COMMONSENSE VALIDATION: SEN-MAKING

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

THE AWARD OF DEGREE OF

BACHELORS OF TECHNOLOGY

In

COMPUTER ENGINEERING

Under the Supervision of

Dr. Waseem Ahmed

Submitted By:

Mauwaz Ahmed Farooqui (18BCS048)

Debal Husain Abbas (18BCS046)

DEPARTMENT OF COMPUTER ENGINEERING

FACULTY OF ENGINEERING AND TECHNOLOGY

JAMIA MILLIA ISLAMIA, NEW DELHI – 110025

CERTIFICATE

This is to certify that the project report entitled **Commonsense Validation: Sen-Making** submitted by **Debal Husain Abbas** and **Mauwaz Ahmed Farooqui** to the Department of Computer Engineering, Faculty of Engineering & Technology, Jamia Millia Islamia, New Delhi – 110025 in partial fulfillment for the award of the degree of **B. Tech in (Computer Engineering)** is a *bona fide* record of project work carried out by them under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institution or University for the award of any degree.

Dr. Waseem Ahmed
Supervisor
Department of Computer Engineering
Jamia Millia Islamia, New Delhi

New Delhi — Counter signature of HOD with seal

January, 2022

DECLARATION

We declare that this project report titled **Commonsense Validation: Sen-Making** submitted in partial fulfillment of the degree of **B. Tech in (Computer Engineering)** is a record of original work carried out by us under the supervision of **Dr. Waseem Ahmed,** and has not formed the basis for the award of any other degree, in this or any other Institution or University. In keeping with the ethical practice in reporting scientific information, due acknowledgements have been made wherever the findings of others have been cited.

[illegible]

Debal Husain Abbas
18BCS046

Mauwaz ahmed

Mauwaz Ahmed Farooqui
18BCS048

New Delhi-110025

13th January, 2022

ACKNOWLEDGEMENTS

In the present world of competition there is a race of existence in which those who have the will to come forward succeed. Project is like a bridge between theoretical and practical knowledge. With this willing we joined this particular project. First of all, we would like to thank the supreme power the Almighty Allah who is obviously the one who has always guided us to work on the right path of life. Without his grace this project could not become a reality. Next to him are our parents, whom we are greatly indebted to for bringing us with love and encouragement to this stage. We are highly obliged to take the opportunity to express our gratitude and sincerely thank **Dr. Waseem Ahmad (Assistant Professor, D/o Computer Engg., Jamia Millia Islamia)** for his guidance and supervision. A special thanks to our visionary **HoD, Prof. Bashir Alam** for keeping a great balance between theoretical learning and practical knowledge throughout the course**.** Moreover, we are highly obliged in taking the opportunity to sincerely thank all the staff members of the Computer Engg. department for their generous attitude and friendly behaviour. And last but not the least we are thankful to all our teachers and friends who have always been helpful and encouraging throughout the year. A sincere thanks to all.

[illegible] Mauwaz ahmed

DEBAL HUSAIN ABBAS MAUWAZ AHMED FAROOQUI

TABLE OF CONTENTS

ABSTRACT

Introducing common sense to natural language understanding systems has received increasing research attention. It remains a fundamental question on how to evaluate whether a system has the sense-making capability. Existing benchmarks measure commonsense knowledge indirectly or without reasoning.

In this project, we train different transformer models upon labelled dataset and test whether a system can differentiate natural language statements that make sense from those that do not make sense. The results are evaluated based on the accuracy score.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Natural Language Understanding (NLU) has received increasing research attention in recent years. With language models trained on large corpora, algorithms show better performance than humans on some benchmarks. Compared to humans, however, most end-to-end trained systems are rather weak on common sense.

For example, it is straightforward for a human to understand that someone can put a turkey into a fridge but he can never put an elephant into a fridge with basic commonsense reasoning, but it can be non-trivial for a system to tell the difference.

Existing datasets test common sense indirectly through tasks that require extra knowledge. They verify whether a system is equipped with common sense by testing whether it can give a correct answer where the input does not contain such knowledge.

The project is based on the following research paper [1]–

"Does it Make Sense? And Why? A Pilot Study for Sense Making and Explanation" authored by Wang Cunxiang, Liang Shuailong, Jin Yili, Wang Yilong, Zhu Xiaodan and Zhang Yue

Human performance on the benchmark is 99.1% for the Sen-Making task. In this pilot study, we evaluate contextualized representations trained over large-scale language modeling tasks on our benchmark. Results show that there is still a large gap behind human performance despite that the models are trained over 100 million natural language sentences.

CHAPTER 2

LITERATURE

2.1 About The Research

The following is an extract from the citation of the research paper used as the basis for the project [1].

```
   title = "Does it Make Sense? And Why? A Pilot Study for Sense Making
and Explanation",

   author = "Wang, Cunxiang and Liang, Shuailong and

             Zhang, Yue and Li, Xiaonan and Gao, Tian",

   booktitle = "Proceedings of the 57th Annual Meeting of the Association
for Computational Linguistics",

   month = jul,

   year = "2019",

   address = "Florence, Itay",

   publisher = "Association for Computational Linguistics",

   url = "https://www.aclweb.org/anthology/P19-1393",
```

The research area of the paper became the problem statement for the SemEval-2020 competition [2].

The paper mentions two tasks (example mentioned in Figure 2.1)–

- First task is to choose from two natural language statements with similar wordings which one makes sense and which one does not make sense;
- The second task is to find the key reason why a given statement does not make sense.

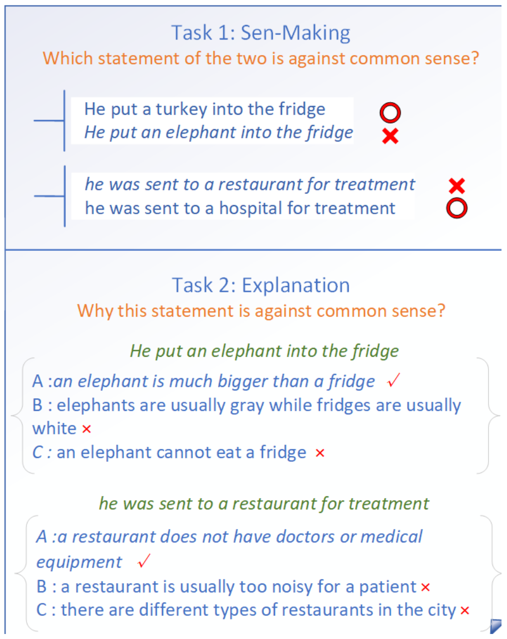
The project is an implementation of the first part of the research paper. The research area of the paper became the problem statement of the SemEval 2020 competition

With language models trained on large corpora (Peters et al., 2018; Devlin et al., 2018), algorithms show better performance than humans on some benchmarks

(Group, 2017; Devlin et al., 2018). Compared to humans, however, most end-to- end trained systems are rather weak on common sense. For example, it is straightforward for a human to understand that someone can put a turkey into a fridge but he can never put an elephant into a fridge with basic commonsense reasoning, but it can be non-trivial for a system to tell the difference.

Arguably, commonsense reasoning should be a central capability in a practical NLU system (Davis, 2017); it is, therefore, important to be able to evaluate how well a model can do for sense making.

Existing datasets test common sense indirectly through tasks that require extra knowledge, such as co-reference resolution (Levesque et al., 2012; Morgenstern and Ortiz, 2015), subsequent event prediction (Roemmele et al., 2011; Zhang et al., 2017; Zellers et al., 2018), or reading comprehension (Mostafazadeh et al., 2016; Ostermann et al., 2018b). They verify whether a system is equipped with common sense by testing whether it can give a correct answer where the input does not contain such knowledge. However, there are two limitations to such benchmarks. First, they do not give a direct metric to quantitatively measure sense making capability. Second, they do not explicitly identify the key factors required in a sense making process. We address these issues by creating a test set for direct sense making.


Task 1: Sen-Making
Which statement of the two is against common sense?

He put a turkey into the fridge ○
He put an elephant into the fridge ✗

he was sent to a restaurant for treatment ✗
he was sent to a hospital for treatment ○

Task 2: Explanation
Why this statement is against common sense?

He put an elephant into the fridge
A :*an elephant is much bigger than a fridge* ✓
B : elephants are usually gray while fridges are usually white ×
C : an elephant cannot eat a fridge ×

he was sent to a restaurant for treatment
A :*a restaurant does not have doctors or medical equipment* ✓
B : a restaurant is usually too noisy for a patient ×
C : there are different types of restaurants in the city ×

Figure 2.1: Tasks mentioned in Research Paper

The first task is to choose from two natural language statements with similar wordings which one makes sense and which one does not make sense; The second task is to find the key reason why a given statement does not make sense. Version 1.0 of our dataset, which will be released with this paper, contains 2021 instances for each subtask, manually labeled by 7 annotators. Human performance on the benchmark is 99.1% for the Sen-Making task and 97.3% for the Explanation task.

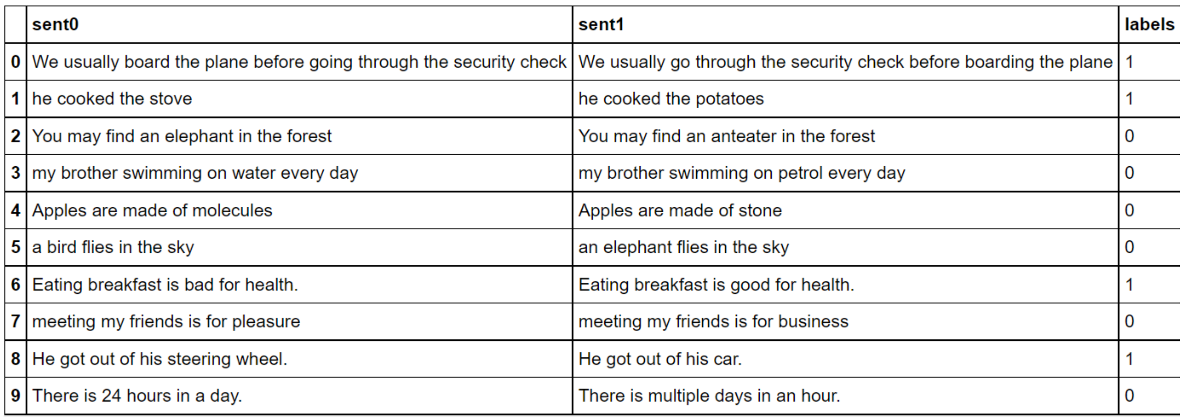
In the Sen-Making task, we use statement pair classification rather than labelling each statement 'true' or 'false' in the absolute sense because it is easy to cite a counterexample for any single **'True'** or **'False'** statement. For example, 'toy elephant' for 'he put an elephant into fridge'. But it is apparent that 'he put a turkey into fridge' is more reasonable than 'he put an elephant into fridge'.

In this pilot study, we evaluate contextualized representations trained over large-scale language modeling tasks on our benchmark. Results show that there is still a large gap behind human performance despite that the models are trained over 100 million natural language sentences. Detailed examination shows that inference remains a challenge for such systems.

2.2 Dataset Used for Training and Testing the Model

The dataset is distributed into **Train, Test and Validation subsets**. Each instance of the data consists of 2 sentences – sentence0 and sentence1, one of which makes sense while the other does not, the context of both remaining the same.

