CH5019: Mathematical Foundations of Data Science

Group 10 Project

Answer Similarity Calculator

Group Members:

Devansh Sanghvi (BE19B002), Pradhyumn Jain (BS19B021), Ishan Chokshi (BE19B018), Siddharth Betala (BE19B032)

Acknowledgement

We would like to thank Professor Raghunathan Rengaswamy for giving us this wonderful opportunity to work on this project as part of the Mathematical Foundations in Data Science course, CH5019. We would also like to thank the TAs for this course who have helped us to gain the most through the weekly meetings and short tests which have sharpened our basics required in Data Science. Through this project, we were able to thoroughly understand how a few models in Natural Language Processing work and were able to implement them on a very interesting problem: Automatic Subjective Answer grading.

Introduction

To build an autograding model, we've experimented with various NLP algorithms. The following flowchart is the gist of the approach we've taken. In this report we'll go on to describe our approach in detail, the preprocessing steps, the NLP models we've used and the calculation of the similarity metric. Our model will take in the student answer and the model answer provided by the instructor as input and will return a similarity score. The dataset and a link to the model code has been provided in this report.

Link to Code : Google Colab

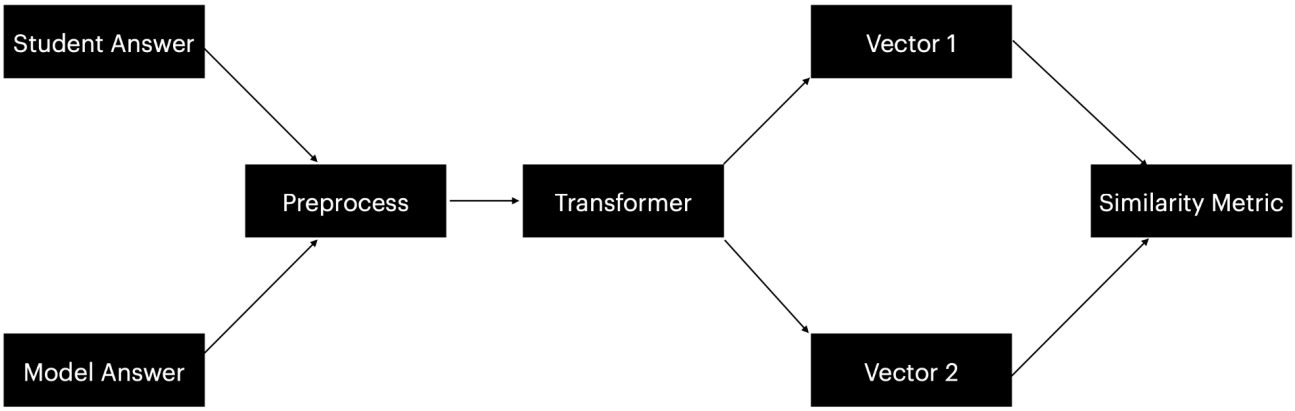


Fig 1. Model Outline

1. **Preprocessing the text**

We preprocessed the text before feeding it into the model. This included converting all characters to lowercase letters, removing punctuations and stop words (Note : Some models use punctuations and stop words to infer the meaning of the text). Preprocessing steps for each model have been described under their respective headings.

2. **Feeding the text into a Vectorizer**

The vectorizer is where the NLP model is deployed. It takes in a text and returns a vector. This vector is calculated based on factors such as the sentiment of the text, the type of words used, the length of the text to name a few. Please refer to the models section for a detailed explanation of how the vectorizer works. Once we have two vectors, one for the model answer and one for the student's answer, we use them to calculate a similarity metric.

3. Cosine Similarity

The similarity metric that our function returns is cosine similarity. It is calculated using the following formula:

$$Cos\theta = \frac{V_1 \cdot V_2}{|V_1| \cdot |V_2|}$$

Here, V_1 and V_2 are the vectors obtained by vectorizing the model answer and the student answer. The correctness of the answer can be gauged from the cosine similarity between the two vectors.

Models

1. Doc2Vec

The goal of using the Doc2Vec Model is to create a numeric representation of the text document, regardless of the length of the document. The Doc2Vec Model is heavily based on another model called the **Word2Vec Model** and therefore it's best to understand Word2Vec first.

