phrases set, ranking based on the total number of sentences.



(a) EmbedRank (without diversity)

(b) EmbedRank++ (with diversity)

Figure- 13. Differences in embedding ranking.

Attention.

In many tasks, such as gadget translation or speak generation, we have a chain of phrases as an input (an original textual content in English) and would love to generate another series of phrases as an output (a translation to Korean). Neural networks, specifically recurrent ones (RNN), are well suited for solving this type of task. The "sequence-to-sequence" neural community models are widely used for NLP.

A popular type of these models is an "encoder-decoder". There, one part of the network — encoder — encodes the input series into a fixed-length context vector. This vector is an internal representation of the text. This context vector is then decoded into the output sequence through the decoder.

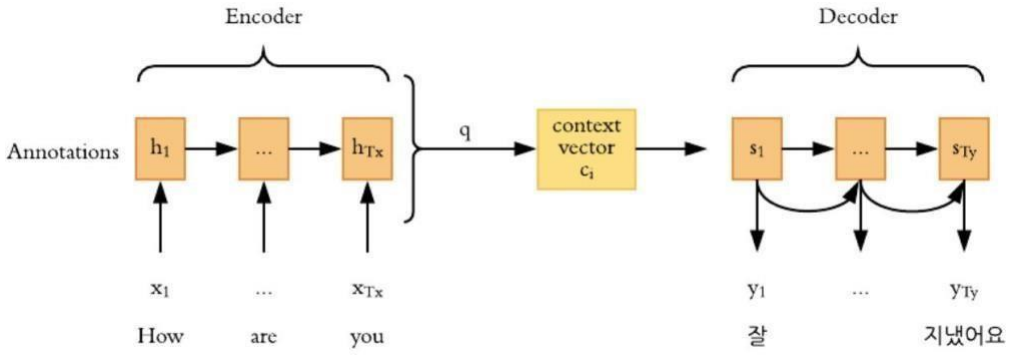


Figure- 14. Encoder-Decoder neural network architecture.

Here *h* denotes hidden states of the encoder and *s* of the decoder. *Tx* and *Ty* are the lengths of the input and output word sequences respectively. *q* is a function that generates the context vector out of the encoder's hidden states. It can

be, for example, just q({h_i}) = h_T. So, the last hidden state is taken as an internal representation of the entire sentence.[11]

Each time the version predicts an output word, it best uses elements of an entry in which the maximum relevant statistics is concentrated alternatively of a whole sentence. In other words, it simply pays interest to some input words.

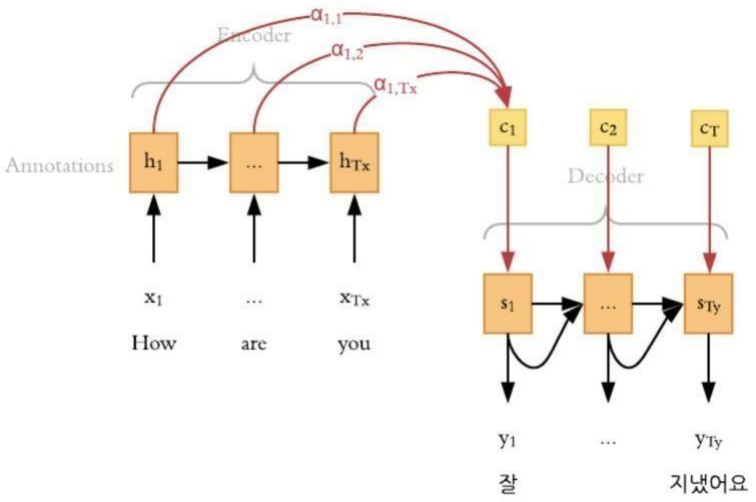


Figure- 15. An illustration of Attention mechanism.

Encoder works as usual, and the distinction is best on the decoder's part. As we can see from a picture, the decoder's hidden state is computed with a context vector. we do not consider a single context vector c, however a separate context vector c_i for each target word.

These context vectors are computed as a weighted sum of annotations generated by way of the encoder. Bidirectional LSTM's annotations are concatenations of hidden states in ahead and backward directions.

The weight of every annotation is computed by using an alignment version which scores how nicely the inputs and the output match. An alignment version is a feedforward neural network, for instance. In general, it may be any other version as well.

As a result, the alphas — the weights of hidden states whilst computing a context vector — display how essential a given annotation is in deciding the following country and generating the output word. These are the attention scores.

BERT.

BERT (from bi-directional representations of encoders from Transformers) is a neural network-primarily based era that facilitates to recognize and procedure natural language. It is utilized by Google to distinguish context in the seek query. For example, within the phrases “nine to five” and “at quarter to five,” the preposition “k” has two exceptional meanings, which are not continually obvious for locating systems. BERT enables distinguish among language nuances to provide more relevant results. BERT is an open-source era. This method can be used to educate systemic language methods and other tasks.

Pre-trained representations can either be context-unfastened or contextual, and contextual representations can similarly be unidirectional or bidirectional. Context-free models consisting of word2vec or GloVe generate a single word embedding representation for each word in the vocabulary and do not capture polysemy. An essential note right here is that BERT isn't skilled for semantic sentence similarity directly just like the Universal Sentence Encoder or InferSent models. Therefore, BERT embeddings cannot be used directly to apply cosine distance to measure similarity. However, there are easy wrapper offerings and implementations just like the famous bert-as-a-service that may be used to that effect.[12]

For example, the word “train” would have the same context-free representation in “goods train” and “train the model”. Contextual models instead generate a representation of each word that is based on the other words the

sentence. For example, in the sentence “we will train the model” a unidirectional contextual model would represent “train” based on “we will” but not “the model” However, BERT represents “train” using both its previous and next context — “we will … the model” — starting from the very bottom of a deep neural network, making it deeply bidirectional.

BERT is designed to pre-train deep bidirectional representations from unlabeled text by way of mutually conditioning on each left and proper context in all layers. As a result, the pre-skilled BERT model can be finetuned with simply one extra output layer to create contemporary fashions for a wide variety of tasks, consisting of query answering and language inference, without giant project-specific architecture modifications. BERT is conceptually easy and empirically powerful. It obtains new contemporary effects on eleven herbal language processing tasks.

BERT learns example representations with the aid of attending to critical content words, wherein the significance is signaled with the aid of word and role embeddings as well as pairwise word relationships. Nonetheless, it remains an open query of whether BERT can effectively weave the means of topically essential words into representations. The word “border” is topically essential if the input document discusses border security.

A topic word is possible to be repeatedly cited inside the input record but less frequently elsewhere. Because sentences containing topical phrases are frequently deemed precisely worthy, it is ideal to represent sentence singletons and pairs primarily based on the amount of topical content material they convey.

The predominant hindrance is that modern language models are unidirectional, and this limits the selection of architectures that may be used at some stage in pre-training. Pre-skilled representations reduce the want for many heavily engineered venture precise architectures. BERT is the first finetuning primarily based illustration model that achieves trendy performance on a huge suite of sentence-stage and token-degree tasks, outperforming many challenges unique architectures.

Implementation of BERTSum in official paper defines a good approach to be considered in our work, as a section of fine-tuning parameters using labeled data from the tasks. Input illustration is set as much as unambiguously represents both single sentences and several sentences in one token collection. A “sentence” can be an arbitrary span of contiguous text, instead of an actual linguistic sentence. A “sequence” refers back to the input token sequence to BERT, which may be a single sentence, or two sentences packed together. Input representation can be constructed by summing the corresponding token, segment, and function embeddings.

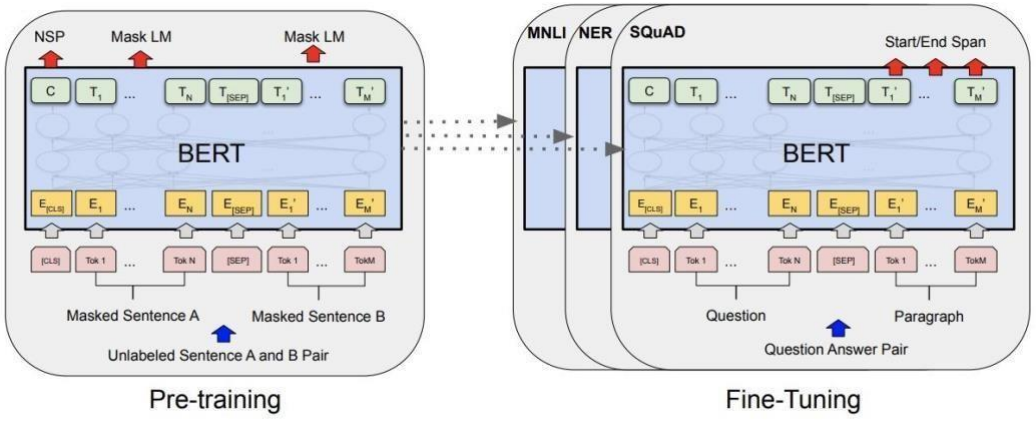


Figure- 16. Pre-training and fine-tuning procedures for BERT.

Since BERT is trained as a masked-language model, the output vectors are grounded to tokens in place of sentences, whilst in extractive summarization, maximum models manage sentence-degree representations. Although segmentation embeddings represent extraordinary sentences in BERT, they only apply to sentence pair inputs, at the same time as in summarization we ought to encode and manage multi-sentential inputs (BERTSum).

Let d denote a document containing sentences [sent<1>, sent<2>, · · · , sent<m>], where senti is the i-th sentence in the document. Extractive summarization can be defined as the task of assigning a label yi ∈ {0, 1} to each sent i, indicating whether the sentence should be included in the summary. It is assumed that summary sentences represent the most important content of the document. With BERTSUM, vector t_i which is the vector of the i-th [CLS] symbol from the top layer can be used as the representation for sent i. Several inter-sentence Transformer layers are then stacked on top of BERT outputs, to capture document-level features for extracting summaries. PosEmb adds sinusoid positional embeddings to T, indicating the position of each sentence.

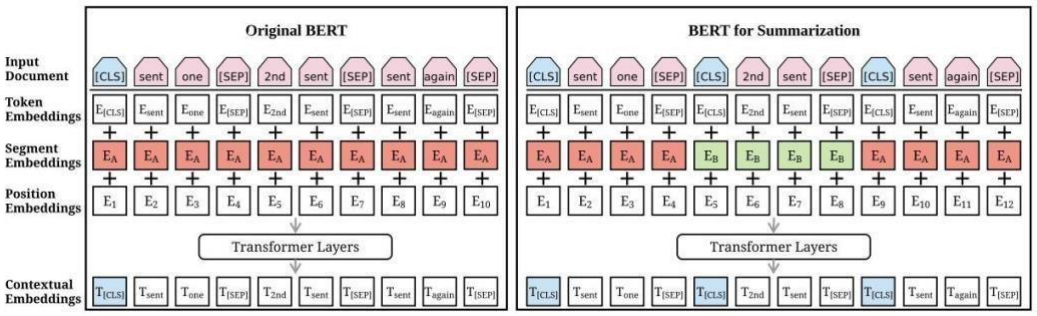


Figure- 17. Differences of original BERT and BERTSUM for summarization.

Oracle precise sentences are fairly smoothly dispensed throughout documents, whilst summaries created with the aid of TransformerEXT ordinarily focus on the first document sentences. BERTSUMEXT outputs are more like Oracle summaries, indicating that with the pre-trained encoder, the version relies much less on shallow role features, and learns deeper record representations.

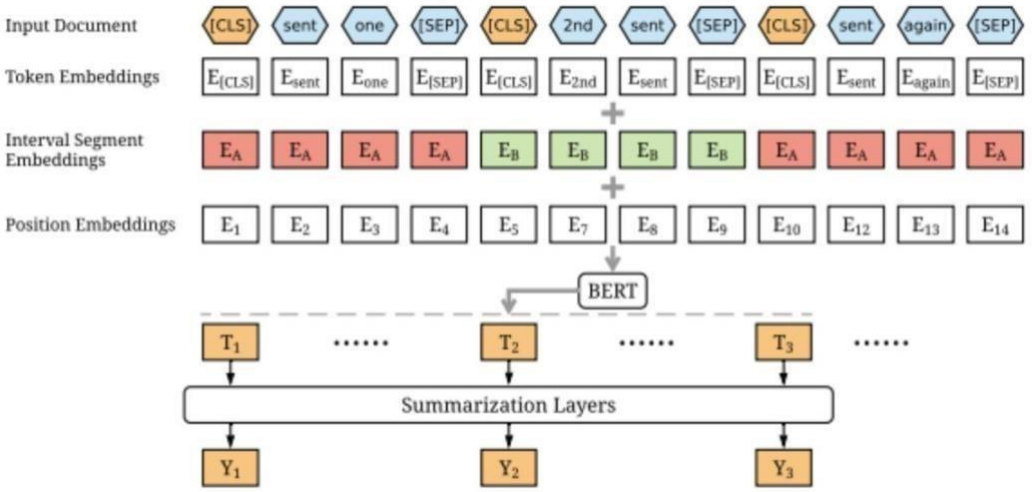


Figure- 18. BertSum model architecture.

Proper language representation is key for general-reason language know-how with the aid of machines. Context-unfastened fashions which include word2vec or GloVe generate single phrase embedding representation for each word inside the vocabulary. For example, the word “bank” might have an equal representation in “bank deposit” and in “riverbank”. Contextual models rather generate an illustration of each word that is based totally on the opposite words in the sentence.