Alongside the pair of sentences is a number which is either 0 or 1 indicating which of the above statements makes sense. Each instance of data can be identified by a unique id [2].



	sent0	sent1	labels
0	We usually board the plane before going through the security check	We usually go through the security check before boarding the plane	1
1	he cooked the stove	he cooked the potatoes	1
2	You may find an elephant in the forest	You may find an anteater in the forest	0
3	my brother swimming on water every day	my brother swimming on petrol every day	0
4	Apples are made of molecules	Apples are made of stone	0
5	a bird flies in the sky	an elephant flies in the sky	0
6	Eating breakfast is bad for health.	Eating breakfast is good for health.	1
7	meeting my friends is for pleasure	meeting my friends is for business	0
8	He got out of his steering wheel.	He got out of his car.	1
9	There is 24 hours in a day.	There is multiple days in an hour.	0

Figure 2.2: Dataset Format

Table 2.1: Exploratory Data Analysis on the dataset

	Total Length of Data	**Label = 1**	**Label = 0**
Train.csv	**10,000**	**4,979**	**5,021**
Test.csv	**1,000**	**508**	**492**
Dev.csv	**997**	**518**	**479**

The dataset has the most direct decision-making process in commonsense reasoning and is the first one asking reasons behind the decision making process.

Note that there has been dataset which focus on non-linguistic world knowledge plausibility (Wang et al., 2018) or only limited attributes or actions of physical knowledge like verbphysics (Forbes and Choi, 2017). They are related to our dataset but serve robotic research mainly. This dataset is the first benchmark for direct linguistic sense making and explanation.

We choose state-of-the-art language models trained over large texts as our baselines, assuming that common sense knowledge is encoded over texts. For the sense making task, we calculate perplexities of both statements, choosing the one with lower scores as the correct one.

CHAPTER 3

TECHNOLOGY

3.1. About Deep Learning

Deep learning is a machine learning technique that teaches computers to do what comes naturally to humans: learn by example. Deep learning is a key technology behind driverless cars, enabling them to recognize a stop sign, or to distinguish a pedestrian from a lamppost. It is the key to voice control in consumer devices like phones, tablets, TVs, and hands-free speakers. Deep learning is getting lots of attention lately and for good reason. It's achieving results that were not possible before. [20]

In deep learning, a computer model learns to perform classification tasks directly from images, text, or sound. Deep learning models can achieve state-of-the-art accuracy, sometimes exceeding human-level performance. Models are trained by using a large set of labeled data and neural network architectures that contain many layers.

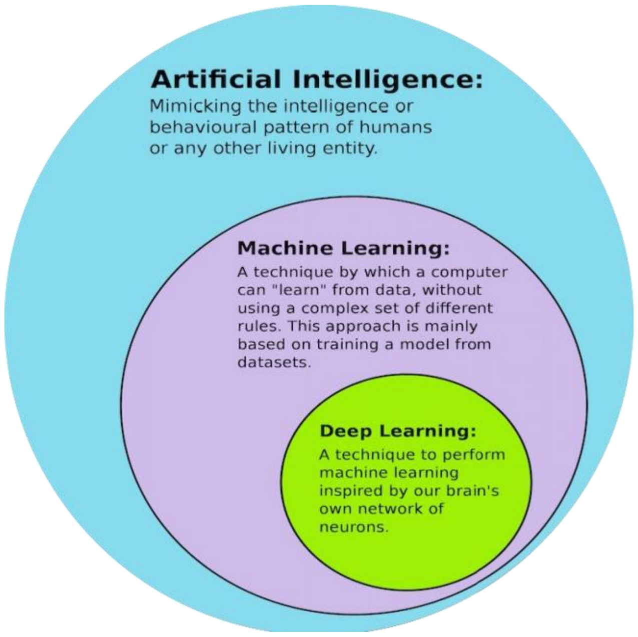

Artificial Intelligence:
Mimicking the intelligence or behavioural pattern of humans or any other living entity.
Machine Learning:
A technique by which a computer can "learn" from data, without using a complex set of different rules. This approach is mainly based on training a model from datasets.
Deep Learning:
A technique to perform machine learning inspired by our brain's own network of neurons.

Figure 3.1: Artificial Intelligence vs. Machine Learning vs. Deep Learning

3.2 Transfer Learning

Transfer learning generally refers to a process where a model trained on one problem is used in some way on a second related problem. In deep learning, transfer learning is a technique whereby a neural network model is first trained on a problem similar to the problem that is being solved. One or more layers from the trained model are then used in a new model trained on the problem of interest. Multiple deep learning domains use this approach, including Image Classification, Natural Language Processing, and even Gaming! The ability to adapt a trained model to another task is incredibly valuable. This is typically understood in a supervised learning context, where the input is the same but the target may be of a different nature. For example, we may learn about one set of visual categories, such as cats and dogs, in the first setting, then learn about a different set of visual categories, such as ants and wasps, in the second setting.

Transfer learning has the benefit of decreasing the training time for a neural network model and can result in lower generalization error. The weights in re-used layers may be used as the starting point for the training process and adapted in response to the new problem. This usage treats transfer learning as a type of weight initialization scheme. This may be useful when the first related problem has a lot more labeled data than the problem of interest and the similarity in the structure of the problem may be useful in both contexts [27].

3.2.1. Transfer Learning for Natural Language Processing:

The word embeddings used as major preprocessing step in all text-based applications are a more informative way of representing words, compared to one-hot encodings. They are widely used, and different variants exist. Typically, these variants differ in the corpus they originate from, such as Wikipedia, news articles, etc., and the differences in the embedding models. It is important to understand the background of these models and corpuses in order to know whether transfer learning with word embeddings is sensible. Essentially, using word embeddings means that you are using a featuriser or the embedding network to convert words to vectors.

An alternative to the standard pre-trained word embeddings is to fine-tune the embeddings on a large unsupervised set of documents. Note that this is only an option if a large set of documents is available. This means that if you have a large corpus on competition law, you could train the word embeddings for the domain-specific words, by starting from pre-trained word embeddings for the other, more general words. Typically, starting for pre-trained word embeddings will speed up the whole process, and will make training your own word embeddings easier. Note that it is still harder using word embeddings out-of-the-box and requires some knowledge on how to prepare the corpus for word embedding training.

Gensim, spacy and FastText are three great frameworks that allow you to quickly use word embeddings in your machine learning application. In addition, they also support the training of custom word embeddings.

3.3 Long Short Term Memory (LSTM)

3.3.1 Neural Networks

An artificial neural network is a layered structure of connected neurons, inspired by biological neural networks. It is not one algorithm but combinations of various algorithms which allows us to do complex operations on data.

3.3.2 Recurrent Neural Networks

It is a class of neural networks tailored to deal with temporal data. The neurons of RNN have a cell state/memory, and input is processed according to this internal state, which is achieved with the help of loops with in the neural network. There are recurring module(s) of *'tanh'* layers in RNNs that allow them to retain information. However, not for a long time, which is why we need LSTM models [10].

3.3.3 LSTM

It is special kind of recurrent neural network that is capable of learning long term dependencies in data. This is achieved because the recurring module of the model has a combination of four layers interacting with each other [9].

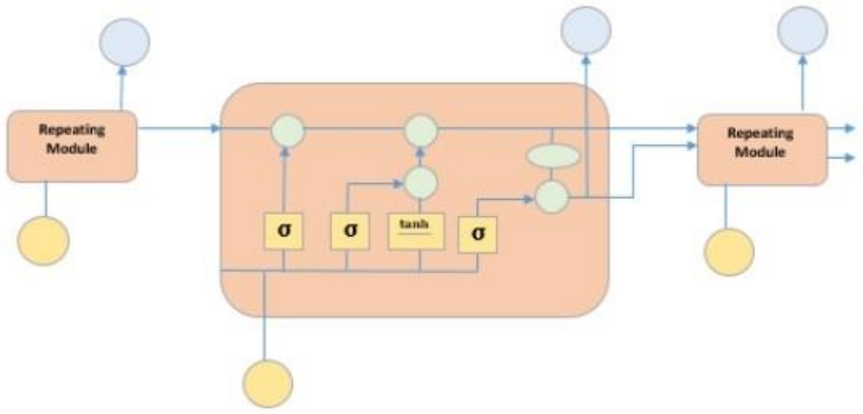
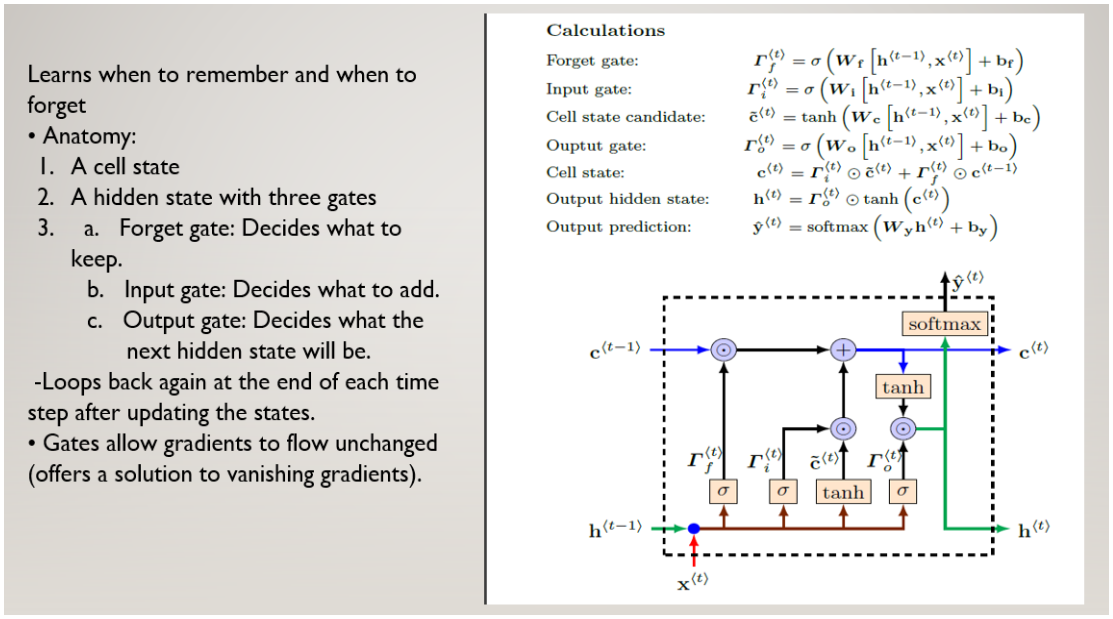


Figure 3.2: LSTM Layer

The picture above depicts four neural network layers in yellow boxes, point wise operators in green circles, input in yellow circles and cell state in blue circles. An LSTM module has a cell state and three gates which provides them with the power to selectively learn, unlearn or retain information from each of the units. The cell state in LSTM helps the information to flow through the units without being altered by allowing only a few linear interactions as evident from Figure 3.2.

Each unit has an input, output and a forget gate which can add or remove the information to the cell state. The forget gate decides which information from the previous cell state should be forgotten for which it uses a sigmoid function. The input gate controls the information flow to the current cell state using a point-wise multiplication operation of 'sigmoid' and 'tanh' respectively. Finally, the output gate decides which information should be passed on to the next hidden state as shown in Figure 3.3.


Learns when to remember and when to forget
- Anatomy:
  1. A cell state
  2. A hidden state with three gates
  3. a. Forget gate: Decides what to keep.
     b. Input gate: Decides what to add.
     c. Output gate: Decides what the next hidden state will be.

-Loops back again at the end of each time step after updating the states.
- Gates allow gradients to flow unchanged (offers a solution to vanishing gradients).