Word2Vec is a common method of generating **word embeddings** of the given text and it is used to extract the notion of similarity between words such as identifying the same context, identifying synonyms in text and being able to understand the semantic relatedness. Before we move ahead, let's understand what word embeddings are.

Word embeddings are a crucial part of **Natural Language Processing**. They essentially depict how humans understand language to a machine. In simple words, word embeddings **convert every individual word into a vector** that tries to capture various characteristics of the word with respect to the entire text. We do this because a machine cannot understand raw text and so that we can feed these vectors to a machine learning algorithm. There are many word embedding techniques such as one-hot encoding, **Word2Vec, TF-IDF and FastText** amongst others, and we have covered/compared a couple of them in this project. Now that word embeddings are clear, let's see what Word2Vec is and how was it used to develop Doc2Vec (one of the models we use in our project).

Using a large enough dataset, Word2Vec models can estimate the meaning of the word based on their occurrences of the text. Through this, we can yield relationships between different words. A simple example is given below:

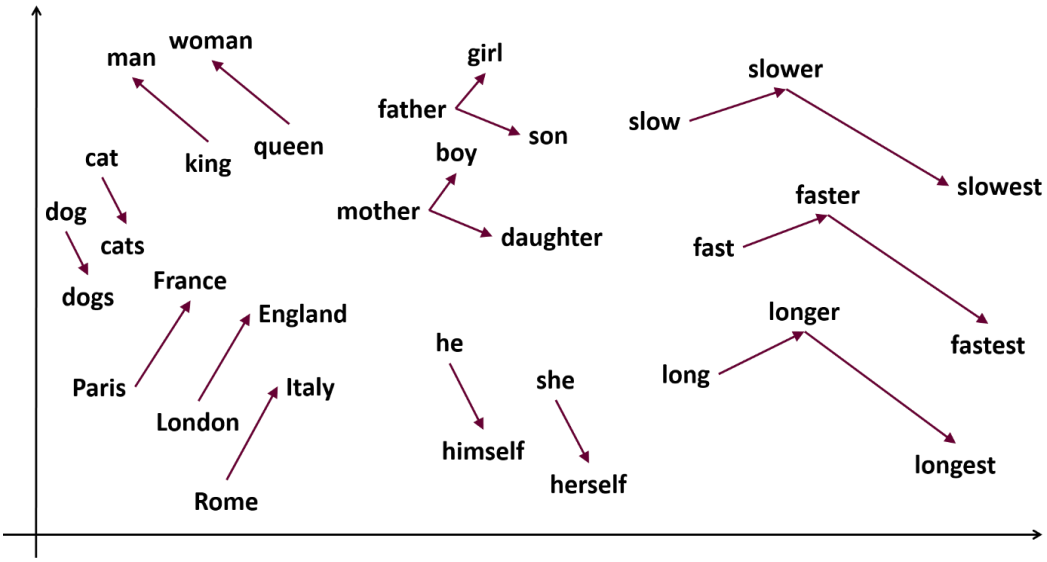


Fig 3. Vector representation of words in a given text.
Each vector represents the distance between two similar words

As seen in the picture above, the distances between two similar meaning words are the least and we can apply appropriate mathematical operations on these vectors. Without going into too much detail, the two architectures that are responsible for Word2Vec's success are: The **skip-gram and the CBOW architectures**.

CBOW or the Continuous Bag Of Words architecture is very similar to a feed forward neural network and it predicts the middle word based on surrounding context words. The Skip-Gram architecture is actually the opposite of CBOW architecture and it predicts the contextual words surrounding the given word. A simple representation of the working of the CBOW model is given below.