BERT, as a contextual model, captures those relationships in a bidirectional way. BERT changed into constructed upon recent work and clever ideas in pre-training contextual representations including Semi-supervised Sequence Learning, Generative

Pre-Training, ELMo, the OpenAI Transformer, ULMFit and the Transformer. Although those fashions are all unidirectional or shallowly bidirectional, BERT is completely bidirectional.

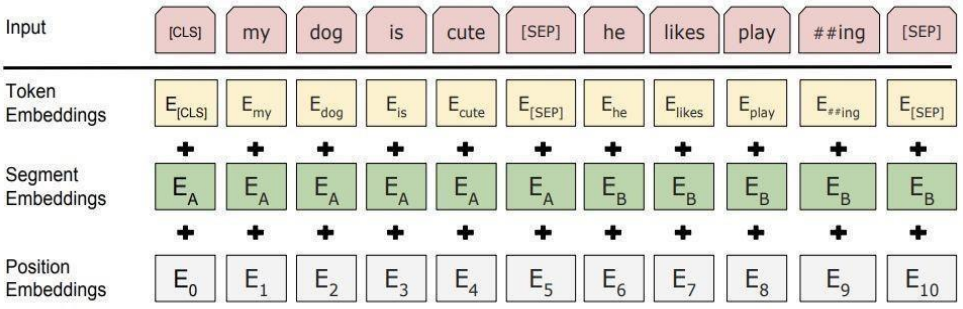


Figure- 19. BERT input representation.

BERT Input:

The BertSum input format differs slightly from the original BERT. To represent individual sentences, add external [CLS] marks at the beginning of each sentence and each symbol [CLS] collects features for the previous sentence. Also use interval range insertion to distinguish between multiple sentences in a document. To embed a segment, assign EA or EB, depending on whether the offer is odd or even. For example, the embedding [EA, EB, EA, EB, EA] is assigned for the document [sent1, sent2, sent3, sent4, sent5]. Therefore, document representations are examined hierarchically, with the lower levels of the transformer representing adjacent sentences, while the higher levels, in combination with attention to oneself, represent a discourse consisting of several

sentences.For training target, BertSum labeled representative sentences, for documents [sent1, sent2, sent3, sent4, sent5] has label [label1, label2, label3, label4, label5], where label is {0, 1}, as the golden standard.

BERT Output:

BertSum publishes a list of ratings that show the representativeness of the proposal against the documents. For documents [sent1, sent2, sent3, sent4, sent5] there are points [score1, score2, score3, score4, score5].

The higher the score, the more representative the proposal will be. Bert was sent to the summary layers for his summary. In the dissertation, the author verified the number of total level structures, and can still be selected on the published github. There are RNN-based, classifier-based, and transformer-based options. It just adds up the number of base layers and ends with the scores.

Tokenization. Bert Tokenizer.

Required Formatting:

Add special tokens (SEP, CLS) at the beginning and end of each sentence. Fill and crop all offer to a constant length (512 tokens, PAD). Distinguish real tokens from wildcard tokens using an attention mask. At the end of each sentence we need to add a special SEP token. This token is an artifact of a two-sentence task, in which BERT receives two separate sentences and is asked to determine something.

We need to provide a special CLS token before each offer starts. This token has a special meaning. BERT consists of 12 transformer layers. Each converter creates a list of token tabs and specifies the same number of tabs at the output. The first token in each sequence is always the custom CLS ranking token. The final hidden status for this token is used to represent the aggregate sequence for classification tasks. Since BERT is trained to use only this CLS token for sorting, we know that the model was motivated to encode everything needed for the sorting step into a single embedding vector with a value of 768.

BERT has two restrictions:

- All offers must be filled in or cut to a fixed length.
- The maximum offer length is 512 tokens.

The tokenizer.encode function combines several steps:

- Break the sentence into tokens.
- Add special CLS and SEP tokens.
- Assign tokens to their identifiers.

Attention helps clarify which tokens are real words and which tokens are. The BERT dictionary does not use the identifier 0, so if the token identifier is 0, it is padded, otherwise, it is a real token.

NLP Transformer is a new architecture that aims to solve tasks from sequence to sequence and to easily manage long-term dependencies. The self-service concept allows the model to look at other words in the input sequence to better understand a particular word in the sequence. Besides, the self-esteem in the transformer architecture is not only calculated once but several times in parallel and independently. Therefore, it is known as multi-head care.

Universal Sentence Encoder.

Many NLP applications need to calculate the similarity of values between two short texts. Modern search engines calculate the relevance of a document for a query and not just a simple word match on both. This is accomplished by using text similarities by creating useful attachments from short texts and calculating the cosine similarities between them.

Word2vec and GloVe similarly use word insertion and have become popular models for finding semantic similarities between two words. However, sentences inherently contain more information with links between multiple words, and a unified approach to calculating sentence attachments is still being considered.

The most common way to assess the semantic similarity of a baseline between a pair of sentences is to average the nested words of all the words in two sentences and calculate the cosine between the nests received. This simple baseline can be improved by methods such as ignoring stop words and calculating TF-IDF weighted average values, etc. Alternatively, methods such as Word Mover Distance (WMD) and Smooth Return Frequency (SIF) can be used in place of the basic approach to achieve greater precision.

All these methods share two important characteristics:

1. They do now not take phrase order into account as they may be primarily based on the bag-of-phrase method. This is a major disadvantage on account that differences in word order can often completely regulate the means of a sentence and sentence embeddings must seize these variations. Capturing references, relationships, and sequence of phrases in sentences are crucial for a system to recognize the herbal language.
2. Their phrase embeddings have been discovered in an unsupervised manner. While this isn't a significant problem in itself, it has been found that supervised training can assist sentence embeddings to examine the meaning of a sentence extra directly.

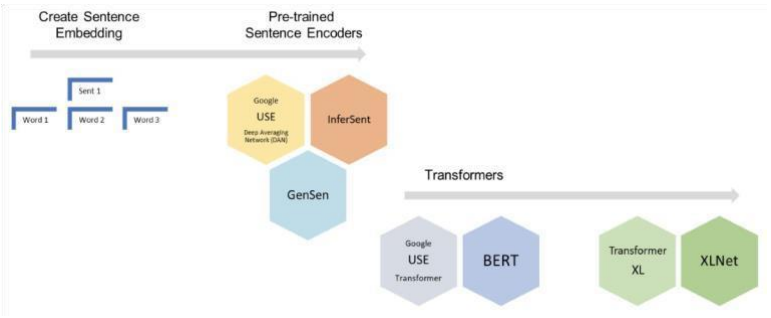


Figure- 20. Sentence similarity models.

The similarity of sentences is a relatively complex phenomenon in comparison to the similarity of words because the meaning of a sentence depends not only on the words it contains but additionally on how they are combined. Semantic similarity can have more than one dimension, and sentences may be similar in one however contrary to another.

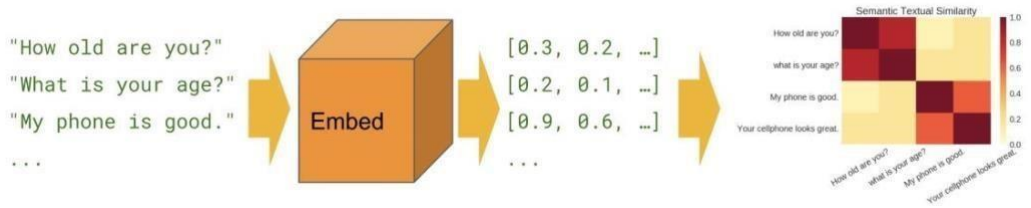


Figure- 21. Embedding similarity.

Semantic similarity is a degree of the pairs to which texts have the same sense. This is generally useful for acquiring good coverage of many methods of expressing mind in speech without the want for manual numbering. Simple applications include enhancing the attain of systems that cause the behavior of positive keywords, phrases, or sentences.[13]

The Centroid embedding method uses phrase embedding as an opportunity to BOW rendering. Here we pay attention to techniques that percentage this function. Methods based totally on matrix factorization, such as hidden semantic analysis (LSA) and non-negative matrix factorization (NMF), aim to grow hidden factors by using growing dense and compact representations of sentences.

Like all models, BERT isn't an ideal answer for all problem regions and, depending on the task, it can be essential to evaluate numerous fashions for their performance. For example, if in a certain area there is a set of phrases that significantly modifies or affects the overall meaning of the text, word fashions like GloVe can capture these nuances as compared to BERT.[14]

Models including generic provide encoders, which were specially designed to embed sentences and similarities, in a few cases can also give better effects than BERT. Therefore, it is really useful to perform some preliminary checks on information samples and empirically test the experience of the version to determine the direction and sort of implementation required.

Universal Sentence Encoder encodes textual content into multidimensional vectors that may be used to classify text, semantic similarity, grouping, and other natural language duties.

The model is skilled and optimized for the text of greater than one phrase, together with sentences, phrases, or brief paragraphs. He is educated in diverse information resources and tasks to dynamically perform various duties to apprehend natural language. The input makes use of variable duration English textual content, and the output makes use of a 512-dimensional vector. This encoder differs from built-in word-level fashions in that we train some herbal language prediction problems, which include modeling the meaning of phrases and now not just identical words.

Chapter 3. Results. Future Work and Conclusion.

Evaluation

In this segment we especially refer to assessment strategies that can be carried out to brief descriptions. These summaries encompass excerpts from the source textual content. This manner allows you to decide their exceptional by comparing their content (a sequence of words) with reference summaries.

If this comparison isn't possible (inside the case of summary), manual strategies are the simplest dependable solution inside the framework of the modern nation of the art. Even for a summary, the assessment strategies vary from simple terms manual approaches to in basic terms automatic strategies, and of course there are many alternatives among them. Manual procedures are understood as techniques in which someone evaluates the candidate's data from exclusive points of view, reporting, grammar, or style. This sort of evaluation is necessary, however, is known to be very subjective.

The benefits of the proposed method are certainly visible. It produces very competitive results, which seemingly outperforms the coverage baseline in both years. More important, it is in advance of the first-class machine in DUC2004 on ROUGE-1. It is pretty encouraging to us. Notice that during our contemporary experiments which attention on the rating and clustering relationships, the position of a sentence in the report isn't always considered yet. Nevertheless, the position characteristic has been hired in all of the participating structures as one of the most substantial features.

DATASET	SOURCE	SUMMARY	#PAIRS
Gigaword (Rush et al., 2015)	**the first sentence** of a news article	**8.3 words** title-like	**4 Million**
CNN/Daily Mail (Hermann et al., 2015)	**a news article**	**56 words** multi-sent	**312 K**
TAC (08-11) (Dang et al., 2008)	**10 news articles** related to a topic	**100 words** multi-sent	**728**
DUC (03-04) (Over and Yen, 2004)	**10 news articles** related to a topic	**100 words** multi-sent	**320**

Table 1: A comparison of datasets available for sent. summarization (Gigaword), single-doc (CNN/DM) and multi-doc summarization (DUC/TAC). The labelled data for multi-doc summarization are much less.

Figure – 22. A comparison of datasets available for sentences summarization (Gigaword), single-document (CNN/DM) and multi-document summarization (DUC/TAC). The labeled data for multi-document summarization is much less.

Automated strategies compare textual content segments in the dataset with one or more abstracts. This method is simple to reproduce, however, cannot be used if the system makes use of reformulation methods. With blended strategies, you may manually examine and compare the most important statistics and classify the information of the applicants accordingly (the maximum important facts ought to be included inside the data regardless of the language composition). Many one-of-a-kind strategies to final evaluation had been proposed within the past two decades. This prevents us from being exhaustive. Instead, we are conscious of three strategies that have been extensively used in recent assessment campaigns (especially in the latest NIST textual content analysis conferences): ROUGE is a completely automatic method, PYRAMID is a combined method.[15]

The maximum obvious and easiest way to assess a resume is to have the firstrate of the evaluators evaluated. For example, for the DUC, the judges needed

to determine the scope of the curriculum vitae, this means that that they needed to make a comprehensive evaluation to evaluate the quantity to which the textual content supplied as entered covers the applicant's curriculum vitae. In later structures, especially inside the TAC, consultation-oriented summaries must be created: The judges ought to determine the responsiveness of the call, i.e. to what volume this precis reacts to the request specified in the entry.

A guide markup can also include some data for assessing the satisfaction and clarity of textual content. An accurate resume ought to be:

- syntactically correct;
- semantically consistent;
- logically organized
- without reservation.

These numerous factors are too complex to be fully calculated, mainly the semantic consistency and logical organization. Human judgment is required to get a reliable assessment of these diverse aspects. For the DUC and TAC campaigns, human experts had to give every candidate extraordinary rating based on the subsequent indicators:

- grammar;
- non-redundancy;
- focus (integration of the maximum important facts within the supply textual content);
- structure and coherence.

The experts needed to assign a rating from 0 (empty) to 10 (excellent) for each of these signs.

The ROUGE (Recall-Oriented Understudy for Gisty Evaluation) measures are primarily based on the assessment of n-grams (i.E. a chain of n factors) among the candidate summary (the precis to be evaluated) and one or several reference summaries. Most of the time, several reference summaries are used for contrast, which permits more flexibility and a fairer assessment. There are numerous editions of ROUGE and we present the maximum widely used of them below.