Calculations

Forget gate:	$\boldsymbol{\Gamma}_f^{\langle t\rangle} = \sigma\left(\boldsymbol{W}_\mathbf{f}\left[\mathbf{h}^{\langle t-1\rangle}, \mathbf{x}^{\langle t\rangle}\right] + \mathbf{b_f}\right)$
Input gate:	$\boldsymbol{\Gamma}_i^{\langle t\rangle} = \sigma\left(\boldsymbol{W}_\mathbf{i}\left[\mathbf{h}^{\langle t-1\rangle}, \mathbf{x}^{\langle t\rangle}\right] + \mathbf{b_i}\right)$
Cell state candidate:	$\tilde{\mathbf{c}}^{\langle t\rangle} = \tanh\left(\boldsymbol{W}_\mathbf{c}\left[\mathbf{h}^{\langle t-1\rangle}, \mathbf{x}^{\langle t\rangle}\right] + \mathbf{b_c}\right)$
Ouptut gate:	$\boldsymbol{\Gamma}_o^{\langle t\rangle} = \sigma\left(\boldsymbol{W}_\mathbf{o}\left[\mathbf{h}^{\langle t-1\rangle}, \mathbf{x}^{\langle t\rangle}\right] + \mathbf{b_o}\right)$
Cell state:	$\mathbf{c}^{\langle t\rangle} = \boldsymbol{\Gamma}_i^{\langle t\rangle} \odot \tilde{\mathbf{c}}^{\langle t\rangle} + \boldsymbol{\Gamma}_f^{\langle t\rangle} \odot \mathbf{c}^{\langle t-1\rangle}$
Output hidden state:	$\mathbf{h}^{\langle t\rangle} = \boldsymbol{\Gamma}_o^{\langle t\rangle} \odot \tanh\left(\mathbf{c}^{\langle t\rangle}\right)$
Output prediction:	$\hat{\mathbf{y}}^{\langle t\rangle} = \text{softmax}\left(\boldsymbol{W}_\mathbf{y}\mathbf{h}^{\langle t\rangle} + \mathbf{b_y}\right)$

$\hat{\mathbf{y}}^{\langle t\rangle}$
softmax
$\mathbf{c}^{\langle t-1\rangle}$
$\mathbf{c}^{\langle t\rangle}$
tanh
$\boldsymbol{\Gamma}_f^{\langle t\rangle}$
$\boldsymbol{\Gamma}_i^{\langle t\rangle}$
$\tilde{\mathbf{c}}^{\langle t\rangle}$
$\boldsymbol{\Gamma}_o^{\langle t\rangle}$
σ
σ
tanh
σ
$\mathbf{h}^{\langle t-1\rangle}$
$\mathbf{h}^{\langle t\rangle}$
$\mathbf{x}^{\langle t\rangle}$

Figure 3.3: Architecture of LSTM Layer

3.4 Attention Models(LSTM)

A neural network is considered to be an effort to mimic human brain actions in a simplified manner. Attention Mechanism is also an attempt to implement the same action of selectively concentrating on a few relevant things, while ignoring others in deep neural networks. The attention mechanism emerged as an improvement over the encoder decoder-based neural machine translation system in natural language processing (NLP). Later, this mechanism, or its variants, was used in other applications, including computer vision, speech processing, etc. [25].

Before Bahdanau et al proposed the first Attention model in 2015, neural machine translation was based on encoder-decoder RNNs/LSTMs. Both encoder and decoder are stacks of LSTM/RNN units. In short, there are two RNNs/LSTMs. One we call the encoder – this reads the input sentence and tries to make sense of it, before summarizing it. It passes the summary (context vector) to the decoder which translates the input sentence by just seeing it.

The main drawback of this approach is evident. If the encoder makes a bad summary, the translation will also be bad. And indeed it has been observed that the encoder creates a bad summary when it tries to understand longer sentences. It is called the *long-range dependency problem of RNN/LSTMs*.

RNNs cannot remember longer sentences and sequences due to the vanishing/exploding gradient problem. It can remember the parts which it has just seen. Even *Cho et al (2014)*, who proposed the encoder-decoder network, demonstrated that *the performance of the encoder-decoder network degrades rapidly as the length of the input sentence increases.*

Although an LSTM is supposed to capture the long-range dependency better than the RNN, it tends to become forgetful in specific cases. Another problem is that there is no way to give more importance to some of the input words compared to others while translating the sentence.

Now, let’s say, we want to predict the next word in a sentence, and its context is located a few words back. Here’s an example – *“Despite originally being from Uttar Pradesh, as he was brought up in Bengal, he is more comfortable in Bengali”*. In these groups of sentences, if we want to predict the word *“Bengali”*, the phrase *“brought up”* and *“Bengal”*- these two should be given more weight while predicting it. And although *Uttar Pradesh* is another state’s name, it should be “ignored”.

So is there any way we can keep all the relevant information in the input sentences intact while creating the context vector?

Bahdanau et al (2015) came up with a simple but elegant idea where they suggested that not only can all the input words be taken into account in the context vector, but relative importance should also be given to each one of them.

So, whenever the proposed model generates a sentence, it searches for a set of positions in the encoder hidden states where the most relevant information is available. This idea is called ‘Attention’.

3.5 Transformer Models

The Transformer architecture follows an encoder-decoder structure, but does not rely on recurrence and convolutions in order to generate an output.

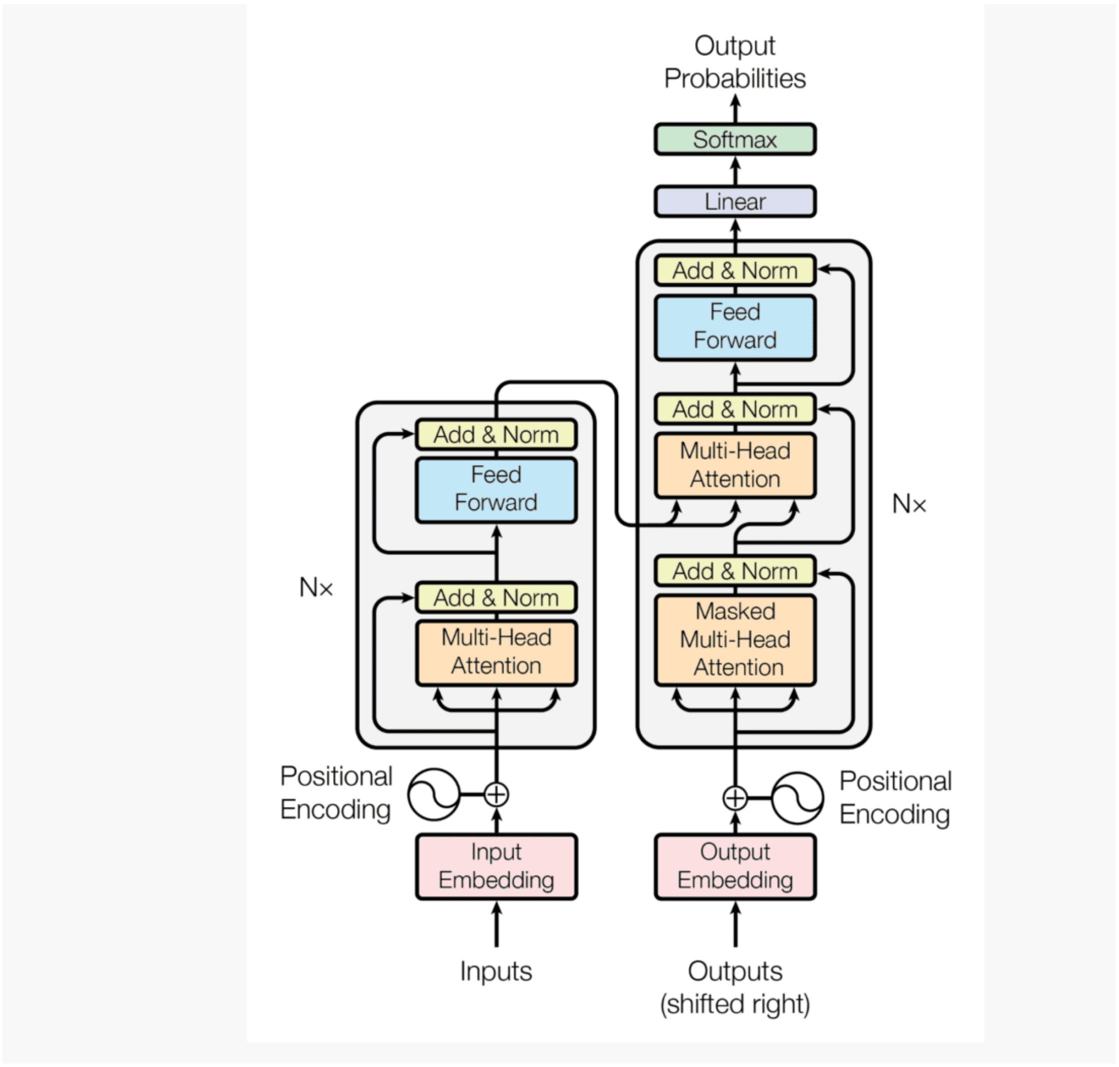


Figure 3.4: The Encoder-Decoder Structure of the Transformer Architecture

In a nutshell, the task of the encoder, on the left half of the Transformer architecture, is to map an input sequence to a sequence of continuous representations, which is then fed into a decoder as shown in Figure 3.4.

The decoder, on the right half of the architecture, receives the output of the encoder together with the decoder output at the previous time step, to generate an output sequence [26].

The Transformer model runs as follows:

1. Each word forming an input sequence is transformed into a dmodel-dimensional embedding vector.

2. Each embedding vector representing an input word is augmented by summing it (element-wise) to a positional encoding vector of the same dmodel length, hence introducing positional information into the input.

3. The augmented embedding vectors are fed into the encoder block, consisting of the two sublayers explained above. Since the encoder attends to all words in the input sequence, irrespective if they precede or succeed the word under consideration, then the Transformer encoder is *bidirectional*.

4. The decoder receives as input its own predicted output word at time-step, t–1.

5. The input to the decoder is also augmented by positional encoding, in the same manner as this is done on the encoder side.

6. The augmented decoder input is fed into the three sublayers comprising the decoder block explained above. Masking is applied in the first sublayer, in order to stop the decoder from attending to succeeding words. At the second sublayer, the decoder also receives the output of the encoder, which now allows the decoder to attend to all of the words in the input sequence.

7. The output of the decoder finally passes through a fully connected layer, followed by a softmax layer, to generate a prediction for the next word of the output sequence.

3.5.1 Transformer Models in NLP

The Transformer model structure has largely replaced other NLP model implementations such as RNNs [22]. Current SOTA NLP models use the Transformer architecture in part or as a whole**.** The GPT model only uses the decoder of the Transformer structure (unidirectional) [23] , while BERT is based on the Transformer encoder (bidirectional) [4]. T5 utilizes an encoder-decoder Transformer structure very similar to the original implementation [22]. These general architectures also differ in the number and dimension of the elements that comprise an encoder or decoder (i.e., the number of layers, the hidden size, and the number of self-attention heads they employ [24]).

3.5.1.1 BERT and DistilBERT and RoBERTa

BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. It is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context. As a result, the pre-trained BERT model can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of NLP tasks [11] [12].