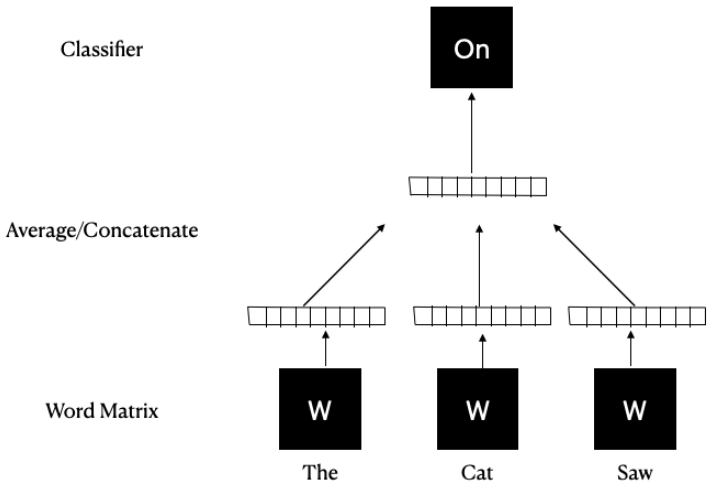


Fig 4. CBOW algorithm used to predict the word "on"
using the context words "the","cat","sat"

Having understood word embeddings and Word2Vec, we will now look at what Doc2Vec is and why we chose Doc2Vec as one of our models.

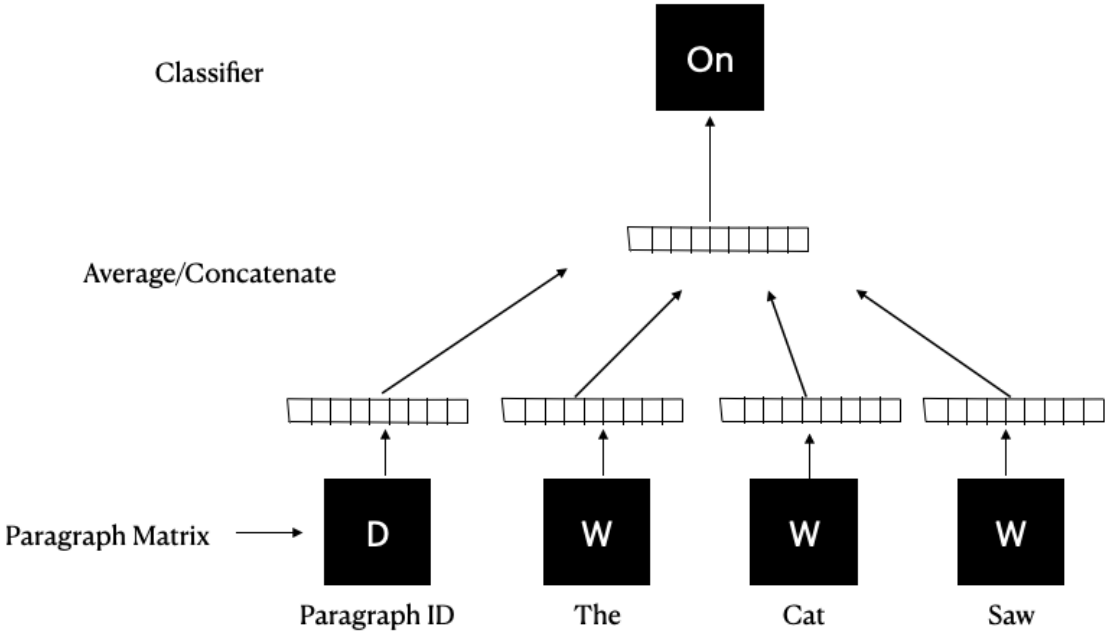


Fig 4. PV-DM algorithm used to predict the word “on”,
using the context words “the”,”cat”,”sat”.

It essentially adds another vector called the **Paragraph ID** to the existing framework. With the training of the words, the model trains the document vector D as well, to hold a numeric representation of the document. This model is called the **Distributed Memory version of Paragraph Vector (PV-DM)**. Similar to the skip-gram model which works in the opposite direction to the CBOW model, supporting Doc2Vec is the **Distributed Bag of Words version of Paragraph Vector (PV-DBOW)** model, which tries to predict the contextual words in a paragraph when given the paragraph ID.

The basics of Doc2Vec are covered, but how does Doc2Vec help in our project?

As with the other embedding and transformer models mentioned below, we use Doc2Vec to create the word embeddings for the two answers that have been provided to the model. Using these vectors, we calculate the similarity between the two texts using the similarity measures mentioned above.