• ROUGE-n: This measure is based on a simple assessment of n-grams (maximum of the time a chain of 2 or 3 elements, more not often 4). A series of n-grams (subsequently a series of sequences of n consecutive words) is extracted from the reference summaries and the candidate summary. The score is the ratio between the range of not unusual ngrams (between the candidate summary and the reference precis) and the variety of n-grams extracted from the reference summary best.

• ROUGE-L: To conquer some of the shortcomings of ROUGE-n, greater precisely the truth that the measure can be based totally on too small sequences of text, ROUGE-L takes into consideration the longest, not unusual sequence among two sequences of text divided by using the length of one of the textual contents. Even if this method is extra flexible than the preceding one, it nonetheless suffers from the reality that each one n-grams have to be non-stop.

• ROUGE-SU: Skip-bi-gram and unigram ROUGE takes into consideration bi-grams in addition to unigrams. However, the bi-grams, rather than being just non-stop

sequences of words, allow insertions of words among their first and closing elements. The maximal distance among the two elements of the bi-gram corresponds to a parameter (n) of the degree (often, the degree is instantiated with n = 4). During TAC 2008, it has been shown that ROUGE-SU becomes the most correlated measure with human judgments.

ROUGE has been very beneficial for comparing extraordinary summaries based totally on extractive methods, however, the use of n-grams handiest is a sturdy limitation because it requires an exact match of specific gadgets of textual content. More recently, other evaluation measures have been developed to better seize the semantics of texts.[16]

Results.

Once we apply the summarization approach proposed in this work, we evaluated the summary on the rouge score and received such kind of results, which is shown below:

```
1 ROUGE-1 Average_R: 0.20950 (95%-conf.int. 0.18916 - 0.23098)
1 ROUGE-1 Average_P: 0.98189 (95%-conf.int. 0.94727 - 1.00000)
1 ROUGE-1 Average_F: 0.34431 (95%-conf.int. 0.31538 - 0.37358)
---------------------------------------------
1 ROUGE-2 Average_R: 0.19830 (95%-conf.int. 0.17045 - 0.22311)
1 ROUGE-2 Average_P: 0.93133 (95%-conf.int. 0.84590 - 0.97823)
1 ROUGE-2 Average_F: 0.32601 (95%-conf.int. 0.28438 - 0.36193)
---------------------------------------------
1 ROUGE-3 Average_R: 0.19046 (95%-conf.int. 0.16186 - 0.21566)
1 ROUGE-3 Average_P: 0.89818 (95%-conf.int. 0.79999 - 0.95390)
1 ROUGE-3 Average_F: 0.31334 (95%-conf.int. 0.26966 - 0.35003)
---------------------------------------------
1 ROUGE-4 Average_R: 0.18402 (95%-conf.int. 0.15614 - 0.20894)
1 ROUGE-4 Average_P: 0.87200 (95%-conf.int. 0.77268 - 0.93010)
1 ROUGE-4 Average_F: 0.30299 (95%-conf.int. 0.26092 - 0.33922)
---------------------------------------------
1 ROUGE-L Average_R: 0.12902 (95%-conf.int. 0.11138 - 0.14694)
1 ROUGE-L Average_P: 0.60625 (95%-conf.int. 0.53738 - 0.67159)
1 ROUGE-L Average_F: 0.21213 (95%-conf.int. 0.18546 - 0.23899)
---------------------------------------------
1 ROUGE-W-1.2 Average_R: 0.02564 (95%-conf.int. 0.02137 - 0.02974)
1 ROUGE-W-1.2 Average_P: 0.46902 (95%-conf.int. 0.39694 - 0.54119)
1 ROUGE-W-1.2 Average_F: 0.04856 (95%-conf.int. 0.04056 - 0.05611)
---------------------------------------------
1 ROUGE-S* Average_R: 0.03873 (95%-conf.int. 0.03111 - 0.04732)
1 ROUGE-S* Average_P: 0.83631 (95%-conf.int. 0.75750 - 0.90234)
1 ROUGE-S* Average_F: 0.07377 (95%-conf.int. 0.05971 - 0.08950)
---------------------------------------------
1 ROUGE-SU* Average_R: 0.03915 (95%-conf.int. 0.03149 - 0.04772)
1 ROUGE-SU* Average_P: 0.83820 (95%-conf.int. 0.75970 - 0.90361)
1 ROUGE-SU* Average_F: 0.07453 (95%-conf.int. 0.06043 - 0.09022)
```

Figure – 23. Final Rouge score for the proposed method.

For an exact score of each rouge section we have to take an average of them, so then it could be represented in such a way:

Rouge-1	Rouge-2	Rouge-3	Rouge-4	Rouge-L	Rouge-S*	Rouge-SU*	Rouge-W-1.2
0.512	0.486	0.468	0.454	0.316	0.317	0.318	0.182

Figure – 24. Exact Rouge scores.

For the best representation of the summary, we offer a solution to reorder sentences of the final summary. We provide a method reordering the sentences in a text, using BERT for Next Sentence Prediction. It is an important NLP task, especially when you have unordered text coming from different sources such as in Summarization of Multi-Summarization tasks.

The transformer library from HuggingFace to import BertTokenizer, BertForNextSentencePrediction is used. The tokenizer is used to tokenize the sentences used in the Next-Sentence-Prediction Model. The bert-base-multilingual-cased as pretrained weights are used to support cased text and several languages. The transformer library is used to build the tensors for the model, Finally, tensorflow.keras is used later to compute the softmax from the logits in the Next-Sentence- Prediction task.

The predict_next_sentence_prob function aims to return the probability that the second sentence provided in the input is the continuation of the first one.

```
def predict_next_sentence_prob(sent1, sent2):
    # encode the two sequences. Particularly, make clear that they must be
    # encoded as "one" input to the model by using 'seq_B' as the 'text_pair'
    # NOTE how the token_type_ids are 0 for all tokens in seq_A and 1 for seq_B,
    # this way the model knows which token belongs to which sequence
    encoded = tokenizer.encode_plus(sent1, text_pair=sent2)
    encoded["input_ids"] = tf.constant(encoded["input_ids"])[None, :]
    encoded["token_type_ids"] = tf.constant(encoded["token_type_ids"])[None, :]
    encoded["attention_mask"] = tf.constant(encoded["attention_mask"])[None, :]

    # a model's output is a tuple, we only need the output tensor containing
    # the relationships which is the first item in the tuple
    outputs = nsp_model(encoded)
    seq_relationship_scores = outputs[0]

    # we need softmax to convert the logits into probabilities
    # index 0: sequence B is a continuation of sequence A
    # index 1: sequence B is a random sequence
    probs = tf.keras.activations.softmax(seq_relationship_scores, axis=-1)
    return probs.numpy()[0][0]
```

Figure – 25. Predict the next sentence function.

The create_correlation_matrix takes a string array-like and creates a correlation matrix which takes care of all probabilities computed over the sentences given in input.

```
import numpy as np
def create_correlation_matrix(sentences: list):
    num_sentences = len(sentences)
    correlation_matrix = np.empty(shape=(num_sentences, num_sentences),
                                  dtype=float, order='C')
    for i, s1 in enumerate(sentences):
        for j, s2 in enumerate(sentences):
            correlation_matrix[i][j] = predict_next_sentence_prob(s1, s2)
    return correlation_matrix
```

Figure – 26. Correlation matrix function.

The reorder_sentences takes a string array-like. It creates a correlation matrix with the previous function and iteratively finds the best association in the matrix. It deletes the sentences selected in this way and continues until no association can be done. The association is expressed as the tuple (X, Y), where X is the first sentence and the Y is the most probable next sentence. The latter is used as the first sentence from which to search for the best next one.

```
def reorder_sentences(sentences: list):
    ordering = []
    correlation_matrix = create_correlation_matrix(sentences)
    hint = None
    while correlation_matrix.any():
        if hint == None:
            ind = np.unravel_index(np.argmax(correlation_matrix, axis=None),
                                   correlation_matrix.shape)
        else:
            ind = np.unravel_index(np.argmax(correlation_matrix[hint,:],
                                             axis=None),
                                   correlation_matrix[hint,:].shape)
            ind = (hint, ind[0])
        hint = ind[1]
        correlation_matrix[ind[0], :] = 0
        correlation_matrix[:, ind[0]] = 0
        ordering.append(ind[0])
    return ordering
```

Figure – 27. Reorder sentences function.

Future work.

In the final step we reached an autonomous program, which could be applied for current tasks and can be supplemented by the other approaches and methods. This approach was chosen thanks to the latest technological purposes. In the future work, there could be supplied a more accurate approach to multilingual and as addition could be implemented BART technology. In the human analysis the summary produced by the proposed approach is satisfied by the set of rules. The assessment through metrics is still in the process and will be reached.

References.

[1] Logan Lebanoff, Kaiqiang Song, Franck Dernoncourt, Doo Soon Kim, Seokhwan Kim, Walter Chang, and Fei Liu (2019). Scoring Sentence Singletons and Pairs for Abstractive Summarization.

[2] Yue Dong, Yikang Shen, Eric Crawford, Herke van Hoof, and Jackie C.K. Cheung (2019). BanditSum: Extractive Summarization as a Contextual Bandit.

[3] Shanghvan Bae, Taeuk Kim, Jihoon Kim, and Sang-goo Lee (2019). Summary Level Training of Sentence Rewriting for Abstractive Summarization.

[4] Simeng Han, Xiang Lin, and Shafiq Joty (2019). Resurrecting Submodularity in Neural Abstractive Summarization.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

[6] Gaetano Rossiello, Pierpaolo Basile, and Giovanni Semeraro (2017). Centroid-based Text Summarization through Compositionality of Word Embeddings.

[7] Derek Miller (2019). Leveraging BERT for Extractive Text Summarization on Lectures.

[8] Makbule Gulcin Ozsoy, Ferda Nur Alpaslan, and Ilyas Cicekli (2011). Journal of Information Science.

Text Summarization using Latent Semantic Analysis.

[9] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze (2008). Cambridge University Press. Introduction to Information Retrieval.

[10] Yang Liu (2019). Fine-tune BERT for Extractive Summarization.

[11] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N.Gomez, Łukasz Kaiser, and Illia Polosukhin (2017). Attention Is All You Need.

[12] Ashtoush Vashisht (https://iq.opengenus.org/bert-for-text-summarization/). Bert for text summarization.

[13] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limitiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Yun-Hsuan Sung, Brian Strope, Ray Kurzweil (2019). Universal Sentence Encoder.

[14] Neil Wu (2019). (https://medium.com/lsc-psd/a-bert-based-summarization-model-bertsum-88b1fc1b3177) . A Bert based summarization model “BertSum”.

[15] Horacio Saggion, Thierry Poibeau (2016). Automatic Text Summarization: Past, Present and Future.

[16] Yang Liu and Mirella Lapata (2019). Text Summarization with Pre-trained Encoders.

Code.

main.py

```
import os from src.bert_ext_summary import
continuous_bert_ext_summary from src.bert_ext_summary import
bert_ext_summary from  src.summary_reduction.sum_red import
symmary_reduction from src.reordering.reorder import reorder_sentences


if __name__ == "__main__":


   folder = '/Users/abylai/PycharmProjects/bertsum/resources/data/duc20024/'
      for each_dir in os.listdir(folder):
current_folder = (folder + each_dir)
for each_file in os.listdir(current_folder):
current_file = (current_folder + '/' +
each_file)
namefile = str(each_file)           namedir = str(each_dir)
f = open(current_file, "r", encoding="utf-8")           text
= f.read()
print(namedir, namefile)


        with open(f'/Users/abylai/PycharmProjects/bertsum/resources/data/duc20024/{namedir}/summary{namefile}.txt',
                        'w', encoding="utf-8") as file:
              max_num_sentences = 100
              max_num_words = 1300
              max_tokens_span = 500
              overlapping_sentences = 1
              long_summary = continuous_bert_ext_summary(text, max_tokens_span,
                                    overlapping_sentences,
                                    max_num_sentences,
                                    max_num_words)
                 text1 =
long_summary
             topic_summary =
             symmary_reduction(text1)
             goodorder
             =reorder_sentences(topic_summary)
             file.write(goodorder)
            file.close()
#f.close()
```

reorder.py

```
from transformers import BertTokenizer, TFBertForNextSentencePrediction
import tensorflow as tf import tensorflow.keras import numpy as np


pretrained_weights = 'bert-base-multilingual-cased' tokenizer =
BertTokenizer.from_pretrained(pretrained_weights) nsp_model =
TFBertForNextSentencePrediction.from_pretrained(pretrained_weights)



def reorder_sentences(sentences: list):     ordering = []
correlation_matrix = create_correlation_matrix(sentences)
hint = None    while correlation_matrix.any():       if hint
== None:
          ind = np.unravel_index(np.argmax(correlation_matrix, axis=None),
correlation_matrix.shape)         else:             ind =
np.unravel_index(np.argmax(correlation_matrix[hint,:], axis=None),
correlation_matrix[hint,:].shape)             ind = (hint, ind[0])       hint = ind[1]
correlation_matrix[ind[0], :] = 0         correlation_matrix[:, ind[0]] = 0       ordering.append(ind[0])


   reordered_sentences = [sentences[idx] for idx in ordering]
return reordered_sentences


def create_correlation_matrix(sentences: list):     num_sentences = len(sentences)
correlation_matrix = np.empty(shape=(num_sentences, num_sentences), dtype=float, order='C')
for i, s1 in enumerate(sentences):       for j, s2 in enumerate(sentences):
correlation_matrix[i][j] = predict_next_sentence_prob(s1, s2)
return correlation_matrix


def predict_next_sentence_prob(sent1, sent2):


   encoded = tokenizer.encode_plus(sent1, text_pair=sent2)     encoded["input_ids"] =
tf.constant(encoded["input_ids"])[None, :]    encoded["token_type_ids"]
= tf.constant(encoded["token_type_ids"])[None, :]    encoded["attention_mask"] =
tf.constant(encoded["attention_mask"])[None, :]


   outputs = nsp_model(encoded)
seq_relationship_scores = outputs[0]


   probs = tf.keras.activations.softmax(seq_relationship_scores, axis=-1)
return probs.numpy()[0][0]
```