The major features of BERT are as follows:

- BERT is based on the Transformer architecture.
- BERT is pre-trained on a large corpus of unlabelled text including the entire Wikipedia (that's 2,500 million words!) and Book Corpus (800 million words). This **pre-training** step is half the magic behind BERT's success. This is because as we train a model on a large text corpus, our model starts to pick up the deeper and intimate understandings of how the language works. This knowledge is the **swiss army knife** that is useful for almost any NLP task.
- BERT is a **"deeply bidirectional"** model. Bidirectional means that BERT learns information from both the left and the right side of a token's context during the training phase.
- The bidirectionality of a model is important for truly understanding the meaning of a language.

Let's see an example to illustrate this. There are two sentences in this example and both of them involve the word "bank":

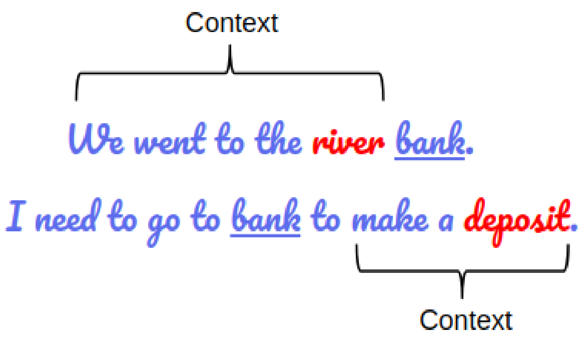

Context
We went to the river bank.
I need to go to bank to make a deposit.
Context

Figure 3.5: BERT captures both the left and right context

If we try to predict the nature of the word "bank" by only taking either the left or the right context, then we will be making an error in at least one of the two given examples.

One way to deal with this is to consider both the left and the right context before making a prediction. That's exactly what BERT does! We will see later in the article how this is achieved.

The most impressive aspect of BERT. We can fine-tune it by adding just a couple of additional output layers to create state-of-the-art models for a variety of NLP tasks. So, the new approach to solving NLP tasks became a 2-step process:

1. Train a language model on a large unlabelled text corpus (unsupervised or semi-supervised)
2. Fine-tune this large model to specific NLP tasks to utilize the large repository of knowledge this model has gained (supervised)

With that context, let's understand how BERT takes over from here to build a model that will become a benchmark of excellence in NLP for a long time [8].

BERT's Architecture

Let's look a bit closely at BERT and understand why it is such an effective method to model language. We've already seen what BERT can do earlier – but how does it do it? We'll answer this pertinent question in this section [7].

The BERT architecture builds on top of Transformer. We currently have two variants available:

- *BERT Base*: 12 layers (transformer blocks), 12 attention heads, and 110 million parameters
- *BERT Large*: 24 layers (transformer blocks), 16 attention heads and, 340 million parameters

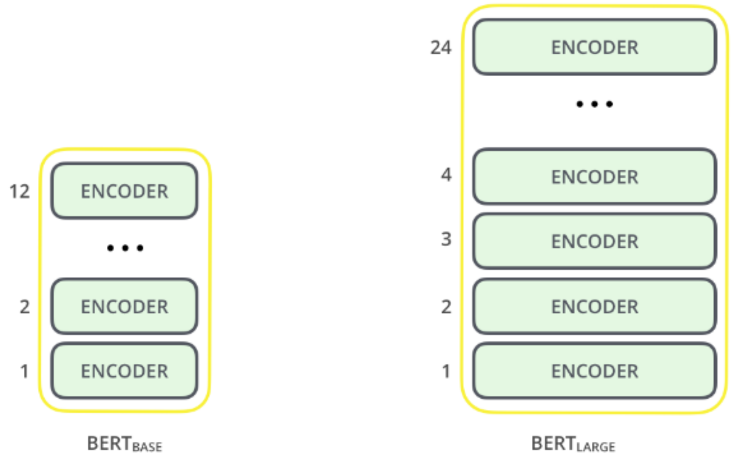


Figure 3.6: BERT Architecture

The BERT Base architecture has the same model size as OpenAI's GPT for comparison purposes. All of these Transformer layers are **Encoder**-only blocks.

Text Preprocessing

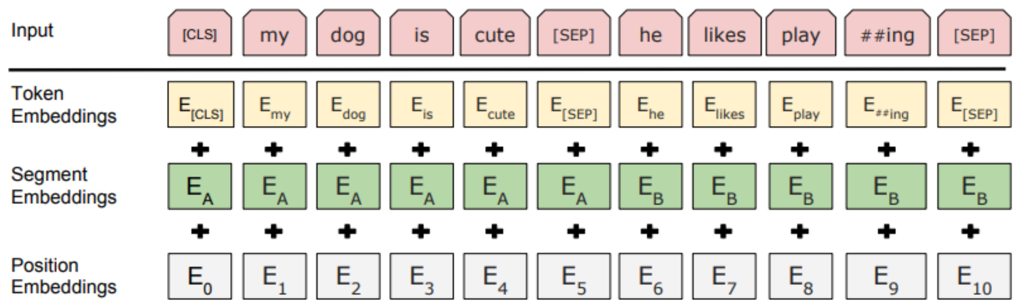

Input
[CLS]
my
dog
is
cute
[SEP]
he
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} E_{dog} E_{is} E_{cute} $E_{[SEP]}$ E_{he} E_{likes} E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Segment Embeddings
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Position Embeddings
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

Figure 3.7: Word Embedding

The developers behind BERT have added a specific set of rules to represent the input text for the model. Many of these are creative design choices that make the model even better.

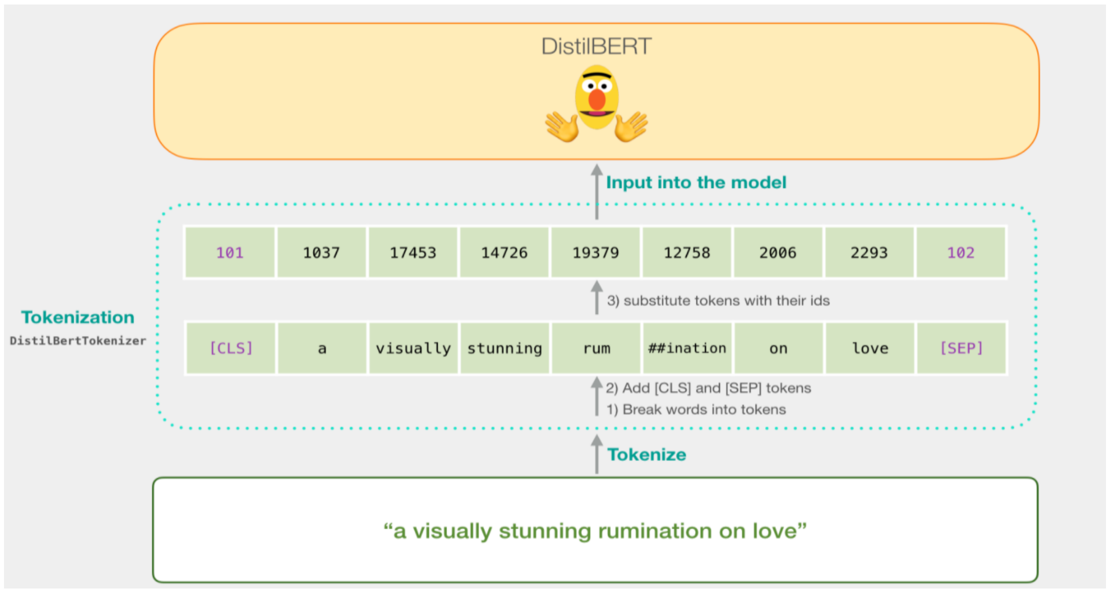


Figure 3.8: Process of Tokenization in DistilBERT

For starters, every input embedding is a combination of 3 embeddings:

1. **Position Embeddings**: BERT learns and uses positional embeddings to express the position of words in a sentence. These are added to overcome the limitation of

Transformer which, unlike an RNN, is not able to capture “sequence” or “order” information

2. **Segment Embeddings**: BERT can also take sentence pairs as inputs for tasks (Question-Answering). That’s why it learns a unique embedding for the first and the second sentences to help the model distinguish between them. In the above example, all the tokens marked as EA belong to sentence A (and similarly for EB)
3. **Token Embeddings**: These are the embeddings learned for the specific token from the WordPiece token vocabulary

For a given token, its input representation is constructed by summing the corresponding token, segment, and position embeddings. Such a comprehensive embedding scheme contains a lot of useful information for the model.

These combinations of preprocessing steps make BERT so versatile. This implies that without making any major change in the model’s architecture, we can easily train it on multiple kinds of NLP tasks [12].

BERT is pre-trained on two NLP tasks:

- Masked Language Modeling
- Next Sentence Prediction

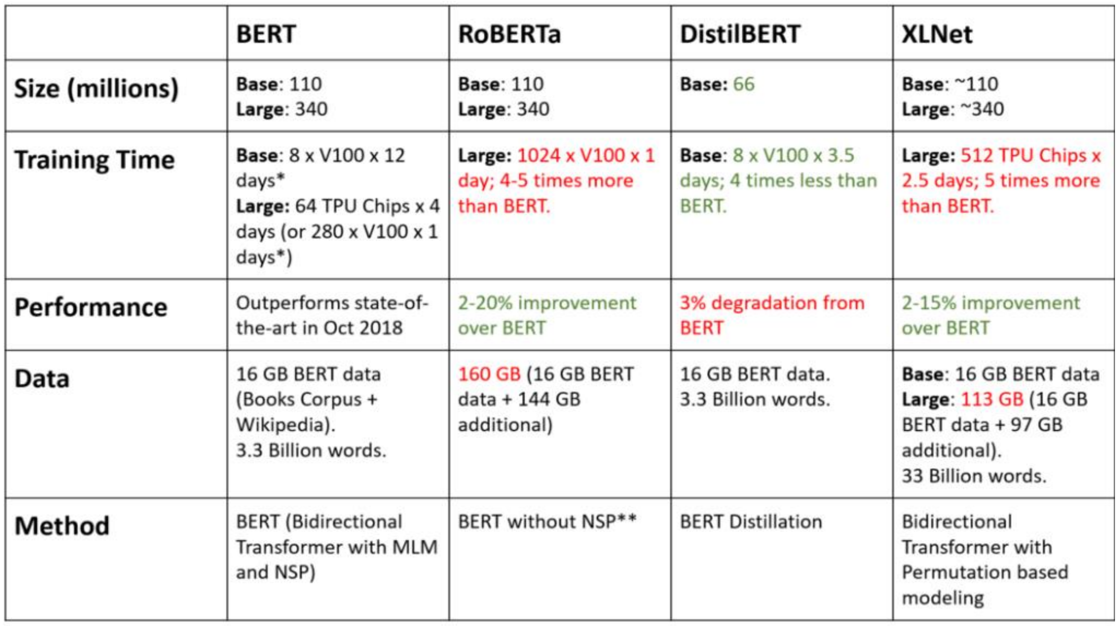
BERT has inspired great interest in the field of NLP, especially the application of the Transformer for NLP tasks. This has led to a spurt in the number of research labs and organizations that started experimenting with different aspects of pre-training, transformers and fine-tuning.

Many of these projects outperformed BERT on multiple NLP tasks. Some of the most interesting developments were RoBERTa, which was Facebook AI’s improvement over BERT and DistilBERT, which is a compact and faster version of BERT [15] [16].

To improve the training procedure, RoBERTa removes the Next Sentence Prediction (NSP) task from BERT’s pre-training and introduces dynamic masking so that the masked token changes during the training epochs. Larger batch-training sizes were also found to be more useful in the training procedure [17].