2. TF-IDF

TF-IDF (term frequency-inverse document frequency) is a transformer used to determine the significance of a word to a document in a document or set of documents. When it comes to the importance of articles, prepositions, and conjunctions, as well as the relevance of a word that is not repeated more than once, traditional methods such as Bag of Words that determine the importance of words based on their

frequency in a text usually fail. As the frequency of a word increases, its importance is counterbalanced by the term's frequency in the dataset. We can solve this problem with the aid of TF-IDF. Words that are used too frequently do not overshadow words that are used less frequently yet are vital.
This model accounts for the TF: term frequency (# times word occurs in document/ #words in the document) and the IDF: inverse document frequency log(# total documents in the dataset/ # documents containing the word 'X').

$$TF(t,d) = \frac{f(t)}{T(d)} \qquad IDF(t) = \frac{N}{df(t)}$$

$f(t) = frequency\ of\ t\ document\ d$

$T(d) = Total\ words\ in\ document\ d$

$N = Total\ documents\ in\ the\ set$

$d(t) = Occurrence\ of\ in\ documents$

However, as N increases, the value of IDF will explode. To avoid this, we use a modified formula of IDF:

$$IDF(t) = log\frac{N}{df+1}$$

The final formula becomes:

$$TF - IDF(t,d) = TF(t,d) * log\frac{N}{df+1}$$

For our model, sklearn TF-IDF Vectorizer was used which generates multidimensional vectors for the answers. This model is trained and fitted on the model answers to generate a matrix of TF-IDF features. We build a vocabulary that only considers the top 100 features ordered by term frequency. All characters are converted to lowercase before tokenizing. Stop words have been accounted for using the 'stop_words' argument. This model is then tested on the student answers and the cosine similarity is used as a metric to check similarity between the answers.

3. BERT (Bidirectional Encoder Representations from Transformers)

Background and Motivation

The absence of sufficient training data is one of the most significant issues in NLP. Although there is a vast amount of text data available, we must divide it into many different fields in order to build task-specific datasets. And we only wind up with a few thousand or a few hundred thousand human-labeled training instances when we do this. Unfortunately, deep learning-based NLP models require significantly more data to perform well – they observe significant increases when trained on lots of annotated training instances. Researchers have devised several ways for training general-purpose

language representation models using the massive mounds of unannotated material on the web to assist bridge this data gap. When working with challenges like sentiment analysis, these general-purpose pre-trained models can subsequently be fine-tuned on smaller task-specific datasets. This approach results in greater accuracy compared to training on the smaller task-specific datasets from scratch. BERT is a relatively new addition to these NLP pre-training approaches.

In the field of computer vision, researchers have repeatedly shown the value of transfer learning - pre-training a neural network model on a known task, and then performing fine-tuning - using the trained neural network as the basis of a new purpose-specific model. Similarly, in BERT, a pre-trained neural network produces word embeddings which are then used as features in NLP models.

A language model would look at a particular text sequence during training from either left-to-right or a combination of left-to-right and right-to-left in the pre-BERT world. For producing sentences, this one-directional strategy works well: we can predict the next word, append it to the sequence, and then predict the next to the next word until we have a complete phrase. Now comes BERT, a bi-directionally trained language model. In comparison to single-direction language models, this implies we can now have a better understanding of language context and flow. Instead of predicting the next word in a sequence, BERT employs a novel technique known as MLM (Masked Language Modelling): it randomly masks words in the sentence before attempting to predict them. Masking implies that the model looks in both directions and uses the entire context of the sentence, including both the left and right surroundings, to predict the masked word. Unlike previous language models, it considers both the previous and next tokens at the same time. This "same-time part" was missing from existing combined left-to-right and right-to-left LSTM-based models.

Working

BERT is powered by a Transformer, which works by taking a small, continuous number of steps. It uses an attention mechanism in each step to understand relationships between all words in a sentence, regardless of their position. Transformers are typically made up of an encoder that reads the text input and a decoder that produces a prediction for the task. BERT only requires the encoder part because its goal is to generate a language representation model. The encoder for BERT receives a sequence of tokens, which are then converted into vectors and processed in the neural network.