sum_red.py

```
import numpy as np #import seaborn as sns import spacy import
tensorflow.compat.v1 as tf tf.disable_v2_behavior() import
tensorflow_hub as hub from tensorflow import keras import
tf_sentencepiece import ssl #import urllib #import certifi
ssl._create_default_https_context = ssl._create_unverified_context
#link = urllib.request.urlopen(module_url)
```

```
#%matplotlib inline module_url = 'https://tfhub.dev/google/universal-sentenceencoder-multilingual/1'
def symmary_reduction(text1):
      doc = [text1]
g = tf.Graph()
with g.as_default():
      text_input = tf.placeholder(dtype=tf.string, shape=[None])        xling_embed =
hub.Module(module_url)        embedded_text = xling_embed(text_input)        init_options =
tf.group([tf.compat.v1.global_variables_initializer(), tf.compat.v1.tables_initializer()])    g.finalize()


   session = tf.Session(graph=g)
session.run(init_options)


   #ranker = 'EmbedRank++'    rank_fn
= embedrankpp


   tokenizer = spacy.load('en_core_web_sm')
#tokenizer = spacy.load('en')
sents = [str(s).replace('\n', '') for s in
tokenizer(''.join(doc)).sents]
key_size = len(sents)/3    # Embedding
   doc_emb = session.run(embedded_text, feed_dict={text_input: doc})
sent_embs = session.run(embedded_text, feed_dict={text_input: sents})


   # Ranking     keys = rank_fn(doc_emb,
sent_embs, key_size)
      summary = []    for i, s
in enumerate(sents):        if
i in keys:
summary.append(s)        else:
pass       finalsum =
''.join(summary)     return
finalsum


def ncossim(embs_1, embs_2, axis=0):     sims = np.inner(embs_1, embs_2)     std
= np.std(sims, axis=axis)     ex = np.mean((sims-np.min(sims,
axis=axis))/np.max(sims, axis=axis),axis=axis)     return 0.5 + (sims-ex)/std



def mmr(doc_emb, cand_embs, key_embs):
   param = 0.5    scores = param * ncossim(cand_embs,   doc_emb, axis=0)    if key_embs is not None:
scores -= (1-param) * np.max(ncossim(cand_embs, key_embs), axis=1).reshape(scores.shape[0], -1)
return scores


def embedrankpp(doc_emb, sent_embs, n_keys):
   assert 0 < n_keys, 'Please `key_size` value set more than 0'    assert n_keys <
len(sent_embs), 'Please `key_size` value set lower than `#sentences`'    cand_idx =
list(range(len(sent_embs)))    key_idx = []    while len(key_idx) < n_keys:
cand_embs = sent_embs[cand_idx]        key_embs =
sent_embs[key_idx] if len(key_idx) > 0 else None        scores =
mmr(doc_emb, cand_embs, key_embs)
key_idx.append(cand_idx[np.argmax(scores)])
cand_idx.pop(np.argmax(scores))    return key_idx
```

bert_ext_summary.py

```
from nltk import word_tokenize, sent_tokenize
```

```
from src.preprocessing.text_splitting import text_split, \
change_split_parameters from src.processing.ExtractiveTextSummarizer import
ExtractiveTextSummarizer



def bert_ext_summary(text, max_tokens_span=500, overlapping_sentences=1):
data = text_split(text, max_tokens_span, overlapping_sentences)
ext_summarizer = ExtractiveTextSummarizer()


  long_summary = ''
results = {
"summaries": []    }
for text_part in data:
      result = ext_summarizer.predict([text_part["text"]])
results["summaries"].append(result["summaries"][0][0])
long_summary += ' ' + result["summaries"][0][0]    long_summary
= long_summary.replace('<q>', ' ')    return
long_summary



def continuous_bert_ext_summary(text, max_tokens_span,
                    overlapping_sentences,
max_num_sentences=100,                          max_num_words=1300):
  summary_history = dict()
counter = 0    do_summary =
True    long_summary = text
while do_summary:
      long_summary = bert_ext_summary(long_summary, max_tokens_span,
                          overlapping_sentences)
sentences = sent_tokenize(long_summary)        words
= word_tokenize(long_summary)
summary_history[counter] = {
"num_sentences": len(sentences),
         "num_words": len(words)
      }          print("iteration #",
counter,
           " --- max_tokens_span:", max_tokens_span,
           ", overlapping_sentences:", overlapping_sentences,
           ", sentences:", len(sentences),             ", words:", len(words))
print("summary:",       long_summary)                                    if
summary_history[counter]["num_sentences"] < max_num_sentences or \
summary_history[counter]["num_words"]    <              max_num_words:
do_summary = False
if counter > 0:
        if    summary_history[counter           -              1]["num_sentences"]  <
            \                summary_history[counter]["num_sentences"] or \
summary_history[counter - 1]["num_words"] < \
summary_history[counter]["num_words"]:
change_split_parameters(max_tokens_span, overlapping_sentences)
counter += 1    return long_summary
```

TextSummarizer.py

```
from abc import abstractmethod  from
src.preprocessing.Preprocessing import Preprocessing


class TextSummarizer:    def
__init__(self):       self.preprocessing
= Preprocessing()     @abstractmethod
def predict(self,data):          pass
```

ExtractiveTextSummarizer.py

```
import torch from src.data.DataLoader import Dataloader from
src.data.DataLoader import dataset_loader from src.general.Singleton
```

```
import Singleton from src.processing.TextSummarizer import
TextSummarizer from src.processing.predictor.ExtPredictor import
ExtPredictor


class ExtractiveTextSummarizer(TextSummarizer, metaclass=Singleton):
def __init__(self):
super().__init__()        self.predictor
= ExtPredictor()
def predict(self,
data):
      processed_data      =      self.preprocessing.preprocessing(data)
data_iterator = Dataloader(datasets=dataset_loader(processed_data),
batch_size=64, device=torch.device('cpu'),
shuffle=False, is_test=True)        summaries =
self.predictor.predict(data_iterator)        response = {"summaries": [[s] for
s in summaries]}        return response
```

AbstractiveTextSummarizer.py

```
import torch
from src.data.DataLoader import Dataloader from
src.data.DataLoader import dataset_loader from src.general.Singleton
import Singleton from
src.processing.predictor.AbsPredictor import AbsPredictor from
src.processing import TextSummarizer


class AbstractiveTextSummarizer(TextSummarizer,
metaclass=Singleton):        def __init__(self):
super().__init__()        self.predictor
= AbsPredictor()
    def predict(self, data):        processed_data =
self.preprocessing.preprocessing(data)        data_iterator =
Dataloader(datasets=dataset_loader(processed_data),
batch_size=64, device=torch.device('cpu'),
shuffle=False, is_test=True)        summaries =
self.predictor.predict(data_iterator)        response = {"summaries": [[s] for
s in summaries]}        return response
```

ExtractiveTranslator.py

```
import numpy as np import
torch


class ExtractiveTraslator(object):
def __init__(self, args, model,
optim,
grad_accum_count=1, n_gpu=1, gpu_rank=1):        #
Basic attributes.        self.args = args
self.max_length=self.args.max_length        self.model =
model        self.optim = optim
self.grad_accum_count = grad_accum_count
self.n_gpu = n_gpu        self.gpu_rank =
gpu_rank


      self.loss = torch.nn.BCELoss(reduction='none')
assert grad_accum_count > 0
```

```
   def translate_batch(self, batch,
fast=False):          with torch.no_grad():
return self._fast_translate_batch(
batch)
     def
_fast_translate_batch(self,
data_iterator):


      for batch in data_iterator:               src, segs, clss, mask, mask_cls = batch.src, batch.segs, batch.clss,
batch.mask_src, batch.mask_cls             pred = []
          sent_scores, mask = self.model(src, segs, clss, mask, mask_cls)
sent_scores = sent_scores + mask.float()           sent_scores =
sent_scores.cpu().data.numpy()           selected_ids =
np.argsort(sent_scores, 1)           for i, idx in enumerate(selected_ids):
              _pred = []               if
(len(batch.src_str[i]) == 0):
                  continue               for j in
selected_ids[i][:len(batch.src_str[i])]:                   if (j >=
len(batch.src_str[i])):
                      continue                    candidate =
batch.src_str[i][j].strip()                   if
(self.args.block_trigram):                        if (not
_block_tri(candidate, _pred)):
_pred.append(candidate)                   else:
                      _pred.append(candidate)
_pred = '<q>'.join(_pred)
pred.append(_pred)       return pred


def _get_ngrams(n, text):    ngram_set = set()
text_length = len(text)
max_index_ngram_start = text_length - n    for i
in range(max_index_ngram_start + 1):
ngram_set.add(tuple(text[i:i + n]))
return ngram_set


   def _block_tri(c, p):
tri_c = _get_ngrams(3,
c.split())     for s in p:
      tri_s = _get_ngrams(3,
s.split())        if
len(tri_c.intersection(tri_s)) > 0:
return True     return False


def build_trainer(args, device_id, model, optim):


  grad_accum_count = args.accum_count
n_gpu = args.world_size
     if device_id >=
0:
      gpu_rank = int(args.gpu_ranks[device_id])     else:         gpu_rank = 0
n_gpu = 0    trainer = ExtractiveTraslator(args, model, optim,   grad_accum_count,
n_gpu, gpu_rank)    return trainer
```