Importantly, RoBERTa uses 160 GB of text for pre-training, including 16GB of Books Corpus and English Wikipedia used in BERT. The additional data included CommonCrawl News dataset (63 million articles, 76 GB), Web text corpus (38 GB) and Stories from Common Crawl (31 GB). As a result, RoBERTa outperforms both BERT and XLNet on GLUE benchmark results [18].

Table 3.1: Comparison of BERT and recent improvements over it



	BERT	**RoBERTa**	**DistilBERT**	**XLNet**
Size (millions)	**Base**: 110 **Large**: 340	**Base**: 110 **Large**: 340	**Base:** 66	**Base**: ~110 **Large**: ~340
Training Time	**Base**: 8 x V100 x 12 days* **Large:** 64 TPU Chips x 4 days (or 280 x V100 x 1 days*)	**Large:** 1024 x V100 x 1 day; 4-5 times more than BERT.	**Base**: 8 x V100 x 3.5 days; 4 times less than BERT.	**Large:** 512 TPU Chips x 2.5 days; 5 times more than BERT.
Performance	Outperforms state-of-the-art in Oct 2018	2-20% improvement over BERT	3% degradation from BERT	2-15% improvement over BERT
Data	16 GB BERT data (Books Corpus + Wikipedia). 3.3 Billion words.	160 GB (16 GB BERT data + 144 GB additional)	16 GB BERT data. 3.3 Billion words.	**Base**: 16 GB BERT data **Large**: 113 GB (16 GB BERT data + 97 GB additional). 33 Billion words.
Method	BERT (Bidirectional Transformer with MLM and NSP)	BERT without NSP**	BERT Distillation	Bidirectional Transformer with Permutation based modeling

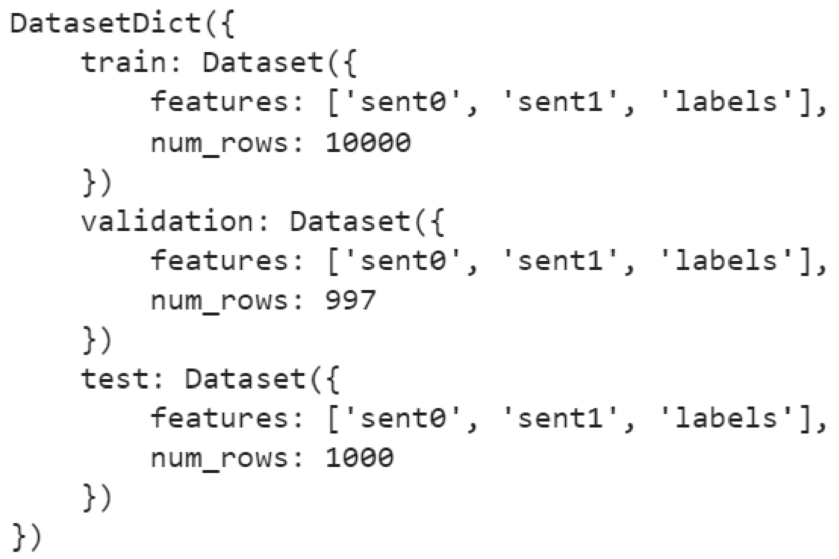
CHAPTER 4

METHODOLOGY

4.1. Exploratory Data Analysis

The dataset is provided alongside the research paper. More about data collection can be found in section 1.2. We perform exploratory data analysis on the 3 sets of data provided namely, Train, Test and Validation sets.

We will use the Huggingface Datasets library to load the data. This can be easily done with the functions *load_datas*et. The dataset object itself is *DatasetDict*, which contains one key for the training, validation and test set as evident from
Figure 4.1.



```
DatasetDict({
    train: Dataset({
        features: ['sent0', 'sent1', 'labels'],
        num_rows: 10000
    })
    validation: Dataset({
        features: ['sent0', 'sent1', 'labels'],
        num_rows: 997
    })
    test: Dataset({
        features: ['sent0', 'sent1', 'labels'],
        num_rows: 1000
    })
})
```

Figure 4.1: Dataset object which will be further used for model training

Using *Pandas* library, we find the distribution of data across the class labels 0 and 1 respectively. Figure 11 shows that the data is almost equally distributed among both the classes (i.e. labels=1 and labels=0), therefore the dataset is balanced and there is no chance of overfitting.

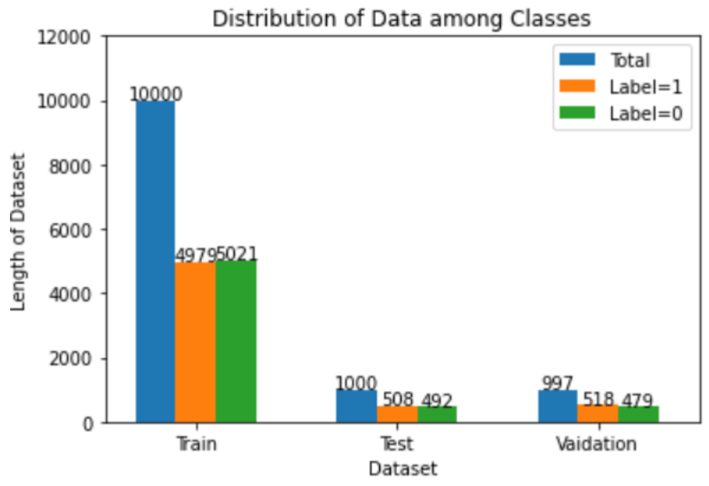


Figure 4.2: Frequency of Class Labels in Train, Test and Validation Datasets

4.2. Choosing Best Classification Model:

For this purpose, we did a comparative analysis of three models after transfer learning on our dataset:

1. Sequence-to-Sequence Model
2. DistilBERT
3. RoBERTa

4.2.1. Sequence-to-Sequence Model

We used sequence model having a single Bi-Directional LSTM layer having a Single Dense Layer on top of it for prediction.

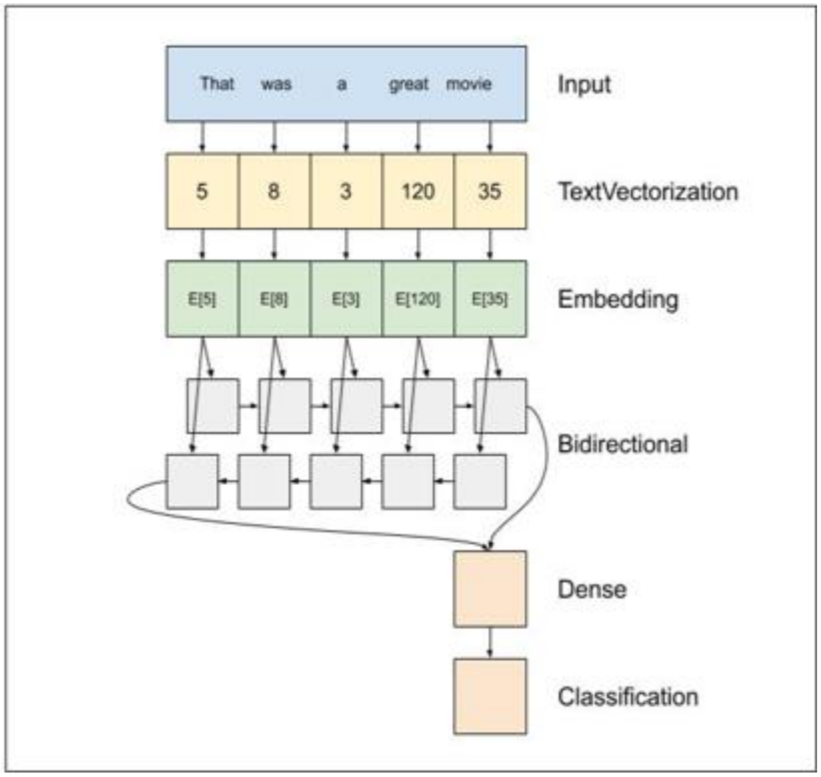


Figure 4.3: Bidirectional LSTM with Dense Layer Architecture

4.2.1.1. Model Training

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 64)]              0

 embedding (Embedding)       (None, 64, 128)           1764608

 lstm (LSTM)                 (None, 32)                20608

 batch_normalization (BatchN  (None, 32)               128
 ormalization)

 dropout (Dropout)           (None, 32)                0

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 1,785,377
Trainable params: 1,785,313
Non-trainable params: 64
_________________________________________________________________
```

Figure 4.4: Bidirectional LSTM model training

4.2.1.2. Results

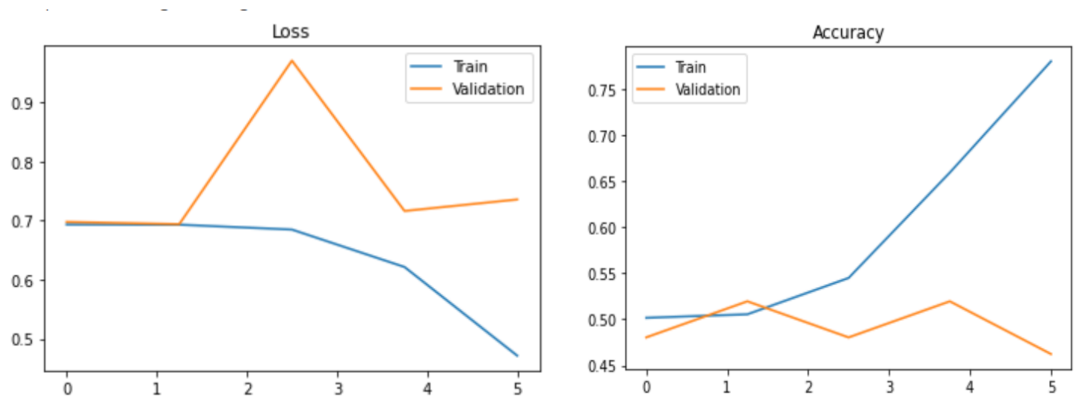


Figure 4.5: Bidirectional LSTM model performance graphs

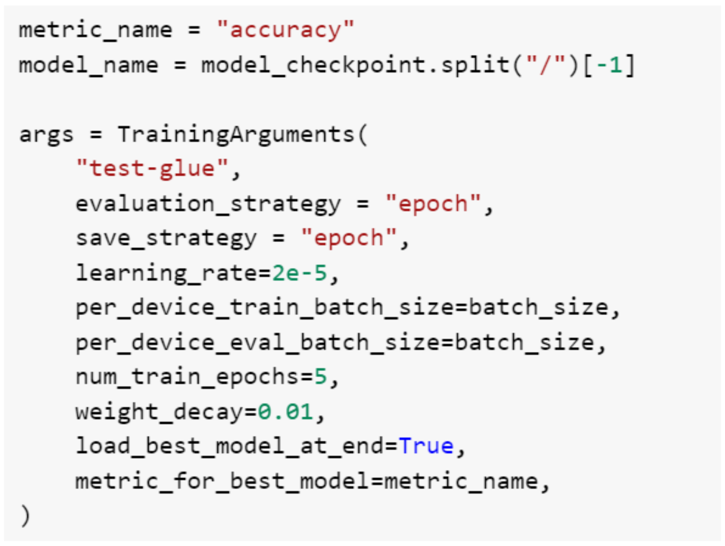
From the above graph we can see that this kind of Sequence model tends to overfit on the training data and gives low accuracy on validation set. We will try to tackle this issue using transformer models

4.2.2. Transformer Models - DistilBERT

State-of-the-art language models trained over large texts as our baselines, assuming that common sense knowledge is encoded over texts, having a single feed forward layer on top of the transformer language model having two neurons and softmax as activation function. Both the models will output **0** if the first sentence makes sense and **1** if the second sentence makes sense.

Since our task is about sentence classification, we use the *AutoModelForSequenceClassification* class. Like with the tokenizer, the *from_pretrained* method will download and cache the model for us. We will also have to specify the number of labels for our problem which is 2.

To instantiate a Trainer, we will need to define two more things. The most important is the *TrainingArguments*, which is a class that contains all the attributes to customize the training. It requires one folder name, which will be used to save the checkpoints of the model, and all other arguments are optional:

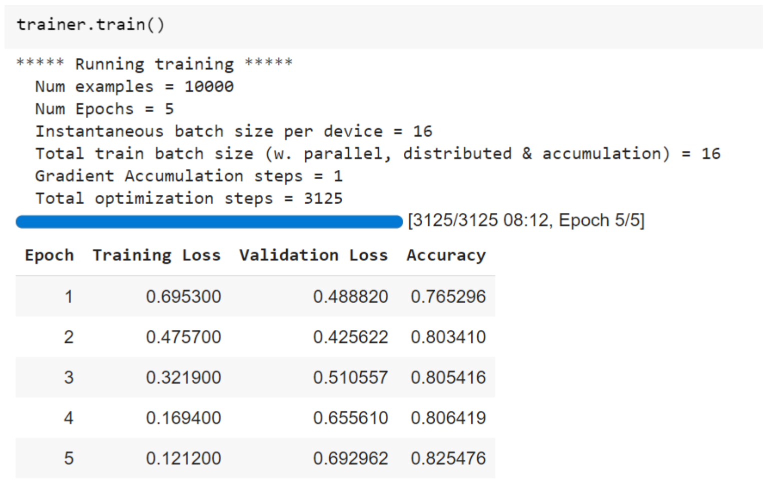