Training language models comes with the challenge of defining a prediction goal. Many models predict the next word in a sequence (for example, "The man returned from ___"), a directional approach that limits context learning. BERT employs two training strategies to overcome this obstacle:

1. **MLM**: The concept is "simple": Mask out 15% of the words in the input at random, replacing them with a [MASK] token, then run the entire sequence through the BERT attention-based encoder, predicting only the masked words based on the context provided by the remaining non-masked words in the sequence. However, there is a flaw in this naive masking approach: the model only attempts to predict when the [MASK] token appears in the input, whereas we want the model to attempt to predict the correct tokens regardless of which token appears in the input. To address this issue, 15 percent of the tokens chosen for masking:

a. 80% of the tokens are actually replaced with the token [MASK].
b. 10% of the time tokens are replaced with a random token.
c. 10% of the time tokens are left unchanged.

The model then attempts to predict the original value of the masked words, based on the context provided by the other, non-masked, words in the sequence. In technical terms, the prediction of the output words requires the following steps:

a. On top of the encoder output, a classification layer is added.
b. The output vectors are transformed into the vocabulary dimension by multiplying them by the embedding matrix.
c. Softmax is used to calculate the probability of each word in the vocabulary.

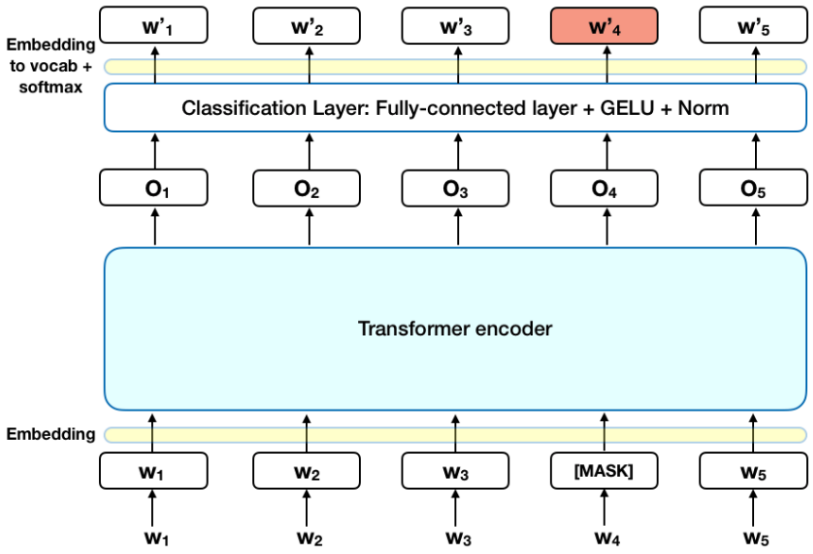


Fig 4. The Transformer encoder is depicted in the image above. The input is a series of tokens that are embedded into vectors before being processed by the neural network. The output is a sequence of H-dimensional vectors, each vector corresponding to an input token with the same index.

During training, the BERT loss function only considers masked token predictions and ignores non-masked token predictions. As a result, the model converges much slower than left-to-right or right-to-left models.

2. **NSP (Next Sentence Prediction)**:

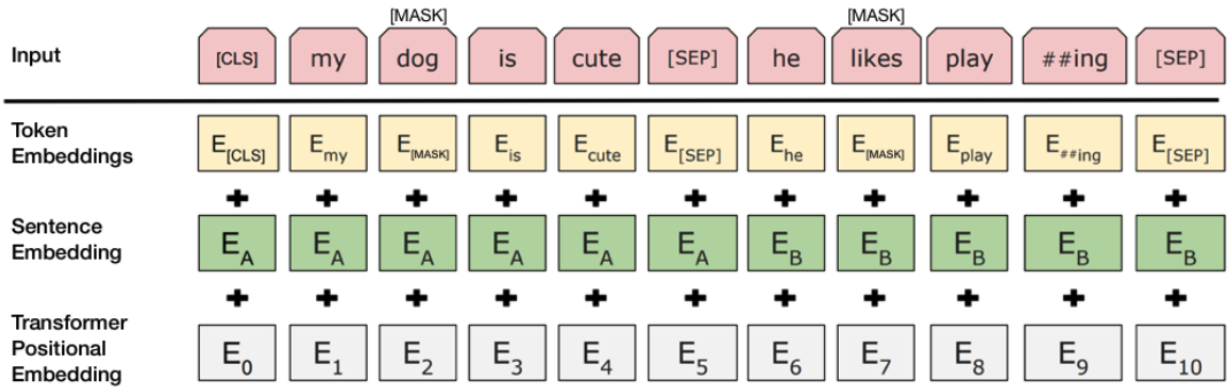
BERT training uses next sentence prediction to understand the relationship between two sentences. During training, the model is fed pairs of sentences and learns to predict whether the second sentence is also the next sentence in the original text. During training, the model is fed with two input sentences at a time such that:

a. 50% of the time the second sentence comes after the first one.
b. 50% of the time it is a random sentence from the full corpus.

BERT is then required to predict whether the second sentence is random or not, with the assumption that the random sentence will be disconnected from the first sentence.

To help the model distinguish between the two sentences in training, the input is processed in the following way before entering the model:

a. **Token embeddings:** At the beginning of the first sentence, a [CLS] token is added to the input word tokens, and a [SEP] token is inserted at the end of each sentence.
b. **Segment embeddings:** Each token has a marker that indicates whether it is Sentence A or Sentence B. This enables the encoder to differentiate between sentences.
c. **Positional embeddings:** A positional embedding is added to each token to indicate its position in the sentence.



To predict if the second sentence is indeed connected to the first, the following steps are performed:

a. The entire input sequence goes through the Transformer model.
b. The output of the [CLS] token is transformed into a 2×1 shaped vector, using a simple classification layer (learned matrices of weights and biases).
c. Calculating the probability of the second sentence being next in the sequence with softmax.

Masked LM and Next Sentence Prediction are trained together in the BERT model with the goal of minimizing the combined loss function of the two strategies.

Finetuning and Architecture

Using BERT for a specific task is relatively straightforward. Classification tasks such as in this case are done similarly to Next Sentence classification, by adding a classification layer on top of the Transformer output for the [CLS] token. In the fine-tuning training, most hyper-parameters stay the same as in BERT training.
We have used the BERT-Base model which has 12-layers, 768-hidden and equal embedding layers, 12-attention-heads, and 110 million parameters.

4. RoBERTa (Robustly Optimized BERT Pre-Training Approach)

The differences between BERT and RoBERTa are as follows:

1. **Dynamic Masking:** BERT employs static masking, which means that the same part of the sentence is masked in each Epoch. In contrast, RoBERTa employs dynamic masking, in which different parts of the sentences are masked for different epochs.This makes the model more robust.
2. **No NSP:** RoBERTa removes the Next Sentence Prediction (NSP) task from BERT's pre-training to improve the training procedure. Larger batch-training sizes were also discovered to be more beneficial in the training procedure.
3. **More Data Points:** RoBERTa uses 160 GB of text for pre-training, including 16GB of Books Corpus and English Wikipedia, both of which were used in BERT. Additional data included the CommonCrawl News dataset (63 million articles, 76 GB), a Web text corpus (38 GB), and Common Crawl Stories (31 GB).

Owing to these factors, it is expected that RoBERTa will give better results than BERT.