AbstractiveTranslator.py

```
import torch


class AbstractiveTranslator(object):

def
__init__(self,
args,                model,
vocab,               symbols,
global_scorer=None,
logger=None,
dump_beam=""):       self.logger
= logger
      #self.cuda = args.visible_gpus != '-
1'           self.args = args       self.model
= model       self.generator =
self.model.generator       self.vocab =
vocab       self.symbols = symbols
self.start_token = symbols['BOS']
self.end_token = symbols['EOS']


      self.global_scorer                =
            global_scorer       self.beam_size =
args.beam_size       self.min_length = args.min_length
self.max_length =
args.max_length


      self.dump_beam = dump_beam


      self.beam_trace = self.dump_beam != ""
self.beam_accum = None
       if self.beam_trace:
self.beam_accum = {
"predicted_ids": [],
             "beam_parent_ids": [],
             "scores": [],
             "log_probs": []}
      def from_batch(self, translation_batch):
batch = translation_batch["batch"]
assert (len(translation_batch["gold_score"])
==
len(translation_batch["predictions"]))
batch_size = batch.batch_size
      preds, pred_score, gold_score,src = translation_batch["predictions"],
translation_batch["scores"],translation_batch["gold_score"], batch.src
translations = []       for b in range(batch_size):
          pred_sents = self.vocab.convert_ids_to_tokens([int(n) for n in preds[b][0]])
pred_sents  =  ' '.join(pred_sents).replace(' ##','')                raw_src  =
[self.vocab.ids_to_tokens[int(t)] for t in src[b]][:500]          raw_src = '
'.join(raw_src)                          translation  =  (pred_sents,  raw_src)
translations.append(translation)
        return
translations     def
translate_batch(self,
batch, fast=False):
          with
torch.no_grad():
          return self._fast_translate_batch(
batch,              self.max_length,
min_length=self.min_length)


   def _fast_translate_batch(self,
                        batch,
max_length,                          min_length=0):
      # TODO: faster code path for beam_size == 1.
```

```
        # TODO: support these blacklisted
features.
assert not self.dump_beam
        beam_size  =  self.beam_size
batch_size = batch.batch_size      src
= batch.src          segs = batch.segs
mask_src = batch.mask_src


      src_features = self.model.bert(src, segs, mask_src)        dec_states =
self.model.decoder.init_decoder_state(src, src_features,
with_cache=True)        device = src_features.device
        # Tile states and memory beam_size
times.        dec_states.map_batch_fn(
          lambda state, dim: tile(state, beam_size, dim=dim))
src_features = tile(src_features, beam_size, dim=0)
batch_offset = torch.arange(          batch_size, dtype=torch.long,
device=device)      beam_offset = torch.arange(
          0,          batch_size *
beam_size,
step=beam_size,
dtype=torch.long,          device=device)
alive_seq = torch.full(          [batch_size
*          beam_size, 1],          self.start_token,
dtype=torch.long,          device=device)


      # Give full probability to the first beam on the first step.
topk_log_probs = (          torch.tensor([0.0] + [float("-inf")]
*          (beam_size - 1),
device=device).repeat(batch_size))
        # Structure that holds finished hypotheses.
hypotheses = [[] for _ in range(batch_size)]  # noqa:
F812          results = {}      results["predictions"] = [[]
for _ in
range(batch_size)]  # noqa: F812      results["scores"] = [[] for _ in
range(batch_size)]  # noqa: F812      results["gold_score"] = [0] *
batch_size      results["batch"] = batch
        for step in range(max_length):
decoder_input = alive_seq[:, -1].view(1, -
1)


        # Decoder forward.          decoder_input =
decoder_input.transpose(0,1)
          dec_out, dec_states = self.model.decoder(decoder_input, src_features,
dec_states,
step=step)


        # Generator forward.          log_probs =
self.generator.forward(dec_out.transpose(0,1).squeeze(0)
)          vocab_size = log_probs.size(-1)          if
step < min_length:          log_probs[:, self.end_token]
= -1e20


        # Multiply probs by the beam probability.
log_probs += topk_log_probs.view(-1).unsqueeze(1)
          alpha = self.global_scorer.alpha
length_penalty = ((5.0 + (step + 1)) / 6.0) ** alpha
          # Flatten probs into a list of
possibilities.
curr_scores = log_probs / length_penalty


if(self.args.block_trigram):          cur_len = alive_seq.size(1)
if(cur_len>3):          for i in range(alive_seq.size(0)):          fail =
False          words = [int(w) for w in  alive_seq[i]]          words
= [self.vocab.ids_to_tokens[w] for w in words]          words = '
'.join(words).replace(' ##','').split()          if(len(words)<=3):
```

```
continue                    trigrams = [(words[i-1],words[i],words[i+1]) for i in
range(1,len(words)-1)]                    trigram = tuple(trigrams[-1])                    if
trigram in trigrams[:-1]:
                    fail = True
if fail:                        curr_scores[i] = -
10e20


        curr_scores = curr_scores.reshape(-1, beam_size * vocab_size)
topk_scores, topk_ids = curr_scores.topk(beam_size, dim=-1)


        # Recover log probs.          topk_log_probs =
topk_scores * length_penalty


        # Resolve beam origin and true word ids.
topk_beam_index = topk_ids.div(vocab_size)          topk_ids =
topk_ids.fmod(vocab_size)


        # Map beam_index to batch_index in the flat
representation.          batch_index = (
topk_beam_index
                + beam_offset[:topk_beam_index.size(0)].unsqueeze(1))
select_indices = batch_index.view(-1)
          # Append last
prediction.          alive_seq =
torch.cat(
            [alive_seq.index_select(0, select_indices),
topk_ids.view(-1, 1)], -1)
is_finished =
topk_ids.eq(self.end_token)
if step + 1 == max_length:              is_finished.fill_(1)              # End
condition is top beam is finished.          end_condition = is_finished[:,
0].eq(1)          # Save finished hypotheses.              if is_finished.any():
predictions = alive_seq.view(-1, beam_size, alive_seq.size(-1))
for i in range(is_finished.size(0)):
                b = batch_offset[i]                  if
end_condition[i]:                    is_finished[i].fill_(1)
finished_hyp = is_finished[i].nonzero().view(-1)
# Store finished hypotheses for this batch.                  for j in
finished_hyp:                    hypotheses[b].append((
topk_scores[i, j],                        predictions[i, j, 1:]))
                # If the batch reached the end, save the n_best hypotheses.
if end_condition[i]:                    best_hyp = sorted(
hypotheses[b], key=lambda x: x[0], reverse=True)
score, pred = best_hyp[0]


results["scores"][b].append(score)
results["predictions"][b].append(pred)              non_finished =
end_condition.eq(0).nonzero().view(-1)              # If all sentences
are translated, no need to go further.              if len(non_finished)
== 0:
                break
            # Remove finished batches for the next step.
topk_log_probs = topk_log_probs.index_select(0, non_finished)
batch_index = batch_index.index_select(0, non_finished)
batch_offset = batch_offset.index_select(0, non_finished)
alive_seq = predictions.index_select(0, non_finished) \
.view(-1, alive_seq.size(-1))          # Reorder states.
select_indices = batch_index.view(-1)          src_features =
src_features.index_select(0, select_indices)
dec_states.map_batch_fn(              lambda state, dim:
state.index_select(dim, select_indices))
        return
results


def tile(x, count, dim=0):
"""
```

```
  Tiles x on dimension dim count times.
  """
  perm = list(range(len(x.size())))     if dim != 0:
perm[0], perm[dim] = perm[dim], perm[0]         x =
x.permute(perm).contiguous()    out_size = list(x.size())
out_size[0] *= count    batch =
x.size(0)    x = x.view(batch, -1) \
.transpose(0, 1) \
       .repeat(count, 1) \
       .transpose(0, 1) \
.contiguous() \        .view(*out_size)    if
dim != 0:       x =
x.permute(perm).contiguous()    return
x
```

ExtPredictor.py

```
import torch from src.models.ExtSummarizer import ExtSummarizer
from src.processing.predictor.Predictor import Predictor from
src.processing.traslator.ExtractiveTraslator import build_trainer



class ExtPredictor(Predictor):
    def __init__(self):       super().__init__()       self.device
=  torch.device('cpu')       self.model =
torch.load('./resources/models/summarizer_extractive.pt',
map_location=self.device)       self.model_ext =
ExtSummarizer(self.args, self.device, self.model)
self.model_ext.eval()       self.trainer = self.build_predictor()
     def build_predictor(self):          trainer =
build_trainer(self.args, -1, self.model_ext, None)         return
trainer        def predict(self,data_iterator):       with
torch.no_grad():
         return self.trainer.translate_batch(data_iterator)
```

Penalties.py

```
class PenaltyBuilder(object):
    def      __init__(self,
length_pen):       self.length_pen
= length_pen        def
length_penalty(self):       if
self.length_pen == "wu":
return self.length_wu       elif
self.length_pen == "avg":
return self.length_average
else:            return
self.length_none     def
length_wu(self, beam, logprobs,
alpha=0.):

     modifier = (((5 + len(beam.next_ys)) ** alpha) /
              ((5 + 1) ** alpha))       return
(logprobs / modifier)

  def length_average(self, beam, logprobs, alpha=0.):

     return logprobs / len(beam.next_ys)
```

```
  def length_none(self, beam, logprobs, alpha=0., beta=0.):
      return logprobs
```

text_splitting.py

```
# -*- coding: ISO-8859-1 -*- import nltk    from nltk import
word_tokenize, sent_tokenize nltk.download('punkt')


def change_split_parameters(max_tokens_span, overlapping_sentences):
   max_tokens_span += 20
overlapping_sentences -= 1    if max_tokens_span >
512:        max_tokens_span = 512    if
overlapping_sentences < 0:
overlapping_sentences = 0    return
max_tokens_span, overlapping_sentences


def
initialize_text_part(sentence_data_list=()):
text_part = {        "num_tokens": 0,
      "num_sentences": 0,
      "text": "
   }
   text_part = aggregate_text_part(text_part, sentence_data_list)
return text_part


def aggregate_text_part(text_part, sentence_data_list):
for sentence_data in sentence_data_list:
text_part["num_tokens"] += sentence_data["num_tokens"]
text_part["num_sentences"] += 1        text_part["text"] += '
' + sentence_data["text"]    return text_part


def text_split(text, max_tokens_span=400, overlapping_sentences=1):
   sentences = sent_tokenize(text)
sentence_data_map = dict()    for idx, sentence
in enumerate(sentences):
sentence_data_map[idx] = {
         "num_tokens": len(word_tokenize(sentence)),
         "text": sentence,
      }        text_parts = []    text_part = initialize_text_part()    for idx,
sentence_data    in    sentence_data_map.items():                    if
text_part["num_tokens"]                <                max_tokens_span:
aggregate_text_part(text_part,    [sentence_data])                    else:
text_parts.append(text_part)            sentence_data_list = []            if
idx  >  overlapping_sentences:                        for  i  in  range(1,
overlapping_sentences):
sentence_data_list.append(sentence_data_map[idx            -            i])
text_part = initialize_text_part(sentence_data_list)    return text_parts
```

preprocessing.py

```
from nltk.tokenize import word_tokenize, sent_tokenize from
src.preprocessing.BertTransformation import BertTransformation


class Preprocessing():
      def __init__(self):
self.bertTransformation = BertTransformation()
```

```
    def tokenize(self, stories):        stories_list =
list()        for story in stories:            sentences =
sent_tokenize(story)            src = list()            for
sentence in sentences:                str_tokens =
word_tokenize(sentence)
src.append(str_tokens)            data = dict()
data['src'] = src            stories_list.append(data)
return stories_list         def preprocessing(self,
stories):        stories_list
=
self.tokenize(stories)        processed_data =
self.bertTransformation.to_bert(stories_list)        return processed_data
```

BertTransformation.py

```
from argparse import Namespace   from
src.preprocessing.BertData import BertData


args = Namespace(lower=True, use_bert_basic_tokenizer=True,
min_src_ntokens_per_sent=5, max_src_ntokens_per_sent=200,
max_src_nsents=100, max_tgt_ntokens=500)


class BertTransformation:
      def __init__(self):
self.args
= args         def
to_bert(self, json_data):
bert =
BertData(self.args)
datasets = []        for d
in json_data:
          source, tgt = d['src'], ''
sent_labels =
''''
if self.args.lower:                    source = ['
'.join(s).lower().split() for s in source]                tgt = ['
'.join(s).lower().split() for s in tgt]            b_data =
bert.preprocess(source, tgt, sent_labels,
use_bert_basic_tokenizer=False,
                                  is_test=True)            if b_data is None:                    continue
src_subtoken_idxs, sent_labels, tgt_subtoken_idxs, segments_ids, cls_ids, src_txt, tgt_txt = b_data
b_data_dict = {"src": src_subtoken_idxs, "tgt": tgt_subtoken_idxs,
                        "src_sent_labels": sent_labels, "segs": segments_ids,
'clss': cls_ids,
                        'src_txt': src_txt, "tgt_txt": tgt_txt}
datasets.append(b_data_dict)        return datasets
```

BertData.py

```
from src.preprocessing.tokenizer.BertTokenizer import BertTokenizer



class BertData(object):
def __init__(self,args):
      self.args=args
      self.tokenizer = BertTokenizer.from_pretrained('bert-base-uncased',
                                          do_lower_case=True)
self.sep_token =
'[SEP]'
self.cls_token = '[CLS]'
self.pad_token = '[PAD]'        self.tgt_bos
= '[unused0]'        self.tgt_eos =
'[unused1]'        self.tgt_sent_split =
```

```
'[unused2]'         self.sep_vid =
self.tokenizer.vocab[self.sep_token]
self.cls_vid = self.tokenizer.vocab[self.cls_token]        self.pad_vid =
self.tokenizer.vocab[self.pad_token]


   def preprocess(self, src, tgt, sent_labels, use_bert_basic_tokenizer=False, is_test=False):
        if ((not is_test) and len(src) ==
0):             return None
original_src_txt = [' '.join(s) for s in
src]


       idxs = [i for i, s in enumerate(src) if (len(s) > self.args.min_src_ntokens_per_sent)]


       _sent_labels = [0] * len(src)
for l in sent_labels:
           _sent_labels[l] = 1
           src = [src[i][:self.args.max_src_ntokens_per_sent] for i in
idxs]         sent_labels = [_sent_labels[i] for i in idxs]         src =
src[:self.args.max_src_nsents]         sent_labels =
sent_labels[:self.args.max_src_nsents]
          if ((not is_test) and len(src) <
self.args.min_src_nsents):
return None
         src_txt = [' '.join(sent) for sent in src]         text = ' {}
{} '.format(self.sep_token, self.cls_token).join(src_txt)


       src_subtokens = self.tokenizer.tokenize(text)


       src_subtokens = [self.cls_token] + src_subtokens + [self.sep_token]
src_subtoken_idxs      =                self.tokenizer.convert_tokens_to_ids(src_subtokens)
_segs = [-1] + [i for i, t in enumerate(src_subtoken_idxs) if t == self.sep_vid]         segs =
[_segs[i] - _segs[i - 1] for i in range(1, len(_segs))]         segments_ids = []         for i, s in
enumerate(segs):             if (i % 2 == 0):                  segments_ids += s *
[0]
else:
                segments_ids += s * [1]         cls_ids = [i for i, t in
enumerate(src_subtoken_idxs) if t == self.cls_vid]         sent_labels =
sent_labels[:len(cls_ids)]


       tgt_subtokens_str = '[unused0] ' + ' [unused2] '.join(
           [' '.join(self.tokenizer.tokenize(' '.join(tt), use_bert_basic_tokenizer=use_bert_basic_tokenizer)) for tt in tgt]) + '
[unused1]'                                                    tgt_subtoken         =
tgt_subtokens_str.split()[:self.args.max_tgt_ntokens]         if ((not is_test)
and len(tgt_subtoken) < self.args.min_tgt_ntokens):               return None


       tgt_subtoken_idxs =
self.tokenizer.convert_tokens_to_ids(tgt_subtoken)              tgt_txt =
'<q>'.join([' '.join(tt) for tt in tgt])
src_txt = [original_src_txt[i] for i in idxs]


       return src_subtoken_idxs, sent_labels, tgt_subtoken_idxs, segments_ids, cls_ids, src_txt, tgt_txt
```