```
metric_name = "accuracy"
model_name = model_checkpoint.split("/")[-1]

args = TrainingArguments(
    "test-glue",
    evaluation_strategy = "epoch",
    save_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=5,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model=metric_name,
)
```

Figure 4.6: DistilBERT Training Arguments

We can now fine-tune our model by just calling the train method:



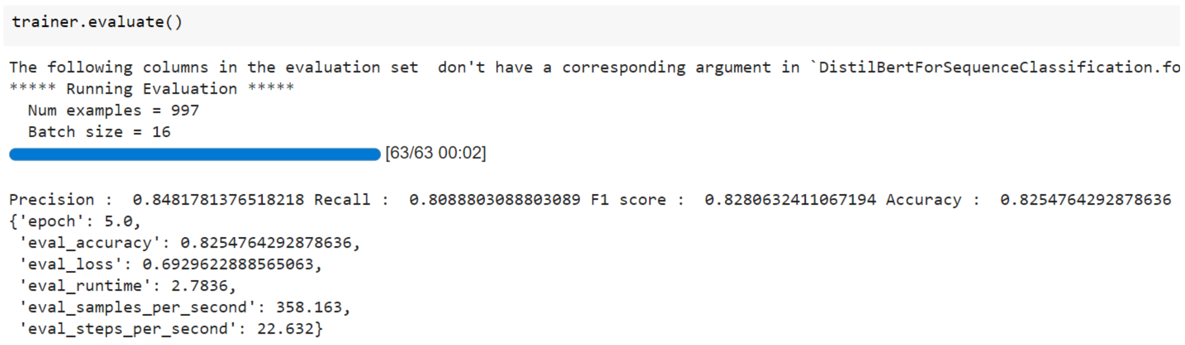
```
trainer.train()

***** Running training *****
  Num examples = 10000
  Num Epochs = 5
  Instantaneous batch size per device = 16
  Total train batch size (w. parallel, distributed & accumulation) = 16
  Gradient Accumulation steps = 1
  Total optimization steps = 3125
[3125/3125 08:12, Epoch 5/5]
```

Epoch	Training Loss	Validation Loss	Accuracy
1	0.695300	0.488820	0.765296
2	0.475700	0.425622	0.803410
3	0.321900	0.510557	0.805416
4	0.169400	0.655610	0.806419
5	0.121200	0.692962	0.825476

Figure 4.7: Fine-Tuning DistilBERT

We can check with the evaluate method that our Trainer did reload the best model properly (if it was not the last one)



```
trainer.evaluate()

The following columns in the evaluation set  don't have a corresponding argument in `DistilBertForSequenceClassification.fo
***** Running Evaluation *****
  Num examples = 997
  Batch size = 16
[63/63 00:02]

Precision :  0.8481781376518218 Recall :  0.8088803088803089 F1 score :  0.8280632411067194 Accuracy :  0.8254764292878636
{'epoch': 5.0,
 'eval_accuracy': 0.8254764292878636,
 'eval_loss': 0.6929622888565063,
 'eval_runtime': 2.7836,
 'eval_samples_per_second': 358.163,
 'eval_steps_per_second': 22.632}
```

Figure 4.8: Final Best Trained DistilBERT Model

4.2.3. Transformer Models – RoBERTa

4.2.3.1. Preprocessing the Data

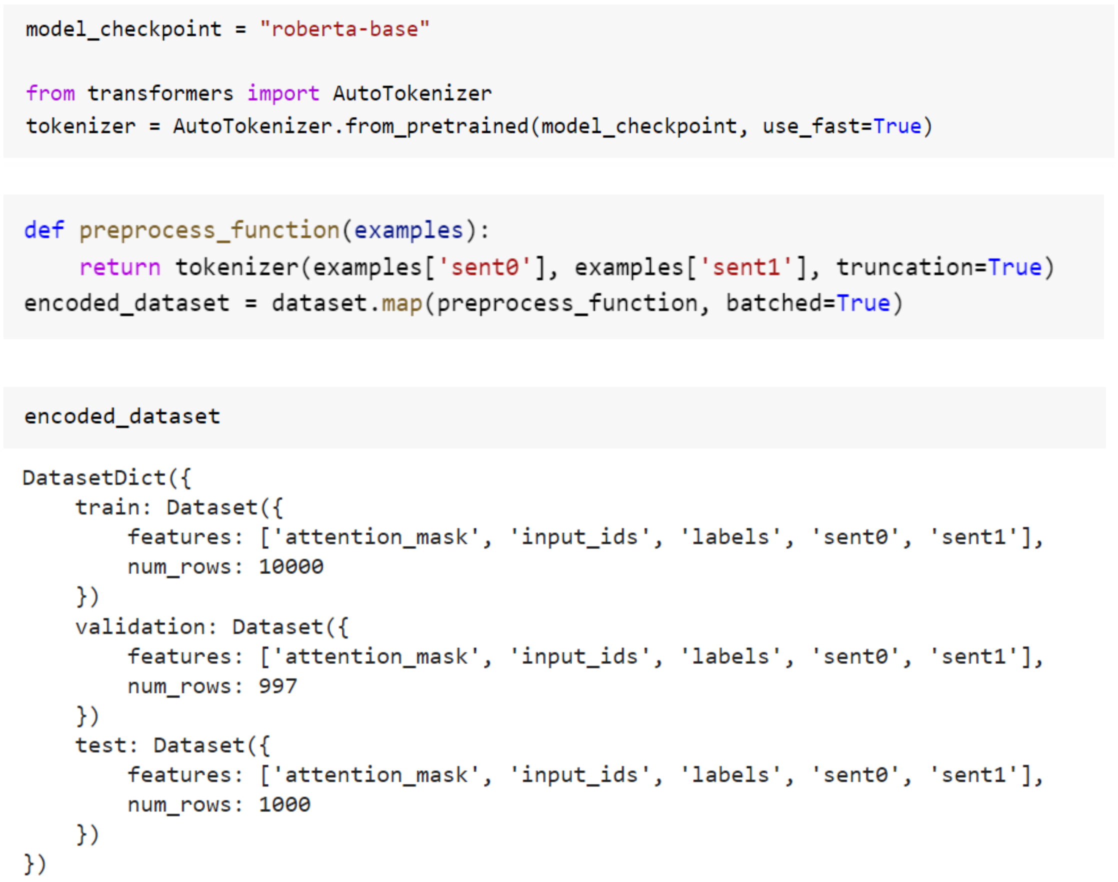
Before we can feed those texts to our model, we need to preprocess them. This is done by HuggingFace Transformers Tokenizer which will (as the name indicates) tokenize the inputs (including converting the tokens to their corresponding IDs in the pre-trained vocabulary) and put it in a format the model expects, as well as generate the other inputs that model requires.

To do all of this, we instantiate our tokenizer with the *AutoTokenizer.from_pretrained* method, which will ensure:

- we get a tokenizer that corresponds to the model architecture we want to use,
- we download the vocabulary used when pre-training this specific checkpoint.

That vocabulary will be cached, so it's not downloaded again the next time we run the cell. We pass along *use_fast=True* to the call above to use one of the fast tokenizers (backed by Rust) from the *HuggingFace* Tokenizers library. Those fast tokenizers are available for almost all models, but if you got an error with the previous call, remove that argument.

We can them write the function that will preprocess our samples. We just feed them to the tokenizer with the argument truncation=True. This will ensure that an input longer that what the model selected can handle will be truncated to the maximum length accepted by the model. To apply this function on all the sentences (or pairs of sentences) in our dataset, we just use the map method of our dataset object we created earlier. This will apply the function on all the elements of all the splits in dataset, so our training, validation and testing data will be preprocessed in one single command.



```
model_checkpoint = "roberta-base"

from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint, use_fast=True)
```

```
def preprocess_function(examples):
    return tokenizer(examples['sent0'], examples['sent1'], truncation=True)
encoded_dataset = dataset.map(preprocess_function, batched=True)
```

```
encoded_dataset
```

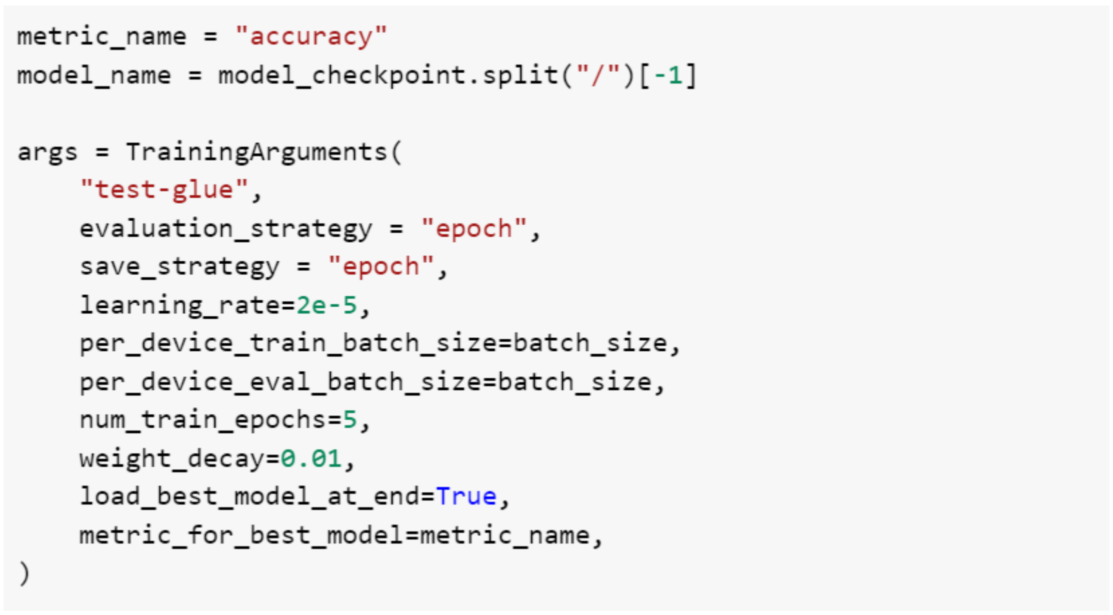
```
DatasetDict({
    train: Dataset({
        features: ['attention_mask', 'input_ids', 'labels', 'sent0', 'sent1'],
        num_rows: 10000
    })
    validation: Dataset({
        features: ['attention_mask', 'input_ids', 'labels', 'sent0', 'sent1'],
        num_rows: 997
    })
    test: Dataset({
        features: ['attention_mask', 'input_ids', 'labels', 'sent0', 'sent1'],
        num_rows: 1000
    })
})
```

Figure 4.9: Preprocessing Dataset- Tokenizing and Encoding

4.2.3.2. Fine-Tuning The Model

Now that our data is ready, we can download the pre-trained model and fine-tune it. Since our task is about sentence classification, we use the AutoModelForSequenceClassification class. Like with the tokenizer, the from_pretrained method will download and cache the model for us. We will also have to specify the number of labels for our problem which is 2.

To instantiate a Trainer, we will need to define two more things. The most important is the *TrainingArguments*, which is a class that contains all the attributes to customize the training. It requires one folder name, which will be used to save the checkpoints of the model, and all other arguments are optional.