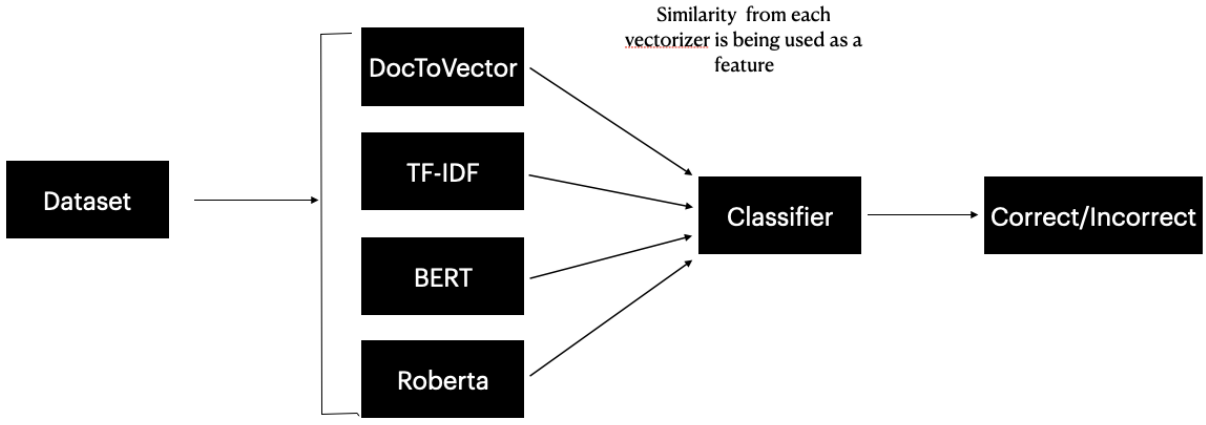
Bottleneck with BERT and RoBERTa

One of the issues with transformer models such as BERT and RoBERTa is that we cannot process long pieces of text. BERT (and many other transformer models) will consume 512 tokens max - truncating anything beyond this length. Initially, we tried splitting an input text into chunks of 512 tokens each, however, we soon realized that this issue can't be solved so easily, since we're using a prebuilt tokenizer.

So for example "This is amazing!", when tokenized using the BERT tokenizer would become something like ["[CLS]", "This", "is", "amaz", "--ing", "!", "[SEP]"] - the two tokens (CLS/SEP) are easy to deal with, but figuring out how we split all of the words/part-words, and punctuation is more complex.

We're working on a fix based on this notebook, to deal with the issue going ahead.

Results



The 10 cases can be found in the Ipython notebook submitted with this report. In this section we shall discuss the model performance on another dataset. This dataset contains ~2400 student answers, correct answer pairs along with a label 1 or 0 for correct or incorrect respectively. We used the cosine similarity from 4 models (DocToVec, Bert, TF-IDF and Roberta) as features. The classifier we used was RandomForestClassifier. After splitting the data into 80% train and 20% test, our **accuracy on the test dataset was ~ 82%.** The dataset was submitted along with the report and the code.

Analysis of the 10 test cases:
From the result of the 10 test cases, we can conclude that:

- BERT is the best model with respect to sentiment analysis while Roberta performs the poorest of all in the same case.
- TF-IDF performs poorly if the sentences are enhanced using synonyms but performs the best when proper nouns such as names or places become involved.
- BERT is the best model when the sentences refer to historical events associated with specific dates. For the same scenarios, doc2vec is the worst performing model.
- All the models performed poorly as the length of the sentences increased.

Analysis of the performance of each method on the bigger dataset:

Method	Accuracy
Doc2Vec	74.84662576687117%
BERT	75.25562372188139%
TF-IDF	78.11860940695297%
RoBERTa	78.7321063394683%

- The overall model accuracy is around **82%.**
- As expected, RoBERTa shows the best performance.
- Doc2Vec was also expected to give the poorest performance because of the non-contextual nature of the embeddings generated by it
- TF-IDF outshines BERT here. This might be attributed to the fact that the dataset consists of short answers, where the nature of TF-IDF to emphasize on important words in sentences can give it the edge.

Contribution Report

Group Members:

1. Ishan Chokshi BE19B018
2. Devansh Sanghvi BE19B002
3. Siddharth Betala BE19B032

4. Pradhyumn Jain BS19B021

**The fifth member of the group, Anutosh Bhat BE19B037, dropped out.*

We all met and discussed the approach, which models to use and then divided the work in the following manner.

Siddharth Betala BE19B032	Suggested the approach and recommended NLP models, worked on TF-IDF, BERT, RoBERTa.
Devansh Sanghvi BE19B002	Literature survey, compiled the dataset, worked on RoBERTa and ElMo.
Ishan Chokshi BE19B018	Processed the dataset, worked on WordtoVec, Elmo (not included in report) and TF-IDF.
Pradhyumn Jain BS19B021	Combined all codes, coded the classifier, resolved dependency issues, Worked on DocToVec

**Each person wrote the code for their respective models*