Tokenizer.py

```
import unicodedata
  class       BasicTokenizer(object):
              def           __init__(self,               do_lower_case=True,
never_split=("[UNK]", "[SEP]", "[PAD]", "[CLS]", "[MASK]")):


      self.do_lower_case =
do_lower_case        self.never_split =
never_split        def tokenize(self, text):
text = self._clean_text(text)
text = self._tokenize_chinese_chars(text)       orig_tokens =
whitespace_tokenize(text)       split_tokens = []       for
i,token in enumerate(orig_tokens):           if
self.do_lower_case and token not in self.never_split:
token = token.lower()               token =
self._run_strip_accents(token)           split_tokens.extend([(i,t) for t
in  self._run_split_on_punc(token)])           return split_tokens
      def _run_strip_accents(self, text):       text
= unicodedata.normalize("NFD", text)
output = []       for char in text:
          cat = unicodedata.category(char)
if cat == "Mn":               continue
output.append(char)       return "".join(output)
      def _run_split_on_punc(self,
text):        if text in self.never_split:
return [text]        chars = list(text)
i
= 0        start_new_word =
True        output = []       while i <
len(chars):           char = chars[i]
if _is_punctuation(char):
output.append([char])
start_new_word = True           else:
if start_new_word:
output.append([])
start_new_word = False
output[-1].append(char)
          i += 1        return ["".join(x)
for x in output]


   def _tokenize_chinese_chars(self, text):
      output = []       for char in
text:           cp = ord(char)           if
self._is_chinese_char(cp):
output.append(" ")
output.append(char)               output.append(" ")
else:               output.append(char)
return "".join(output)


   def _is_chinese_char(self, cp):
         if ((cp >= 0x4E00 and cp <= 0x9FFF) or  #
(cp >= 0x3400 and cp <= 0x4DBF) or  #
            (cp >= 0x20000 and cp <= 0x2A6DF) or  #
            (cp >= 0x2A700 and cp <= 0x2B73F) or  #
            (cp >= 0x2B740 and cp <= 0x2B81F) or  #
            (cp >= 0x2B820 and cp <= 0x2CEAF) or
            (cp >= 0xF900 and cp <= 0xFAFF) or  #
(cp >= 0x2F800 and cp <= 0x2FA1F)):  #           return
True
        return
False
     def _clean_text(self, text):       output = []       for
char in text:           cp = ord(char)           if cp == 0 or
cp == 0xfffd or _is_control(char):               continue
if
```

```
_is_whitespace(char):                output.append("
")           else:                output.append(char)
return "".join(output)


class WordpieceTokenizer(object):


   def __init__(self, vocab, unk_token="[UNK]", max_input_chars_per_word=100):
      self.vocab = vocab        self.unk_token = unk_token
self.max_input_chars_per_word =
max_input_chars_per_word         def tokenize(self, text):
output_tokens = []        for token in whitespace_tokenize(text):
          chars = list(token)           if len(chars) >
self.max_input_chars_per_word:
              output_tokens.append(self.unk_token)
continue
            is_bad =
False
start = 0            sub_tokens = []
while start < len(chars):
end = len(chars)                cur_substr =
None                while start < end:
substr = "".join(chars[start:end])
if start > 0:
                      substr = "##" + substr
if substr in self.vocab:
cur_substr = substr                        break
end -= 1                if cur_substr is
None:                    is_bad = True
break
sub_tokens.append(cur_substr)
start = end               if is_bad:
output_tokens.append(self.unk_token)
else:
output_tokens.extend(sub_tokens)
return output_tokens


def _is_whitespace(char):    if char == " " or char == "\t" or
char == "\n" or char == "\r":          return True    cat =
unicodedata.category(char)    if cat == "Zs":        return
True    return False


def _is_control(char):    if char == "\t" or char
== "\n" or char == "\r":          return False
cat = unicodedata.category(char)    if
cat.startswith("C"):        return True    return
False


def _is_punctuation(char):    cp = ord(char)    if ((cp >= 33 and
cp <= 47) or (cp >= 58 and cp <= 64) or            (cp >= 91 and cp
<= 96) or (cp >= 123 and cp <= 126)):
       return True    cat =
unicodedata.category(char)    if
cat.startswith("P"):        return True
return False

def
whitespace_tokenize(text):
text = text.strip()      if not
text:        return
```

```
[]    tokens = text.split()
return tokens
```

BertTokenizer.py

```
import collections import
os

from src.preprocessing.tokenizer.Tokenizer import BasicTokenizer, WordpieceTokenizer from
pytorch_transformers import cached_path

PRETRAINED_VOCAB_ARCHIVE_MAP = {
   'bert-base-uncased': "https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-uncased-vocab.txt",
   'bert-large-uncased': "https://s3.amazonaws.com/models.huggingface.co/bert/bert-large-uncased-vocab.txt",      'bert-base-cased':
"https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-cased-vocab.txt",                                  'bert-largecased':
"https://s3.amazonaws.com/models.huggingface.co/bert/bert-large-cased-vocab.txt",                    'bert-base-multilingualuncased':
"https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-multilingual-uncased-vocab.txt",   'bert-base-multilingualcased':
"https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-multilingual-cased-vocab.txt",
   'bert-base-chinese': "https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-chinese-vocab.txt", }
PRETRAINED_VOCAB_POSITIONAL_EMBEDDINGS_SIZE_MAP = {
   'bert-base-uncased': 512,
   'bert-large-uncased': 512,
   'bert-base-cased': 512,
   'bert-large-cased': 512,
   'bert-base-multilingual-uncased': 512,
   'bert-base-multilingual-cased': 512,
   'bert-base-chinese': 512,
}
VOCAB_NAME = 'vocab.txt'



class BertTokenizer(object):
   """Runs end-to-end tokenization: punctuation splitting + wordpiece"""


   def __init__(self, vocab_file, do_lower_case=True, max_len=None,
never_split=(
               "[UNK]", "[SEP]", "[PAD]", "[CLS]", "[MASK]", "[unused0]",
"[unused1]", "[unused2]", "[unused3]", "[unused4]",                "[unused5]",
"[unused6]")):            if              not os.path.isfile(vocab_file):             raise ValueError(
            "Can't find a vocabulary file at path '{}'. To load the vocabulary from a Google pretrained "
"model use `tokenizer = BertTokenizer.from_pretrained(PRETRAINED_MODEL_NAME)`".format(
vocab_file))          self.do_lower_case = do_lower_case        self.vocab =   load_vocab(vocab_file)       self.ids_to_tokens =
collections.OrderedDict(
[(ids, tok) for tok, ids in self.vocab.items()])       self.basic_tokenizer =
BasicTokenizer(do_lower_case=do_lower_case,
                                      never_split=never_split)
     self.wordpiece_tokenizer = WordpieceTokenizer(vocab=self.vocab)
self.max_len = max_len if max_len is not None else int(1e12)
```

```
    def tokenize(self,
text,
use_bert_basic_tokenizer=False):        split_tokens =
[]      if (use_bert_basic_tokenizer):              pretokens
= self.basic_tokenizer.tokenize(text)       else:
        pretokens =
list(enumerate(text.split()))           for i, token
in pretokens:
        # if(self.do_lower_case):           #     token =
token.lower()          subtokens =
self.wordpiece_tokenizer.tokenize(token)          for
sub_token in subtokens:
split_tokens.append(sub_token)
return split_tokens


  def convert_tokens_to_ids(self, tokens):
    """Converts a sequence of tokens into ids using the
vocab."""          ids = []       for token in tokens:
ids.append(self.vocab[token])       # if len(ids) > self.max_len:
    #     raise ValueError(
    #         "Token indices sequence length is longer than the specified maximum "
    #         " sequence length for this BERT model ({} > {}). Running
this"
    #         " sequence through BERT will result in indexing errors".format(len(ids), self.max_len)
    #     )       return
ids


  def convert_ids_to_tokens(self, ids):
    """Converts a sequence of ids in wordpiece tokens using the vocab."""          tokens
= []
for i in ids:
tokens.append(self.ids_to_tokens[i])
return tokens


  @classmethod    def from_pretrained(cls, pretrained_model_name_or_path,
cache_dir=None,
                  *inputs, **kwargs):
"""
    Instantiate a PreTrainedBertModel from a pre-trained model file.
    Download and cache the pre-trained model file if needed.
    """
    if pretrained_model_name_or_path in PRETRAINED_VOCAB_ARCHIVE_MAP:
vocab_file = PRETRAINED_VOCAB_ARCHIVE_MAP[
pretrained_model_name_or_path]
else:           vocab_file =
pretrained_model_name_or_path
if os.path.isdir(vocab_file):           vocab_file =
os.path.join(vocab_file, VOCAB_NAME)
    # redirect to the cache, if necessary       try:            resolved_vocab_file =
cached_path(vocab_file, cache_dir=cache_dir)       except EnvironmentError:            return None
if pretrained_model_name_or_path in
PRETRAINED_VOCAB_POSITIONAL_EMBEDDINGS_SIZE_MAP:
        # if we're using a pretrained model, ensure the tokenizer wont index sequences longer
        # than the number of positional embeddings
        max_len = PRETRAINED_VOCAB_POSITIONAL_EMBEDDINGS_SIZE_MAP[
pretrained_model_name_or_path]          kwargs['max_len'] = min(kwargs.get('max_len', int(1e12)),
max_len)       # Instantiate tokenizer.
    tokenizer = cls(resolved_vocab_file, *inputs, **kwargs)
return tokenizer



def load_vocab(vocab_file):
```

```
    """Loads a vocabulary file into a dictionary."""
vocab = collections.OrderedDict()    index = 0
with open(vocab_file, "r", encoding="utf-8") as
reader:        while True:                token =
reader.readline()             if not token:                break
token = token.strip()            vocab[token] = index
index += 1    return vocab
```

optimizer.py

```
import torch import torch.optim as optim
from torch.nn.utils import clip_grad_norm_


def use_gpu(opt):
   return (hasattr(opt, 'gpu_ranks') and len(opt.gpu_ranks) > 0) or \
(hasattr(opt, 'gpu') and opt.gpu > -1)


def build_optim(model, opt, checkpoint):
saved_optimizer_state_dict = None
      if opt.train_from:         optim = checkpoint['optim']
saved_optimizer_state_dict =
optim.optimizer.state_dict()      else:
      optim = Optimizer(            opt.optim, opt.learning_rate,
opt.max_grad_norm,
lr_decay=opt.learning_rate_decay,
start_decay_steps=opt.start_decay_steps,
decay_steps=opt.decay_steps,           beta1=opt.adam_beta1,
beta2=opt.adam_beta2,
adagrad_accum=opt.adagrad_accumulator_init,
decay_method=opt.decay_method,
warmup_steps=opt.warmup_steps)


   optim.set_parameters(model.named_parameters())
      if
opt.train_from:


optim.optimizer.load_state_dict(saved_optimizer_state_dict)
if use_gpu(opt):           for state in
optim.optimizer.state.values():                for k, v in state.items():
if torch.is_tensor(v):
state[k] = v.cuda()
         if (optim.method == 'adam') and (len(optim.optimizer.state) <
1):
raise RuntimeError(
             "Error: loaded Adam optimizer from existing model" +
" but optimizer state is empty")
      return
optim



class MultipleOptimizer(object):
      def __init__(self,
op):
self.optimizers = op
      def zero_grad(self):
for op in
self.optimizers:
op.zero_grad()
     def step(self):       for
op in  self.optimizers:
op.step()        @property
def state(self):
```

```
    return {k: v for op in self.optimizers for k, v in
op.state.items()}     def state_dict(self):
    return [op.state_dict() for op in self.optimizers]
def load_state_dict(self, state_dicts):     assert
len(state_dicts) == len(self.optimizers)     for i in
range(len(state_dicts)):
self.optimizers[i].load_state_dict(state_dicts[i])



class Optimizer(object):


  def __init__(self, method, learning_rate, max_grad_norm,
lr_decay=1, start_decay_steps=None, decay_steps=None,
beta1=0.9, beta2=0.999,              adagrad_accum=0.0,
decay_method=None,              warmup_steps=4000,
weight_decay=0):         self.last_ppl = None         self.learning_rate =
learning_rate
self.original_lr = learning_rate
self.max_grad_norm = max_grad_norm
self.method = method       self.lr_decay =
lr_decay       self.start_decay_steps =
start_decay_steps       self.decay_steps =
decay_steps       self.start_decay = False
self._step = 0       self.betas = [beta1, beta2]
self.adagrad_accum = adagrad_accum
self.decay_method = decay_method
self.warmup_steps = warmup_steps       self.weight_decay
= weight_decay
    def set_parameters(self,
params):         self.params = []
self.sparse_params = []       for k,
p in params:          if
p.requires_grad:
            if self.method != 'sparseadam' or "embed" not in k:
                self.params.append(p)
else:
self.sparse_params.append(p)         if
self.method == 'sgd':             self.optimizer =
optim.SGD(self.params, lr=self.learning_rate)
elif self.method == 'adagrad':
        self.optimizer = optim.Adagrad(self.params, lr=self.learning_rate)
for group in self.optimizer.param_groups:              for p in
group['params']:
                self.optimizer.state[p]['sum'] = self.optimizer\
.state[p]['sum'].fill_(self.adagrad_accum)       elif self.method == 'adadelta':             self.optimizer = optim.Adadelta(self.params,
lr=self.learning_rate)       elif self.method == 'adam':
        self.optimizer = optim.Adam(self.params, lr=self.learning_rate,
betas=self.betas, eps=1e-9)       else:            raise
RuntimeError("Invalid optim method: " + self.method)
def _set_rate(self, learning_rate):       self.learning_rate =
learning_rate       if self.method != 'sparseadam':
self.optimizer.param_groups[0]['lr'] = self.learning_rate
else:          for op in self.optimizer.optimizers:
            op.param_groups[0]['lr'] = self.learning_rate
def step(self):       self._step +=
1
if self.decay_method == "noam":          self._set_rate(
self.original_lr *              min(self._step
** (-0.5),                    self._step *
self.warmup_steps**(-1.5)))


else:          if ((self.start_decay_steps is
not None) and (                    self._step >=
self.start_decay_steps)):
self.start_decay = True            if self.start_decay:
if ((self._step - self.start_decay_steps)                %
self.decay_steps == 0):                  self.learning_rate =
```

```
self.learning_rate * self.lr_decay
        if self.method != 'sparseadam':
self.optimizer.param_groups[0]['lr'] =
self.learning_rate            if self.max_grad_norm:
clip_grad_norm_(self.params, self.max_grad_norm)
self.optimizer.step()
```