```
metric_name = "accuracy"
model_name = model_checkpoint.split("/")[-1]

args = TrainingArguments(
    "test-glue",
    evaluation_strategy = "epoch",
    save_strategy = "epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size,
    num_train_epochs=5,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model=metric_name,
)
```

Figure 4.10: RoBERTa Training Arguments

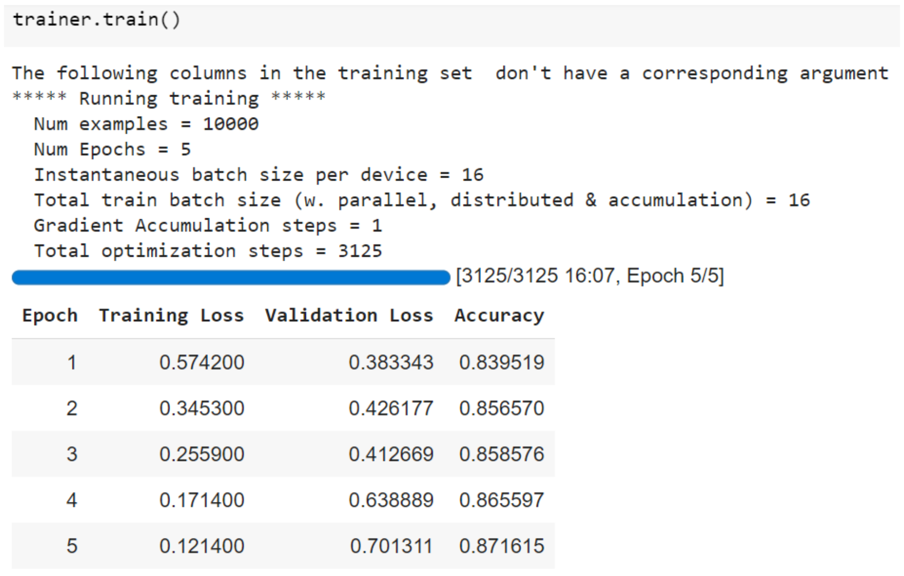
Here we set the evaluation to be done at the end of each epoch, tweak the learning rate, use the *batch_size* defined at the top of the notebook and customize the number of epochs for training, as well as the weight decay. Since the best model might not be the one at the end of training, we ask the Trainer to load the best model it saved (according to accuracy) at the end of training.

The last thing to define for our Trainer is how to compute the metrics from the predictions. We need to define a function for this, which will just use the metric we loaded earlier, the only preprocessing we have to do is to take the argmax of our predicted logics.

Then we just need to pass all of this along with our datasets to the Trainer.

You might wonder why we pass along the tokenizer when we already preprocessed our data. This is because we will use it once last time to make all the samples we gather the same length by applying padding, which requires knowing the model's preferences regarding padding (to the left or right? with which token?). The tokenizer has a pad method that will do all of this right for us, and the Trainer will use it. You can customize this part by defining and passing your own *data_collator* which will receive the samples like the dictionaries seen above and will need to return a dictionary of tensors. We can now fine-tune our model by just calling the train method.

```
trainer = Trainer(
    model,
    args,
    train_dataset=encoded_dataset["train"],
    eval_dataset=encoded_dataset["validation"],
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
```



```
trainer.train()
```

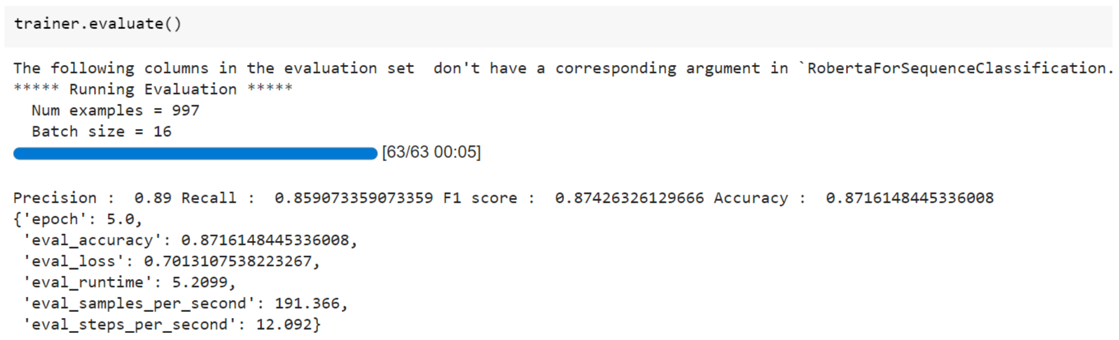
```
The following columns in the training set  don't have a corresponding argument
***** Running training *****
  Num examples = 10000
  Num Epochs = 5
  Instantaneous batch size per device = 16
  Total train batch size (w. parallel, distributed & accumulation) = 16
  Gradient Accumulation steps = 1
  Total optimization steps = 3125
```

[3125/3125 16:07, Epoch 5/5]

Epoch	Training Loss	Validation Loss	Accuracy
1	0.574200	0.383343	0.839519
2	0.345300	0.426177	0.856570
3	0.255900	0.412669	0.858576
4	0.171400	0.638889	0.865597
5	0.121400	0.701311	0.871615

Figure 4.11: RoBERTa Training and Evaluation

We can check with the evaluate method that our Trainer did reload the best model properly (if it was not the last one) [19]:



```
trainer.evaluate()

The following columns in the evaluation set  don't have a corresponding argument in `RobertaForSequenceClassification.
***** Running Evaluation *****
  Num examples = 997
  Batch size = 16
[63/63 00:05]

Precision :  0.89 Recall :  0.859073359073359 F1 score :  0.87426326129666 Accuracy :  0.8716148445336008
{'epoch': 5.0,
 'eval_accuracy': 0.8716148445336008,
 'eval_loss': 0.7013107538223267,
 'eval_runtime': 5.2099,
 'eval_samples_per_second': 191.366,
 'eval_steps_per_second': 12.092}
```

Figure 4.12: Final RoBERTa Model

4.2.4. Comparing Accuracy of Methods Used in Research Paper and Models Proposed Above

Table 4.1: Comparing Different Solutions

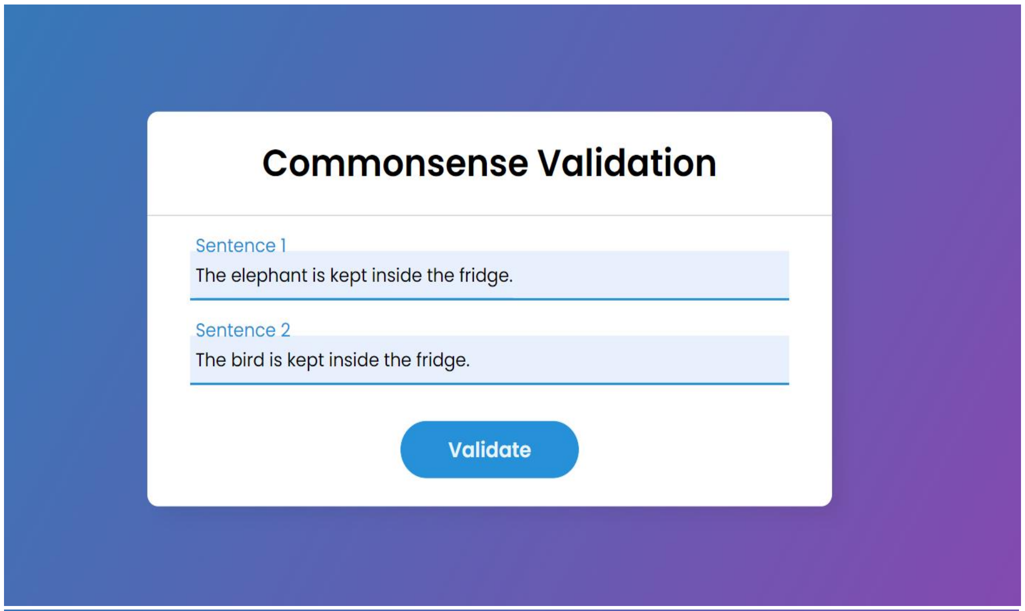
Model	**Sen-Making Accuracy**
Models Mentioned in Research Paper	
Random	50.00%
ELMo	69.40%
BERT	70.10%
Fine-tuned ELMo	***74.10%***
Human Performance	***99.10%***
Models Implemented by us	
Sequence-to-Sequence	52.00%
DistilBERT	***82.04%***
RoBERTa	***87.16%***

From the above comparison we find that RoBERTa model outperforms all other methods used in the research paper and by us. We have trained Sequence to Sequence, DistilBERT and RoBERTa models on train dataset and validated on validation dataset.

We have achieved an accuracy of 87.16% against the highest accuracy of 74.1% attained by the methodology used in research paper.

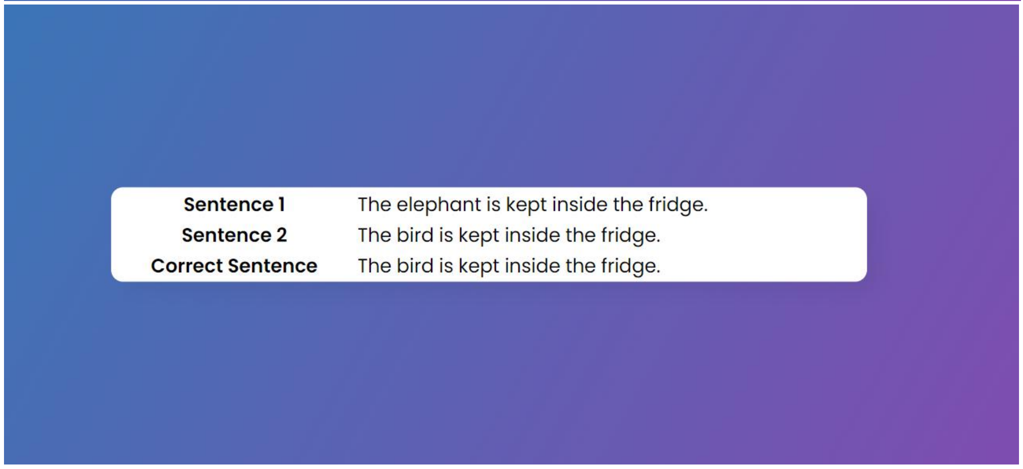
4.3. Building a Web Application

We save the Roberta model checkpoint in the folder. This can be directly used to test between the statements input through front-end after tokenization.



The elephant is kept inside the fridge.

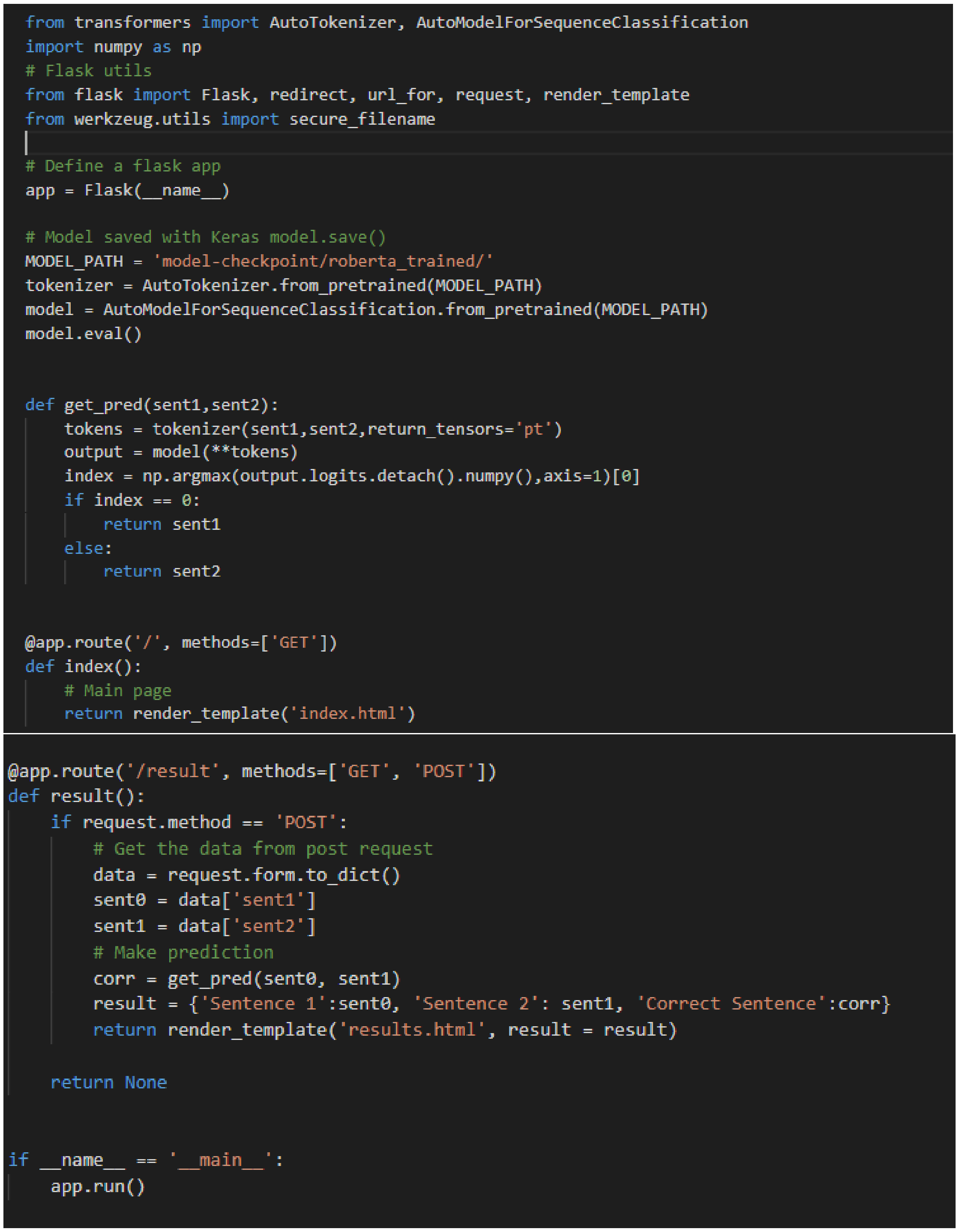
The bird is kept inside the fridge.



Sentence 1	The elephant is kept inside the fridge.
Sentence 2	The bird is kept inside the fridge.
Correct Sentence	The bird is kept inside the fridge.

Figure 4.13: Running the Web App – Homepage and Results

Supposedly, we input two statements as given in Figure 20 in the front page. The two sentences are supplied to the Flask backend. The Flask app code loads the pre-trained *Roberta_checkpoint* and *Autotokenizer*. The sentences are first tokenized and the embeddings of both sentences are supplied as parameters to the model. The model predicts the more sensible statement among the two which is then displayed in the result page of the web app [21].



```python
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import numpy as np
# Flask utils
from flask import Flask, redirect, url_for, request, render_template
from werkzeug.utils import secure_filename

# Define a flask app
app = Flask(__name__)

# Model saved with Keras model.save()
MODEL_PATH = 'model-checkpoint/roberta_trained/'
tokenizer = AutoTokenizer.from_pretrained(MODEL_PATH)
model = AutoModelForSequenceClassification.from_pretrained(MODEL_PATH)
model.eval()


def get_pred(sent1,sent2):
    tokens = tokenizer(sent1,sent2,return_tensors='pt')
    output = model(**tokens)
    index = np.argmax(output.logits.detach().numpy(),axis=1)[0]
    if index == 0:
        return sent1
    else:
        return sent2


@app.route('/', methods=['GET'])
def index():
    # Main page
    return render_template('index.html')
```

```python
@app.route('/result', methods=['GET', 'POST'])
def result():
    if request.method == 'POST':
        # Get the data from post request
        data = request.form.to_dict()
        sent0 = data['sent1']
        sent1 = data['sent2']
        # Make prediction
        corr = get_pred(sent0, sent1)
        result = {'Sentence 1':sent0, 'Sentence 2': sent1, 'Correct Sentence':corr}
        return render_template('results.html', result = result)

    return None


if __name__ == '__main__':
    app.run()
```

Figure 4.14: Flask code for Backend Processing

4.4. Results and Analysis

From Table 4.1, we can deduce that RoBERTa based transformer model gives highest accuracy of 87.16% among all other methods used by us and others mentioned in the research paper. After training and evaluating Bidirectional LSTM and DistilBERT models, we trained RoBERTa model on our dataset.

To improve the training procedure, RoBERTa removes the Next Sentence Prediction (NSP) task from BERT’s pre-training and introduces dynamic masking so that the masked token changes during the training epochs. Larger batch-training sizes were also found to be more useful in the training procedure.

Importantly, RoBERTa uses 160 GB of text for pre-training, including 16GB of Books Corpus and English Wikipedia used in BERT. The additional data included *CommonCrawl News dataset* (63 million articles, 76 GB), Web text corpus (38 GB) and Stories from Common Crawl (31 GB). As a result, RoBERTa outperforms both BERT and XLNet on GLUE benchmark results [14].

Since our dataset is balanced, ‘Accuracy’ is the sole metric which can be used to compare various models and choose the best among them. We find that RoBERTa model outperforms all other methods and gives an accuracy of 87.16%.

CHAPTER 5

CONCLUSION and FUTURE SCOPE

5.1. Conclusion

Through the methodology proposed in Chapter 4, we have created a benchmark for directly evaluating whether a system has the capability of sense making and explanation, evaluating models trained over the large raw text as well as a common sense database on the test set.

We trained, validated and tested Bidirectional LSTM with single dense layer, Tranformer based DistilBERT and BERT optimized RoBERTa models on the datasets provided and compared the results with the methods used by the researchers as mentioned in the research paper.

On comparison, we found that RoBERTa based transformer model gives highest accuracy as compared to sequence based and other transformer models. However, the Human Performance is 99.1%. This shows that sense making remains a technical challenge for all models.

5.2. Future Scope

Recent research papers have achieved greater accuracy by training large and complex Knowledge Attention Graph models on the given dataset. Training of such complex models is beyond our scope due to lack of sufficient infrastructure [4].

Commonsense validation can be used in search engines and chatbots to provide improved user experience and detect garbage inputs in databases.

5.3. Programming Tools and Environments Used

- Exploratory Data Analysis - Pandas, Numpy, Matplotlib
- Model Training and Evaluation – Keras, Tensorflow, HuggingFace
- IDE - Google Colab, Visual Studio Code
- Web Development – Flask, HTML, CSS

REFERENCES

1. Wang, C., Liang, S., Jin, Y., Wang, Y., Zhu, X., & Zhang, Y. (2020). SemEval-2020 task 4: Commonsense validation and explanation. *arXiv preprint arXiv:2007.00236*.
2. Codalab Competition - CodaLab - Competition
3. Dataset - https://github.com/wangcunxiang/Sen-Making-and-Explanation
4. Knowledge-enhanced Graph Attention Network (KEGAT) - [2007.14200] ECNU-SenseMaker at SemEval-2020 Task 4: Leveraging Heterogeneous Knowledge Resources for Commonsense Validation and Explanation (arxiv.org)
5. Rationale-Inspired Natural Language Explanations with Commonsense - [2106.13876] Rationale-Inspired Natural Language Explanations with Commonsense (arxiv.org)
6. Word Embedding - Simple Tutorial on Word Embedding and Word2Vec | by Zafar Ali | Medium
7. A Visual Guide to Using BERT for the First Time – Jay Alammar – Visualizing machine learning one concept at a time. (jalammar.github.io)
8. The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. (jalammar.github.io)
9. Seq2Seq Model - https://www.analyticsvidhya.com/blog/2020/08/a-simple-introduction-to-sequence-to-sequence-models/
10. Sequence Models - https://cs230.stanford.edu/files/C5M3.pdf
11. BERT - https://towardsdatascience.com/keeping-up-with-the-berts-5b7beb92766
12. https://www.analyticsvidhya.com/blog/2021/06/why-and-how-to-use-bert-for-nlp-text-classification/
13. DistillBert - Smaller, faster, cheaper, lighter: Introducing DistilBERT, a distilled version of BERT | by Victor Sanh | HuggingFace | Medium
14. DistilBert - DistilBERT — transformers 2.11.0 documentation (huggingface.co)
15. BERT, DistilBERT, ROBERTA, XLNet - https://towardsdatascience.com/bert-roberta-distilbert-xlnet-which-one-to-use-3d5ab82ba5f8
16. BERT, DistilBERT, ROBERTA, XLNet - https://www.kdnuggets.com/2019/09/bert-roberta-distilbert-xlnet-one-use.html
17. https://towardsdatascience.com/to-distil-or-not-to-distil-bert-roberta-and-xlnet-c777ad92f8
18. https://datascience.stackexchange.com/questions/97310/what-is-the-difference-between-bert-and-roberta
19. Evaluating Roberta Model - https://github.com/pytorch/fairseq/issues/1324
20. Deep Learning Stanford University Course - CS230: Deep Learning | Autumn 2018 - YouTube
21. Making Web Apps using Flask - How to Deploy Machine Learning Models using Flask (with Code) (analyticsvidhya.com)
22. [1910.10683] Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (arxiv.org)
23. language_understanding_paper.pdf (openai.com)
24. [1810.04805] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (arxiv.org)
25. Brief Introduction to Attention Models | by Abhishek Singh | Towards Data Science
26. Transformers In NLP | State-Of-The-Art-Models (analyticsvidhya.com)
27. Transfer Learning - www.datacamp.com/community/tutorials/transfer-learning