ExtSummarizer.py

```
import torch import torch.nn as nn from src.models.Bert import Bert from
src.models.submodels.Encoder import ExtTransformerEncoder, Classifier from
pytorch_transformers import BertConfig from pytorch_transformers import BertModel



class ExtSummarizer(nn.Module):     def
__init__(self, args, device, checkpoint):
super(ExtSummarizer, self).__init__()       self.args
= args       self.device = device
      self.bert = Bert(args.large, args.temp_dir, args.finetune_bert)
self.ext_layer = ExtTransformerEncoder(
self.bert.model.config.hidden_size, args.ext_ff_size,
args.ext_heads,           args.ext_dropout, args.ext_layers)       if
args.encoder == 'baseline':
          bert_config = BertConfig(self.bert.model.config.vocab_size,
hidden_size=args.ext_hidden_size,
num_hidden_layers=args.ext_layers,
num_attention_heads=args.ext_heads,
intermediate_size=args.ext_ff_size)           self.bert.model =
BertModel(bert_config)            self.ext_layer =
Classifier(self.bert.model.config.hidden_size)


      if args.max_pos > 512:
         my_pos_embeddings = nn.Embedding(args.max_pos,
                                  self.bert.model.config.hidden_size)
my_pos_embeddings.weight.data[
         :512]   =   self.bert.model.embeddings.position_embeddings.weight.data
my_pos_embeddings.weight.data[512:]                              =                              \
self.bert.model.embeddings.position_embeddings.weight.data[-1][None,
         :].repeat(args.max_pos - 512, 1)
self.bert.model.embeddings.position_embeddings = my_pos_embeddings
            self.load_state_dict(checkpoint['model'],
strict=True)        self.to(device)


  def forward(self, src, segs, clss, mask_src, mask_cls):         top_vec =
self.bert(src, segs, mask_src)       sents_vec =
top_vec[torch.arange(top_vec.size(0)).unsqueeze(1), clss]       sents_vec
= sents_vec * mask_cls[:, :, None].float()       sent_scores =
self.ext_layer(sents_vec, mask_cls).squeeze(-1)       return sent_scores,
mask_cls
```

Bert.py

```
import torch import torch.nn as nn from
pytorch_transformers import BertModel


class Bert(nn.Module):    def __init__(self,
large,  temp_dir, finetune=False):
super(Bert, self).__init__()        if(large):
        self.model = BertModel.from_pretrained('bert-large-uncased', cache_dir=temp_dir)
else:
        self.model = BertModel.from_pretrained('bert-base-uncased', cache_dir=temp_dir)
self.finetune = finetune
    def forward(self, x, segs, mask):
if(self.finetune):            top_vec, _ = self.model(x,
segs, attention_mask=mask)        else:
self.eval()            with torch.no_grad():
top_vec, _ = self.model(x, segs, attention_mask=mask)
return top_vec
```

TransformerDecoder.py

```
import torch import torch.nn as nn from
src.models.submodels.Encoder import PositionalEncoding from
src.models.submodels.Decoder import TransformerDecoderLayer from
src.models.submodels.Decoder import TransformerDecoderState


class TransformerDecoder(nn.Module):


  def __init__(self, num_layers, d_model, heads, d_ff, dropout, embeddings):
super(TransformerDecoder, self).__init__()


    # Basic attributes.        self.decoder_type = 'transformer'
self.num_layers = num_layers
self.embeddings = embeddings        self.pos_emb =
PositionalEncoding(dropout,self.embeddings.embedding_dim)


    self.transformer_layers = nn.ModuleList(
        [TransformerDecoderLayer(d_model, heads, d_ff, dropout)
for _ in range(num_layers)])


    self.layer_norm = nn.LayerNorm(d_model, eps=1e-6)


  def forward(self, tgt, memory_bank, state, memory_lengths=None,
step=None, cache=None,memory_masks=None):
        src_words = state.src
tgt_words
= tgt        src_batch, src_len =
src_words.size()        tgt_batch, tgt_len =
tgt_words.size()


    # Run the forward pass of the TransformerDecoder.
# emb = self.embeddings(tgt, step=step)        emb =
self.embeddings(tgt)        assert emb.dim() == 3  # len x
```

batch x embedding_dim

output = self.pos_emb(emb, step)

src_memory_bank = memory_bank padding_idx = self.embeddings.padding_idx tgt_pad_mask = tgt_words.data.eq(padding_idx).unsqueeze(1) \ .expand(tgt_batch, tgt_len, tgt_len)

if (not memory_masks is None): src_len = memory_masks.size(-1) src_pad_mask = memory_masks.expand(src_batch, tgt_len, src_len)

else: src_pad_mask = src_words.data.eq(padding_idx).unsqueeze(1) \ .expand(src_batch, tgt_len, src_len) if state.cache is None: saved_inputs = [] for i in range(self.num_layers): prev_layer_input = None if state.cache is None: if state.previous_input is not None:
prev_layer_input = state.previous_layer_inputs[i] output, all_input \ = self.transformer_layers[i](output, src_memory_bank, src_pad_mask, tgt_pad_mask, previous_input=prev_layer_input, layer_cache=state.cache["layer_{}".format(i)] if state.cache is not None else None, step=step) if state.cache is None: saved_inputs.append(all_input) if state.cache is None:
saved_inputs = torch.stack(saved_inputs) output = self.layer_norm(output) if state.cache is None: state = state.update_state(tgt, saved_inputs) return output, state
def init_decoder_state(self, src, memory_bank, with_cache=False): state = TransformerDecoderState(src) if with_cache: state._init_cache(memory_bank, self.num_layers) return state

Decoder.py

import numpy as np import torch import torch.nn as nn from src.models.submodels import MultiHeadedAttention, PositionwiseFeedForward, DecoderState from src.models.submodels import PositionalEncoding

MAX_SIZE = 5000

class TransformerDecoderLayer(nn.Module):

def __init__(self, d_model, heads, d_ff, dropout): super(TransformerDecoderLayer, self).__init__()

self.self_attn = MultiHeadedAttention(heads, d_model, dropout=dropout)

self.context_attn = MultiHeadedAttention(heads, d_model, dropout=dropout) self.feed_forward = PositionwiseFeedForward(d_model, d_ff, dropout) self.layer_norm_1 = nn.LayerNorm(d_model, eps=1e-6) self.layer_norm_2 = nn.LayerNorm(d_model, eps=1e-6) self.drop = nn.Dropout(dropout)

```
mask = self._get_attn_subsequent_mask(MAX_SIZE)        # Register
self.mask as a buffer in TransformerDecoderLayer, so        # it gets
TransformerDecoderLayer's cuda behavior automatically.
self.register_buffer('mask', mask)


   def forward(self, inputs, memory_bank, src_pad_mask, tgt_pad_mask,
previous_input=None, layer_cache=None, step=None):
        dec_mask = torch.gt(tgt_pad_mask
+
self.mask[:, :tgt_pad_mask.size(1),
                                :tgt_pad_mask.size(1)], 0)
input_norm = self.layer_norm_1(inputs)        all_input =
input_norm        if previous_input is not None:                all_input
= torch.cat((previous_input, input_norm), dim=1)
dec_mask = None
        query = self.self_attn(all_input, all_input,
input_norm,
mask=dec_mask,                                  layer_cache=layer_cache,
type="self")


       query = self.drop(query) + inputs


       query_norm = self.layer_norm_2(query)        mid =
self.context_attn(memory_bank, memory_bank, query_norm,
                                mask=src_pad_mask,
layer_cache=layer_cache,                                  type="context")
output = self.feed_forward(self.drop(mid) + query)
        return output,
all_input         # return
output


   def _get_attn_subsequent_mask(self, size):
        attn_shape = (1, size,
size)
       subsequent_mask = np.triu(np.ones(attn_shape), k=1).astype('uint8')
subsequent_mask = torch.from_numpy(subsequent_mask)        return
subsequent_mask




class TransformerDecoder(nn.Module):


   def __init__(self, num_layers, d_model, heads, d_ff, dropout, embeddings):
super(TransformerDecoder, self).__init__()
         # Basic
attributes.
       self.decoder_type = 'transformer'        self.num_layers = num_layers
self.embeddings = embeddings        self.pos_emb =
PositionalEncoding(dropout,self.embeddings.embedding_dim)



       # Build TransformerDecoder.
       self.transformer_layers = nn.ModuleList(
          [TransformerDecoderLayer(d_model, heads, d_ff, dropout)
for _ in range(num_layers)])


       self.layer_norm = nn.LayerNorm(d_model, eps=1e-6)


   def forward(self, tgt, memory_bank, state, memory_lengths=None,
step=None, cache=None,memory_masks=None):
```

```
        src_words = state.src
tgt_words = tgt        src_batch, src_len =
src_words.size()        tgt_batch, tgt_len
= tgt_words.size()


      # Run the forward pass of the TransformerDecoder.        #
emb = self.embeddings(tgt, step=step)        emb =
self.embeddings(tgt)        assert emb.dim() == 3  # len x
batch x embedding_dim


      output = self.pos_emb(emb, step)


      src_memory_bank = memory_bank          padding_idx =
self.embeddings.padding_idx          tgt_pad_mask =
tgt_words.data.eq(padding_idx).unsqueeze(1) \
.expand(tgt_batch, tgt_len, tgt_len)


      if (not memory_masks is None):            src_len =
memory_masks.size(-1)            src_pad_mask =
memory_masks.expand(src_batch, tgt_len, src_len)   else:
src_pad_mask = src_words.data.eq(padding_idx).unsqueeze(1) \
.expand(src_batch, tgt_len,
src_len)            if state.cache is
None:            saved_inputs = []
for i in range(self.num_layers):
prev_layer_input = None            if state.cache
is None:                if state.previous_input is
not None:
                    prev_layer_input = state.previous_layer_inputs[i]
output, all_input \
              = self.transformer_layers[i](                    output,
src_memory_bank,                src_pad_mask, tgt_pad_mask,
previous_input=prev_layer_input,
layer_cache=state.cache["layer_{}".format(i)]                if
state.cache is not None else None,                    step=step)
if                state.cache                is                None:
saved_inputs.append(all_input)                if state.cache is
None:
saved_inputs = torch.stack(saved_inputs)


      output = self.layer_norm(output)
         # Process the result and update the
attentions.            if state.cache is None:
state = state.update_state(tgt, saved_inputs)
return output, state
    def init_decoder_state(self, src,
memory_bank,
with_cache=False):        """ Init decoder state """                state

= TransformerDecoderState(src)        if with_cache:

state._init_cache(memory_bank,
self.num_layers)        return state




class TransformerDecoderState(DecoderState):
""" Transformer Decoder state base class """

        def __init__(self, src):        """                Args:
src (FloatTensor): a sequence of source words tensors
with optional feature tensors, of size (len x batch).
        """          self.src = src
self.previous_input = None
self.previous_layer_inputs = None
self.cache = None
```

```
@property
def
_all(self):
"""
    Contains attributes that need to be updated in self.beam_update().
    """
    if (self.previous_input is not None and
self.previous_layer_inputs is not None):
        return (self.previous_input,
self.previous_layer_inputs,
self.src)    else:        return
(self.src,)    def detach(self):    if
self.previous_input is not None:
self.previous_input =
self.previous_input.detach()    if
self.previous_layer_inputs is not None:
self.previous_layer_inputs = self.previous_layer_inputs.detach()
self.src = self.src.detach()


  def update_state(self, new_input, previous_layer_inputs):
    state = TransformerDecoderState(self.src)
state.previous_input = new_input
state.previous_layer_inputs = previous_layer_inputs    return
state


  def _init_cache(self, memory_bank,
num_layers):    self.cache = {}        for l in
range(num_layers):        layer_cache = {
"memory_keys": None,
          "memory_values":    None        }
layer_cache["self_keys"]        =        None
layer_cache["self_values"] = None
self.cache["layer_{}".format(l)] = layer_cache


  def repeat_beam_size_times(self, beam_size):    """
Repeat beam_size times along batch dimension. """
self.src = self.src.data.repeat(1, beam_size, 1)
    def map_batch_fn(self, fn):    def
_recursive_map(struct, batch_dim=0):
for k, v in struct.items():        if v is
not None:        if isinstance(v, dict):
_recursive_map(v)        else:
struct[k] = fn(v, batch_dim)
self.src = fn(self.src,
0)
if self.cache is not None:
_recursive_map(self.cache)
```

Encoder.py

```
import math


import torch import  torch.nn as nn  from
src.models.submodels.Neural import PositionwiseFeedForward, \
MultiHeadedAttention



class Classifier(nn.Module):  def __init__(self,
hidden_size):    super(Classifier,
self).__init__()    self.linear1 =
nn.Linear(hidden_size, 1)    self.sigmoid =
nn.Sigmoid()
    def forward(self, x, mask_cls):
h =
self.linear1(x).squeeze(-1)    sent_scores = self.sigmoid(h)
```

```
* mask_cls.float()        return
sent_scores



class PositionalEncoding(nn.Module):


   def __init__(self, dropout, dim, max_len=5000):        pe =
torch.zeros(max_len, dim)        position = torch.arange(0,
max_len).unsqueeze(1)        div_term = torch.exp((torch.arange(0, dim,
2, dtype=torch.float) *
                        -(math.log(10000.0) / dim)))        pe[:,
0::2] = torch.sin(position.float() * div_term)        pe[:,
1::2] = torch.cos(position.float() * div_term)        pe =
pe.unsqueeze(0)        super(PositionalEncoding,
self).__init__()        self.register_buffer('pe', pe)
self.dropout = nn.Dropout(p=dropout)        self.dim =
dim


   def forward(self, emb, step=None):
emb = emb * math.sqrt(self.dim)        if
(step):             emb = emb + self.pe[:, step][:,
None, :]


else:             emb = emb + self.pe[:,
:emb.size(1)]         emb =
self.dropout(emb)        return emb
      def get_emb(self, emb):
return self.pe[:,
:emb.size(1)]



class TransformerEncoderLayer(nn.Module):    def
__init__(self, d_model, heads, d_ff, dropout):
super(TransformerEncoderLayer, self).__init__()


      self.self_attn = MultiHeadedAttention(            heads, d_model,
dropout=dropout)        self.feed_forward =
PositionwiseFeedForward(d_model, d_ff, dropout)        self.layer_norm =
nn.LayerNorm(d_model, eps=1e-6)        self.dropout =
nn.Dropout(dropout)
      def forward(self, iter, query, inputs,
mask):         if (iter != 0):
         input_norm = self.layer_norm(inputs)
else:             input_norm = inputs


      mask = mask.unsqueeze(1)
      context = self.self_attn(input_norm, input_norm, input_norm,
                           mask=mask)         out =
self.dropout(context) + inputs        return
self.feed_forward(out)



class ExtTransformerEncoder(nn.Module):    def __init__(self,
d_model,  d_ff, heads, dropout, num_inter_layers=0):
super(ExtTransformerEncoder, self).__init__()         self.d_model =
d_model        self.num_inter_layers = num_inter_layers
self.pos_emb =
PositionalEncoding(dropout, d_model)        self.transformer_inter
= nn.ModuleList(
         [TransformerEncoderLayer(d_model, heads, d_ff, dropout)
for _ in range(num_inter_layers)])        self.dropout =
nn.Dropout(dropout)        self.layer_norm =
```

```
nn.LayerNorm(d_model, eps=1e-6)        self.wo =
nn.Linear(d_model, 1, bias=True)       self.sigmoid =   nn.Sigmoid()


   def forward(self, top_vecs, mask):
""" See :obj:`EncoderBase.forward()`"""

          batch_size, n_sents = top_vecs.size(0),
top_vecs.size(1)         pos_emb = self.pos_emb.pe[:,
:n_sents]        x = top_vecs * mask[:, :, None].float()        x
= x + pos_emb
          for i in range(self.num_inter_layers):              x = self.transformer_inter[i](i, x,
x, 1 - mask)  # all_sents * max_tokens * dim
          x = self.layer_norm(x)
sent_scores =
self.sigmoid(self.wo(x))        sent_scores = sent_scores.squeeze(-1)
* mask.float()


       return sent_scores
```

Neural.py

```
import math



def aeq(*args):
"""
   Assert all arguments have the same value
"""      arguments = (arg for arg in args)     first =
next(arguments)    assert all(arg == first for arg
in arguments), \
       "Not all arguments have the same value: " + str(args)



def sequence_mask(lengths, max_len=None):
   """
   Creates a boolean mask from sequence lengths.
   """
   batch_size = lengths.numel()      max_len =
max_len or lengths.max()    return
(torch.arange(0, max_len)
.type_as(lengths)
          .repeat(batch_size, 1)
          .lt(lengths.unsqueeze(1)))



def gelu(x):
   return 0.5 * x * (1 + torch.tanh(math.sqrt(2 / math.pi) * (x + 0.044715 * torch.pow(x, 3))))

import torch import torch.nn
as nn import torch.nn.functional
as F
```

```
class GlobalAttention(nn.Module):
      def __init__(self, dim,  attn_type="dot"):
super(GlobalAttention, self).__init__()
self.dim = dim        assert attn_type in ["dot",
"general", "mlp"], (            "Please select a
valid attention type.")       self.attn_type =
attn_type
          if self.attn_type == "general":
self.linear_in = nn.Linear(dim, dim, bias=False)
elif self.attn_type == "mlp":
          self.linear_context = nn.Linear(dim, dim, bias=False)
self.linear_query = nn.Linear(dim, dim, bias=True)            self.v
= nn.Linear(dim, 1, bias=False)
      # mlp wants it with bias        out_bias = self.attn_type
== "mlp"       self.linear_out = nn.Linear(dim * 2, dim,
bias=out_bias)


      def score(self, h_t,
h_s):
      # Check input sizes
      src_batch, src_len, src_dim = h_s.size()
tgt_batch, tgt_len, tgt_dim = h_t.size()       if
self.attn_type in ["general", "dot"]:           if
self.attn_type == "general":               h_t_ =
h_t.view(tgt_batch * tgt_len, tgt_dim)               h_t_ =
self.linear_in(h_t_)               h_t = h_t_.view(tgt_batch,
tgt_len, tgt_dim)           h_s_ = h_s.transpose(1, 2)
          # (batch, t_len, d) x (batch, d, s_len) --> (batch, t_len, s_len)
return torch.bmm(h_t, h_s_)       else:
          dim = self.dim
          wq = self.linear_query(h_t.view(-1, dim))
wq = wq.view(tgt_batch, tgt_len, 1, dim)           wq =
wq.expand(tgt_batch, tgt_len, src_len, dim)
             uh = self.linear_context(h_s.contiguous().view(-1, dim))
uh = uh.view(src_batch, 1, src_len, dim)           uh =
uh.expand(src_batch, tgt_len, src_len, dim)


          # (batch, t_len, s_len, d)
wquh = torch.tanh(wq + uh)


          return self.v(wquh.view(-1, dim)).view(tgt_batch, tgt_len, src_len)


   def forward(self, source, memory_bank, memory_lengths=None, memory_masks=None):


      # one step input       if
source.dim() == 2:           one_step =
True          source =
source.unsqueeze(1)       else:
          one_step = False


      batch, source_l, dim = memory_bank.size()
batch_, target_l, dim_ = source.size()
        # compute attention scores, as in Luong et
al.
align = self.score(source, memory_bank)


      if memory_masks is not None:
memory_masks = memory_masks.transpose(0,1)
memory_masks = memory_masks.transpose(1,2)
align.masked_fill_(1 - memory_masks.byte(), -
float('inf'))
```

```
if memory_lengths is not None:  mask =
sequence_mask(memory_lengths, max_len=align.size(-1))  mask
= mask.unsqueeze(1)  # Make it broadcastable.
align.masked_fill_(1 - mask, -float('inf'))


align_vectors = F.softmax(align.view(batch*target_l, source_l), -1)
align_vectors = align_vectors.view(batch, target_l, source_l)


c = torch.bmm(align_vectors, memory_bank)


# concatenate  concat_c = torch.cat([c, source],
2).view(batch*target_l, dim*2)  attn_h =
self.linear_out(concat_c).view(batch, target_l, dim)
if self.attn_type in ["general", "dot"]:  attn_h =
torch.tanh(attn_h)  if one_step:  attn_h =
attn_h.squeeze(1)  align_vectors =
align_vectors.squeeze(1)  else:
  attn_h = attn_h.transpose(0, 1).contiguous()
align_vectors = align_vectors.transpose(0, 1).contiguous()
  return attn_h,
align_vectors



class PositionwiseFeedForward(nn.Module):


 def __init__(self, d_model, d_ff, dropout=0.1):
super(PositionwiseFeedForward, self).__init__()
self.w_1 = nn.Linear(d_model, d_ff)  self.w_2 =
nn.Linear(d_ff, d_model)  self.layer_norm =
nn.LayerNorm(d_model, eps=1e-6)  self.actv = gelu
self.dropout_1 = nn.Dropout(dropout)  self.dropout_2 =
nn.Dropout(dropout)
 def forward(self, x):
inter =
self.dropout_1(self.actv(self.w_1(self.layer_norm(x))))
output = self.dropout_2(self.w_2(inter))  return output + x



class MultiHeadedAttention(nn.Module):


 def __init__(self, head_count, model_dim, dropout=0.1, use_final_linear=True):
assert model_dim % head_count == 0  self.dim_per_head = model_dim // head_count
self.model_dim = model_dim


 super(MultiHeadedAttention, self).__init__()
self.head_count  =  head_count
self.linear_keys =
nn.Linear(model_dim,
head_count * self.dim_per_head)  self.linear_values
= nn.Linear(model_dim,  head_count
* self.dim_per_head)  self.linear_query
= nn.Linear(model_dim,  head_count
* self.dim_per_head)  self.softmax =
nn.Softmax(dim=1)  self.dropout = nn.Dropout(dropout)
self.use_final_linear = use_final_linear  if
(self.use_final_linear):  self.final_linear =
nn.Linear(model_dim, model_dim)  def forward(self, key,
value, query, mask=None,  layer_cache=None,
type=None, predefined_graph_1=None):
  batch_size = key.size(0)
dim_per_head = self.dim_per_head
head_count = self.head_count
key_len = key.size(1)  query_len =
query.size(1)
```

```
        def           shape(x):
"""  projection """
          return x.view(batch_size, -1, head_count, dim_per_head) \
.transpose(1, 2)
def unshape(x):
"""  compute context """
return                  x.transpose(1,
2).contiguous() \
              .view(batch_size, -1, head_count *
dim_per_head)             # 1) Project key, value, and query.
if layer_cache is not None:             if type == "self":
query, key, value = self.linear_query(query), \
self.linear_keys(query), \
self.linear_values(query)                    key = shape(key)               value
= shape(value)
                if layer_cache is not None:                    device
= key.device                    if layer_cache["self_keys"] is not
None:
                    key = torch.cat(
                        (layer_cache["self_keys"].to(device), key),
dim=2)                    if layer_cache["self_values"] is not None:
value = torch.cat(
                        (layer_cache["self_values"].to(device), value),
dim=2)                    layer_cache["self_keys"] = key
layer_cache["self_values"] = value           elif type == "context":
             query = self.linear_query(query)                    if
layer_cache is not None:                                         if
layer_cache["memory_keys"] is None:                        key,
value              =              self.linear_keys(key),              \
self.linear_values(value)                        key = shape(key)
value = shape(value)                  else:                        key,
value = layer_cache["memory_keys"], \
layer_cache["memory_values"]                  layer_cache["memory_keys"]
= key                  layer_cache["memory_values"] = value                else:
key, value = self.linear_keys(key), \
self.linear_values(value)                    key =
shape(key)                  value = shape(value)
else:            key = self.linear_keys(key)
value = self.linear_values(value)
query = self.linear_query(query)           key =
shape(key)           value = shape(value)


      query = shape(query)
key_len = key.size(2)
query_len = query.size(2)
          # 2) Calculate and scale scores.
query = query / math.sqrt(dim_per_head)
scores = torch.matmul(query, key.transpose(2,
3))
          if mask is not None:             mask =
mask.unsqueeze(1).expand_as(scores)
bool_mask = mask.type(torch.BoolTensor)
scores = scores.masked_fill(bool_mask, -1e18)        # 3)
Apply attention dropout and compute context vectors.
          attn =
self.softmax(scores)
          if (not predefined_graph_1 is None):             attn_masked = attn[:, -
1] * predefined_graph_1           attn_masked =   attn_masked /
(torch.sum(attn_masked, 2).unsqueeze(2) + 1e-9)


          attn = torch.cat([attn[:, :-1], attn_masked.unsqueeze(1)],
1)            drop_attn = self.dropout(attn)         if
(self.use_final_linear):
          context = unshape(torch.matmul(drop_attn, value))
output = self.final_linear(context)           return output
else:             context = torch.matmul(drop_attn, value)
return context
```

```
      # CHECK
      # batch_, q_len_, d_ = output.size()
      # aeq(q_len, q_len_)
# aeq(batch, batch_)
# aeq(d, d_)


      # Return one attn




class DecoderState(object):     def
detach(self):          """ Need to document this
"""
      self.hidden = tuple([_.detach() for _ in self.hidden])
self.input_feed = self.input_feed.detach()


   def beam_update(self, idx, positions, beam_size):
""" Need to document this """               for e in
self._all:          sizes = e.size()          br = sizes[1]          if
len(sizes) == 3:
            sent_states = e.view(sizes[0], beam_size, br // beam_size,
                                 sizes[2])[:, :, idx]
else:
            sent_states = e.view(sizes[0], beam_size,
br // beam_size,                                 sizes[2],
sizes[3])[:, :, idx]

sent_states.data.copy_(
sent_states.data.index_select(1, positions))
      def map_batch_fn(self, fn):
raise NotImplementedError()
```